Real-time scheduling (single CPU)

1) [50 pts] Your system runs three periodic real-time tasks with the characteristics given below:

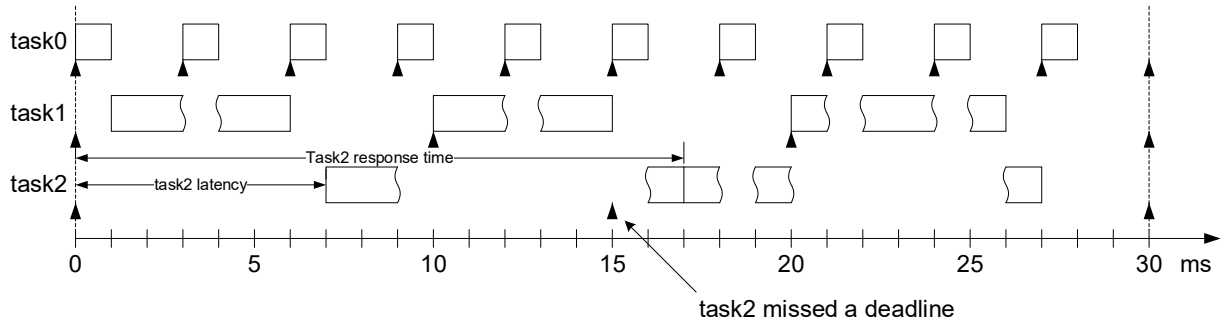
Task	Period	Execution Time
task0	3 ms	1 ms
task1	10 ms	4 ms
task2	15 ms	3 ms

a) [5 pts] Judging by the CPU utilization alone, are these tasks guaranteed to always meet their deadlines? The RMS assumptions are satisfied.

CPU utilization U = 1ms/3ms + 4ms/10ms + 3ms/15ms = 0.9333 = 93.33%
RMS CPU utilization bound = n*(2^(1/n)-1) = 3*(2^(1/3)-1) = 0.7798 = 77.98%
CPU utilization is greater than the RMS utilization bound, so these tasks are <u>not guaranteed to meet all deadlines</u>.

b) [15 pts] Analyze the RMS schedulability of these tasks using the graphical method. Tabulate the task priority, latency, response time and whether each task is schedulable.

Unrolled schedule with all tasks released at the same time (worst case):



The worst case latency and response time are measured in the above schedule from time = 0, when all interrupts occur simultaneously.

Task	Period	Execution Time	**Priority**	**Latency**	**Response Time**	**Schedulable?**
task0	3 ms	1 ms	**high**	**0 ms**	**1 ms**	**Yes**
task1	10 ms	4 ms	**mid**	**1 ms**	**6 ms**	**Yes**
task2	15 ms	3 ms	**low**	**7 ms**	**17 ms**	**No**

Note: You only need to draw the schedule up to 17 ms, where all latencies and response times have been determined.

c) [15 pts] Using the MATLAB source file response_time.m (from the course website), simulate optimization cases where you reduce the execution time of one task at a time by 20%. For each case, only one task is optimized, while the rest use their original execution times. Fill in the following table. Are the results as you expected? Explain.

task CPU load = task execution time / task period

The tasks are already sorted in order of decreasing RMS priority (increasing period). Here are the MATLAB calls to response_time() for all the optimization cases:

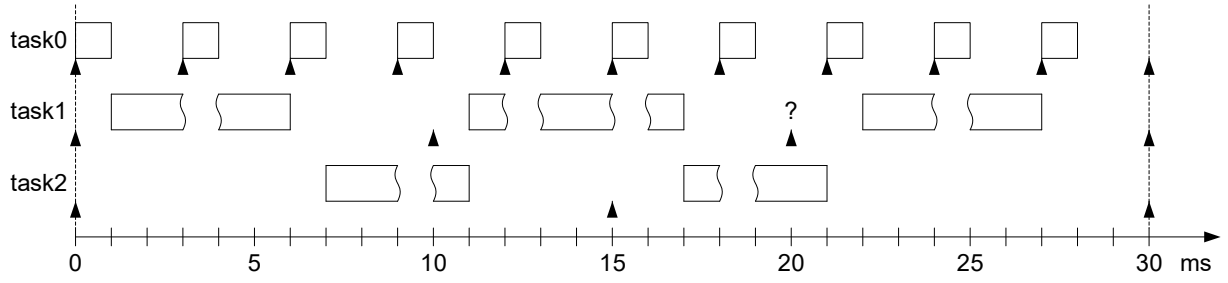
```
>> R = response_time([3,10,15], [1*0.8,4,3])
R =
          0.8          5.6           15
>> R = response_time([3,10,15], [1,4*0.8,3])
R =
            1          5.2         14.4
>> R = response_time([3,10,15], [1,4,3*0.8])
R =
            1            6         16.4
```

Case	Task to optimize	CPU load of the task to optimize, before optimization	task2 response time after optimization
1	task0	**33.3%**	**15 ms**
2	task1	**40%**	**14.4 ms**
3	task2	**20%**	**16.4 ms**

Yes, the results are as expected. Optimizing task1, the task with the highest CPU load, improves the task2 (lowest priority) response time the most. Optimizing task1 makes task2 schedulable.

d) [15 pts] Draw the EDF (Earliest Deadline First) schedule for the original three tasks over the least common multiple of the periods. Assume all tasks are initially released at the same time.

Here is one possible EDF schedule. The point of ambiguity is marked with "?" indicating the alternative task to run next. No deadlines are missed.



