Lab 2: Improve the system

Learning objective: Improve the efficiency of the system by designing an event-based online scheduler on top of the bare metal hardware.

Assessment instructions: You will be graded on a scale from 0-10. You will pass this assignment only if you gain at least 6 points and you have handed-in your code and presented the code to the TA. Note that the Challenge Questions are a bonus for your final grade and not for your current assignment. Namely, it does not contribute to the points that you need to be able to pass this assignment. Instead, the points you get from the challenge questions will be directly added to your final grade for the real-time system’s course.

Instructions for improving the efficiency

As you have noticed in the previous assignment, in the current design, the scheduler imposes a significant amount of overhead to the system (namely, the system spends a lot of time only on scheduling the tasks rather than executing them!). In the second assignment, you are expected to re-design the structure of the scheduler (namely, modify how and when the scheduler is activated) to build an event-based online scheduler.

Recall. An event-based scheduler is the one that will be called (activated) only at certain events, e.g., job-release event and job-completion event. An online scheduler takes its decisions at runtime based on the current status of the system.

Properties of our desired scheduler. The scheduler is called at most only once per job-release event and at most only once per job-completion event. For example, if the system has 2 tasks $\tau_1 = (2, 7)$ and $\tau_2 = (8, 14)$ (where the numbers denote the execution time and period of each task, respectively), then the job-release events happen at times 0, 7, 14, 21, etc., and job-completion events happen at times 2, 9, 12, etc. In this example, the scheduler is called only at times 0, 2, 7, 9, and 12 (in the first hyperperiod). Figure 1 shows the schedule and the aforementioned events.

Hints on how to modify the code. As you may have noticed in Figure 1, in the desired system, timer interrupt is used to handle the job-release events so that there is no need to check if a new job is released at every timer tick (wasn’t that crazy in the legacy system?!). However, a single timer interrupt can only handle one interrupt at a time! Hence, you cannot assign more than one timer interrupt to your hardware timer (since you only have one hardware timer). What is the solution then?

Conceptually, you need to use a queue (e.g., an array) to keep track of the timer events that are going to happen in the future\(^1\). For example, at time 0 after both $\tau_1$ and $\tau_2$ released their jobs, the next timer event for $\tau_1$ will be at time 7 and for $\tau_2$ will be at 14. However, your hardware timer cannot handle more than one interrupt at a time. So you need to first set the timer to provide you an interrupt at time 7, and then set it again for time 14. Looks good?

After sorting out how the release events are handled, you need to call the scheduler again.

---

\(^1\)In terms of implementation, you may not really need an actual queue object, instead, you can add the extra information you need to store for the timer to the task data structure.
Both $\tau_1$ and $\tau_2$ are ready. The scheduler selects $\tau_1$ (e.g., assume a rate-monotonic scheduler).

$\tau_1$ has just arrived! Now both $\tau_1$ and $\tau_2$ are in the ready queue because $\tau_2$ has not yet been finished. The scheduler selects $\tau_1$.

$\tau_1$ finished. Scheduler is called. The scheduler selects $\tau_2$ since it is the only ready job.

$\tau_2$ finished. The scheduler is called, however, nothing is in the ready queue. The scheduler schedules the idle task (or does nothing).

• $\tau_1$ and $\tau_2$ are released and stored in the ready queue.
• For each released task, the next release time is updated: for $\tau_1$ it will be at time 7 and for $\tau_2$ at time 14.
• The next timer interrupt is set to 7 (the minimum between 7 and 14)
• Call the scheduler

Job-release timer interrupt

• $\tau_1$ is released and stored in the ready queue.
• Its next release time is updated to 14.
• The next timer interrupt is set to 14 (the minimum between 14 and 14)
• Call the scheduler

Figure 1. (i) The schedule of two periodic tasks $\tau_1$ and $\tau_2$ with periods 7 and 14 and execution times 2 and 8, respectively, using an online scheduling algorithm, (ii) the time instants at which the scheduler is called, and (iii) the time instants at which the timer-interrupt handler is called to release the new jobs of the tasks.
static void ExecuteTask (Taskp t)
{
    /* ------------------------ INSERT CODE HERE ------------------------ */
    t->Invoked++;
    t->Taskf(t->ExecutionTime); // execute task
    /* ------------------------ INSERT CODE HERE ------------------------ */
}

void Scheduler_P_FP (Task Tasks[])
{
    /* ------------------------ INSERT CODE HERE ------------------------ */

