This exam is open book, open notes. Calculators are allowed. Internet access is not permitted. Show all work for partial credit.

1) Your real-time system contains 3 periodic tasks specified below.

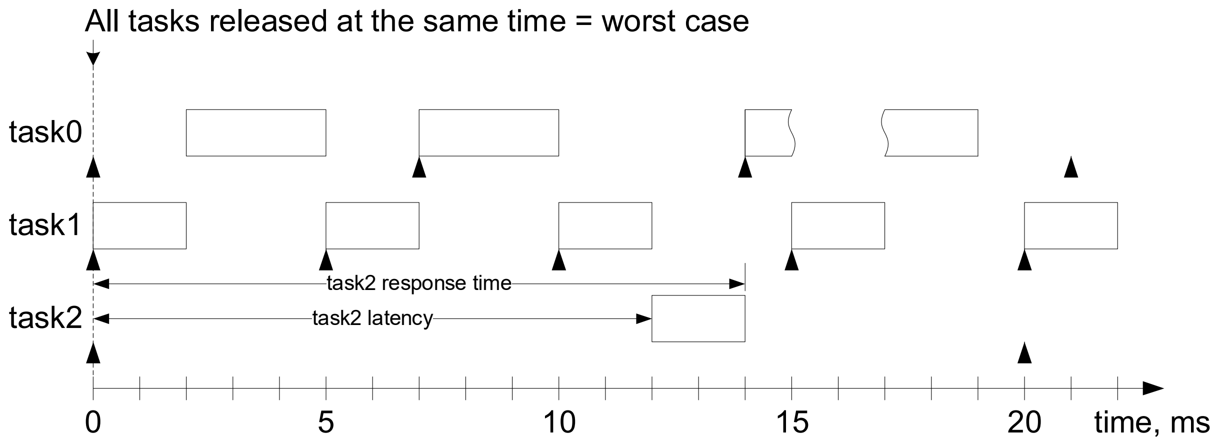
Task	Period	Execution time
task0	7 ms	3 ms
task1	5 ms	2 ms
task2	20 ms	2 ms

a) Assuming all the Rate Monotonic Scheduling (RMS) assumptions are satisfied, what does the CPU utilization of this system say on whether it is schedulable or not?

CPU utilization = 3ms/7ms + 2ms/5ms + 2ms/20ms = 0.9286 = 92.86%
RMS CPU utilization bound = n*(2^(1/n) - 1) = 3*(2^(1/3) - 1) = 0.7798 = 77.98%
CPU utilization (0.9286) > RMS CPU utilization bound (0.7798)
Therefore the CPU utilization does not guarantee that the system is schedulable by RMS. (This is not the same as stating "not schedulable.")

b) Assuming all RMS assumptions are satisfied, fill in the table below. Show your work. You need to draw a part of the RMS schedule for these tasks.

Task	Period	Execution time	Priority	Latency	Response time	Schedulable ?
task0	7 ms	3 ms	**mid**	**2 ms**	**5 ms**	**Yes**
task1	5 ms	2 ms	**high**	**0 ms**	**2 ms**	**Yes**
task2	20 ms	2 ms	**low**	**12 ms**	**14 ms**	**Yes**



We only need to draw the schedule until the longest response time is known.
A task is schedulable if its response time <= period

c) You had to protect a 2 ms long critical section in task2 by disabling interrupts. What are the latency and response time of task1 after this change? Is task1 still schedulable?

taks1 is the highest priority task (cannot be preempted by another task).

task1 response time = task1 latency + task1 execution time

task1 latency is dominated by the duration of the critical section in task2 where interrupts are disabled (assuming other overhead is negligible).

<u>task1 latency</u> = exec time of critical section in task2 = <u>2 ms</u>

<u>task1 response time</u> = 2 ms + 2 ms = <u>4 ms</u>

task1 relative deadline = task1 period = 5 ms > task2 response time (4 ms)

Therefore, task1 is still <u>schedulable</u>.

2) Your embedded system features two digital-to-analog converters that must be written to periodically. A periodic interrupt has been set up for this purpose. The function `SetDAC()` is called from `main()` to modify the DAC outputs. The DAC values may stay the same from last time, but must still be written in every `DAC_ISR()` call. This code runs on a **32-bit** CPU: 32-bit reads and writes are atomic.

```
uint16_t x = 512;
uint16_t y = 512;

void DAC_ISR(void)          // periodic ISR
{
    <clear interrupt flag>; // <...> delineates pseudo-code
    <DAC1 data> = x;        // set the DAC outputs:
    <DAC2 data> = y;        //   must be done every ISR call
}

void SetDAC(uint16_t x1, uint16_t y1) // called from main()
{
    x = x1;  ←— Interrupt
    y = y1;
}
```

a) Where is the shared data bug in this code? Explain the problem that can occur.