2) [20 pts] Your real-time system has 5 periodic ISRs with **constant** execution time and relative deadlines equal to the periods. These ISRs are **preemptive** and **prioritized** in accordance with RMS. You have measured the CPU utilization of these ISRs combined: 75%.
   a) [10 pts] Does this measurement indicate that your ISRs should theoretically meet all deadlines? If it does not, which ISR is most likely to miss deadlines? Explain.

RMS CPU utilization bound = n*(2^(1/n)-1) = 5*(2^(1/5)-1) = 0.7435 = 74.35%
Measured CPU utilization 75% > 74.35%, indicating that all tasks are not guaranteed to meet deadlines.
In practice, this CPU utilization bound tends to be conservative, so this system is likely to be schedulable by RMS.
The lowest priority ISR is the most likely to miss deadlines, as it is starved of the CPU by higher priority ISRs.

   b) [10 pts] You observe that the highest priority ISR is missing deadlines. Explain the most likely cause of this. Hint: Check which RMS assumptions may not be satisfied.

highest priority ISR response time = ISR latency + highest priority ISR execution time

As this ISR is missing deadlines, its max response time must be greater than its period (= relative deadline). Since other ISRs are not missing their deadlines, the highest priority ISR is not using 100% of the CPU, so should be schedulable according to RMS. However, the problem description does not state that all RMS assumptions are satisfied. Notably, the condition of no synchronization between tasks is absent, and ISR call overhead may be significant. Both of these increase the ISR latency, causing the missed deadlines. Long ISR latency is typically due to a critical section in a lower priority thread where interrupts are disabled. Although latency may be the culprit, reducing the execution time of the highest priority ISR may fix the problem (if the latency is smaller than its period).

Shared data problem

3) [30 pts] Your system is set up to interrupt when a fault is detected. The ISR increments a global fault counter, and main() handles the faults, decrementing the counter. The following code outlines the communication between the ISR and main(). Assume this code runs on an unspecified 32-bit CPU architecture: 32-bit **reads** and **writes** are atomic.

```
volatile uint32_t fault_count = 0;

void FaultDetectISR(void) {
  <clear interrupt flag>;
  fault_count++;
}

void main(void) {
  <init>;

  while (1) {
    while (fault_count > 0) {
      fault_count--; // read-modify-write the shared global
      <handle one fault>; // does not use fault_count
    }
    // other unrelated code
  }
}
```

a) [10 pts] Find the shared data bug in this code. Explain the sequence of events that may cause an error.

The above highlighted code indicates where the global is <u>read, modified and written</u> in main(), which is generally a non-atomic operation (assignment only states that reads and writes are atomic). FaultDetectISR() can interrupt main() after `fault_count` is read but before it is written, and increment the global (write to it). After the ISR returns, the global is overwritten with a value derived from an outdated local copy in main(). Thus, one fault is <u>missed</u>. This is an intermittent bug.

There is one other place where main() reads `fault_count`: the condition for the inner while() loop. However, this code, even when combined with the other access to `fault_count`, does not introduce other shared data bugs (such as non-atomic read). Reading `fault_count` to evaluate the while() condition is atomic. The comparison outcome is not invalidated even if the body of the while() loop is interrupted, as the ISR only <u>increments</u> `fault_count`.

b) [10 pts] Fix the shared data bug by disabling, then re-enabling interrupts. Exclude `<handle one fault>` from the critical section to minimize impact on interrupt latency.

Solution if you recognized that the while() loop condition does not introduce shared data bugs:

```
void main(void) {
    <init>;

    while (1) {
        while (fault_count > 0) {
            IntMasterDisable();
            fault_count--; // read-modify-write the shared global
            IntMasterEnable();
            <handle one fault>; // does not use fault_count
        }
        // other unrelated code
    }
}
```

Solution if you thought the while() loop condition is part of the shared data problem:

```
void main(void) {
    uint32_t local_faults;     // new local variable
    <init>;

    while (1) {
        IntMasterDisable();
        local_faults = fault_count; // read global into local var
        fault_count = 0;            // overwrite global
        IntMasterEnable();
        while (local_faults > 0) {  // local variable: not shared
            local_faults--;         // local variable: not shared
            <handle one fault>; // does not use fault_count
        }
        // other unrelated code
    }
}
```

You may come up with other variations, but they should exclude `<handle one fault>` from the critical section (where interrupts are disabled) to receive full credit.

c) [10 pts] Fix the shared data bug without disabling interrupts. Hints: You want to make the communication between the ISR and main() one-directional: if the ISR writes to `fault_count`, main() only reads it. This implies that `fault_count` cannot be decremented or modified in any other way in main(). Use local variables to replace the lost functionality.

It is not possible to make a read-modify-write operation atomic without disabling interrupts (or resorting to special CPU instructions). The solution, following the hints, is to make main() only read the global, but not write to it. We then need a local variable to count handled faults. The number of new faults can be determined as the difference between `fault_count` and the local `handled_faults`.

```
void main(void) {
    uint32_t handled_faults = 0; // new local variable
    <init>;

    while (1) {
        while (fault_count - handled_faults > 0) {  // atomic read
            handled_faults++;   // local variable
            <handle one fault>; // does not use fault_count
        }
        // other unrelated code
    }
}
```

There is an edge case when the fault counts overflow from 0xffffffff to 0. If you write the while() loop condition as `(fault_count > handled_faults)`, it will fail when `fault_count` overflows before `handled_faults`. However, you can write this condition as `(fault_count - handled_faults > 0)` or alternatively as `(fault_count != handled_faults)` and it will work properly. You <u>do not</u> need to make sure you code is safe in this edge case to receive full credit.