    /* Super simple, single task example */
    Taskp t = &Tasks[3];
    if (t->Activated != t->Invoked)
    {
        ExecuteTask(t);
    }
    /* End of example*/
    /* ------------------------ INSERT CODE HERE ------------------------ */
}

interrupt (TIMER0_VECTOR) TimerIntrpt (void)
{
    ContextSwitch();
    /* ------------------------ INSERT CODE HERE ------------------------ */

    /* Insert timer interrupt logic, what tasks are pending? */
    /* When should the next timer interrupt occur? Note: we only want interrupts at job releases */

    /* Super simple, single task example */
    Taskp t = &Tasks[3];
    t->NextRelease = t->Period; // set next release time
    t->Activated++;
    NextInterruptTime = t->NextRelease;
    /* End of example*/
    /* ------------------------ INSERT CODE HERE ------------------------ */

    TACCR0 = NextInterruptTime;

    CALL_SCHEDULER;
    ResumeContext();
}
What you should not change. In order to ensure that you focus only on the design of scheduler and task-activation mechanism, please try to not modify any files or functions other than `TimerIntrpt`, `ExecuteTask` and `Scheduler_P_FP`.

Hints on how to debug the code. Developing software comes with writing bugs. A common way of debugging is using `printf` to show variables values or following the execution path of your program. However, this is not supported in this simulator. Instead, you can use the output ports of the microcontroller to log variables or the show execution path. Example output is shown in figure 4.

The microcontroller has six output ports: `port_out1` to `port_out6`. Note that on `port_out5`, the three LEDs yellow, green and red are placed (inverted compared with the original). You don’t have to use these individual signals anymore for this assignment, you can use `port_out5` to show the LEDs. Use `port_out5` for toggling LEDs and use the other ports to log variable values. Note that these output ports are have eight bits. If you want to log a 16-bit variable, you should use two ports to visualize this.

To import the signals into the viewer, select the output ports, hold `shift` and drag + drop them intro the wave viewer. Then select `port_out5` and press `F3` to expand the signal to 8 individual outputs.

If you see that your simulation is not running up to the default 5 seconds, but gets stuck after say 1 ms, you probably dereferenced a wild pointer or accessed an array out of it’s bounds. So be careful when dereferencing pointers and array elements!

Before you start coding. To verify that your scheduler is correct, you have to compare the result with the correct schedule. So before you start coding, make a sketch of one hyperperiod of task set `Tst1` defined in `SchedTest.c`.

Figure 4. Use the output ports of the microcontroller for debugging
Expected deliverables.

- **(3.5 points)** Hand-in your source code (only the files that must run on top of the hardware simulator as a zip file via BrightSpace Assignment 2. During the demo, you will be asked to run the example that is provided with the code on your system for at least two hyperperiods. Be prepared for some modifications that is asked from you by the TA on the spot. For example, he may ask you to change some parameters to check if the code works nicely. For this lab you only have to implement the preemptive fixed point scheduler in `Scheduler_P_FP.c`, you can ignore the other scheduler source files.

- **(0.5 points)** Take a snapshot of the schedule for one hyperperiod of task set `Tst1` defined in `SchedTest.c`. Compare this with the sketch you drew up front. Write a short description to explain how the scheduler and timer interrupt handler work in your code (no more than 500 words).

- **(2 points)** Calculate the system overhead in one hyperperiod. To backup your answer, add a snapshot of your schedule and explain how much time is spent on anything expect executing the tasks (you do not have to consider idle time). Because measuring all scheduling invocations by measuring time in GTKWave is a tedious job, you can make use of `TimeTracking.c`. There you instrument the code to do measurements for you.

- **(1 point)** Does the method of overhead measurement provided in `TimeTracking.c` interfere with the program execution? Justify your answer in at most 100 words.

- **(2 points)** Plot the overhead (in percentages %) as a function of the number of tasks in your system (ranging from 1 task to 10 tasks). For this question, you can use the given task set in `SchedTest.c`, by defining `TstSweep`.

- **(1 points)** How does the chosen task set parameters (periods and execution times) influence the total overhead of the system in a hyperperiod? Justify your answer in at most 100 words.
Challenge questions for Lab 2 (not mandatory)

This part describes some challenges for those who enjoyed this assignment so far and are willing to take a bigger challenge to expand their experiences with practical aspects of real-time system design. If your response to this part is correct, you will directly receive your bonus points gained in this part on top of your final grade.