The global variables `x` and `y` are written non-atomically in `SetDAC()` (highlighted). These variables control the DAC outputs through `DAC_ISR()`. If the DAC_ISR() interrupt occurs after the write to `x` and before the write to `y`, the two DACs will be set to values inconsistent with each other.

(problem continues on the next page)

b) Fix the shared data bug without disabling interrupts. You may modify both functions and introduce new global variables. Note that there is more than one approach to this. If you cannot figure this out, fix it by disabling interrupts for [less credit] (you may give both answers; the highest scoring one will be counted).

One approach is to pack both 16-bit x and y into a single 32-bit global variable that can be written atomically.

```
uint32_t a = (512 << 16) | 512; // init value not required for credit

void DAC_ISR(void)            // periodic ISR
{
    <clear interrupt flag>;
    <DAC1 data> = a >> 16;    // set the DAC outputs:
    <DAC2 data> = a & 0xffff; //   must be done every ISR call
}

void SetDAC(uint16_t x1, uint16_t y1) // called from main()
{
    a = ((uint32_t)x1 << 16) | y1; // atomic write to a
}
```

The following is a version that disables interrupts.

```
void SetDAC(uint16_t x1, uint16_t y1) // called from main()
{
    IntMasterDisable();
    x = x1;
    y = y1;
    IntMasterEnable();
}
```

(alternative solution on next page)

Another approach is to protect access to the shared globals using a boolean flag that can be written atomically. In this case, because the DACs must be written to in every ISR call, we need more globals to hold the updated DAC values. The update is performed in the ISR.

A similar effect can be achieved with a FIFO.

```
uint16_t x = 512, x_new;
uint16_t y = 512, y_new;
uint8_t update = 0;

void DAC_ISR(void)          // periodic ISR
{
    <clear interrupt flag>; // <...> delineates pseudo-code
    if (update) {
        x = x_new;
        y = y_new;
    }
    <DAC1 data> = x;        // set the DAC outputs:
    <DAC2 data> = y;        //   must be done every ISR call
}

void SetDAC(uint16_t x1, uint16_t y1) // called from main()
{
    update = 0; // disallow updates of x and y
    x_new = x1;
    y_new = y1;
    update = 1; // allow updates of x and y
}
```

3) The function `update()` is to be used by multiple threads in a preemptive multitasking system. This code runs on a **32-bit** CPU: 32-bit reads and writes are atomic.

```
int32_t a[256] = {0};
int32_t f(uint8_t z); // implemented elsewhere (not using a[])
int32_t update(uint8_t x)
{
    int32_t y = f(x);
    a[x] = y;
    return y + x;
}
```

a) **Label** and **explain** all the locations in `update()` that cause you to suspect non-reentrancy (even if they turn out to be benign).

1. The function call to `f()` is potentially non-reentrant if that function itself is non-reentrant. The reentrancy properties of `f()` are unknown.

2. The single write to one entry of the array `a[]` may be non-atomic and therefore cause this function to be non-reentrant. In this case, the write is to a 32-bit variable on a 32-bit system, therefore atomic, and not a problem.

3. The write to `a[]` may interact with the call to `f()` and cause non-reentrancy. In this case, the comments say that `f()` does not use `a[]`, so this is not a problem.

b) Rewrite `update()` to make sure it is reentrant. Disable interrupts only for the shortest duration necessary.

```
int32_t update(uint8_t x)
{
    IntMasterDisable();
    int32_t y = f(x); // calling potentially non-reentrant function
    IntMasterEnable();
    a[x] = y;         // atomic write to global, not used by f()
    return y + x;     // local variables
}
```

4) Short answers.

a) Why does the Nested Vectored Interrupt Controller (NVIC) need to know which ISR is currently executing?

To disable interrupts of <u>same</u> or <u>lower priority</u> compared to the currently executing ISR.

b) Why does the RTOS need to know that an ISR is executing?

To <u>delay any task switch until the ISR returns</u> to task code.

The ISR can unblock tasks, so a task switch may be necessary. If the OS were not aware that an ISR is executing, it would perform the tasks switch in the middle of the ISR, which can crash the OS.

c) Under an RTOS, what is the largest component of the highest priority **Task latency** in the absence of critical sections in the application code? Assume ISRs and Tasks are the only thread types under this RTOS.

The <u>sum of the execution times of all the ISRs in the system</u>.

All ISRs are higher priority than all Tasks. Therefore, even the highest priority Task has to wait for all ISRs to finish running before it can run.