Challenge 1 (0.4 points for the final grade)

As you may have noticed, in the design proposed in Figure 1, each time the timer interrupt is called (namely, a job is released), you need to traverse through the list of tasks and update their next arrival time in case they are released. This is needed because at some points in time, there might be more than one task that release a job, e.g., at times 0 and 14 in our example. Moreover, you also need to find out when the next release event happens, hence, you need to find the minimum value among the next release times and set the timer to that value. In an inefficient implementation, you may traverse through the whole list of tasks to perform these two functions.

Goal. The goal of this challenge question is to reduce the number of times that you go through the whole list whenever the timer interrupt is called. More formally, we want to reduce the average overhead of handling job-release events.

Why should you care about your timer interrupt handler? Many real-time systems use harmonic periods\(^2\). For example, in Automotive industry\(^3\), AutoSAR standard forces the manufacturers to use the following set of periods: \{1, 2, 5, 10, 20, 50, 100, 200, 1000\}ms. As you can see, the task with the smallest period will release 1000 more jobs every second than the task with the largest period.

Moreover, automotive systems usually have a large set of tasks (e.g., a couple of hundred tasks). As a result, if your design requires traversing over the list of tasks each time the task with period 1ms releases a job, the system will be crashed down by the overhead (it is a deadly situation for an engineer)! Taking into account that you are working with a slow microprocessor, overheads of a bad implementation can easily kill your whole system\(^4\).

Instructions. To avoid being eaten alive by the overheads, you need to design a smart solution that takes advantage of the sparsity of period values to be able to reduce the average overhead per job release! More precisely, your design must try to avoid going through the whole list of tasks for every single job-release event in the timer interrupt handler.

One classic approach to reduce this overhead is to use Timer Wheel. You are free to search the Internet for finding suitable implementations to add to your timer interrupt service routine. However, perhaps, there might be even smarter solutions that take into account the fact that periods are harmonic. This will be up to you.

Deliverables.

- Hand-in your code in a separate zip file but at the same time you hand-in your main assignment for Lab 2. Be prepared to demo your code to the TA.
- Briefly explain your design and discuss why and how it reduces the number of iterations in the timer interrupt handler. Your answer can include figures/tables. It should not have more than 500 words.
- Draw a plot that compares the overheads of your new timer interrupt handler with your old one (that traverses through all tasks at each timer event).

---

\(^2\)In harmonic sets, every number divides all smaller numbers, for example, \{7, 14, 42\} are harmonic because \(14 = 2 \times 7\) and \(42 = 3 \times 14 = 5 \times 7\).


\(^4\)Figure 5 is from https://www.istockphoto.com/nl/vector/two-knights-jousting-gm471313717-17005832
Challenge 2 (0.2 points for the final grade)

As you may have experienced in the past, if you continue adding up values to a variable, it will eventually overflow. This can easily happen for your timer interrupt handler if you keep the next release time of a job as an absolute value.

Why should you care about overflows? Because they make your boss and your customers extremely angry! To observe the effect of overflow in your program, try to plot the schedule of 10 hyperperiods and see how your schedule becomes unexplainable! Never under-estimate overflows.

Goal. The goal of this challenge is to implement a solution in which there is no overflow for timer events. More formally, you will implement a solution that can schedule task sets for the rest of the life time of the system.

A possible solution approach. One possible solution would be to keep track of the time instants (for example, job-release times) in the form of "relative" values with respect to the beginning of the hyperperiod. This provides you the opportunity to reset the time values back to zero at the end of each hyperperiod and be prepared for the next one. By doing this, the first job of $\tau_1$ in Figure 1 will always be released at time 0 and the second job of $\tau_1$ will always be released at time 7 because you conceptually reset the time at the beginning of the hyperperiod.

Can you guess what is a pre-requisite for the soundness of this solution? Yes! The hyperperiod itself must be smaller than the time at which an integer variable in your system overflows (e.g., smaller than 65,535 clock ticks), otherwise you will get an overflow within your first hyperperiod.

If you have been tempted to use "mod" instruction$^5$ to ensure that every variable stays within the boundary of the hyperperiod, you should be aware of the overhead of calculating % in the MSP430 microcontroller. Hence, try to come up with a solution that does not need % operation.

The above-mentioned solution is one of the possible solutions. Feel free to come up with your own! ;-)

Deliverables.

- Hand-in your code for this challenge in a separate zip file but at the same time that you hand-in your main assignment for Lab 2. Be prepared to demo your code to the TA.
- Briefly explain your design and discuss how you have avoided overflows. Your answer can include figures/tables. It should not have more than 500 words.
- Draw a plot that shows your solution can schedule tasks for long duration of time (e.g., for 20 sec).

---

$^5$Just to remind you about modulo operator: $13 \% 5 = 3$