ECE 3849 D2019 Exam 1 solutions

Name ______________________________ **ECE mailbox** __________

Problem	Max pts	Score
1	16	
2	37	
3	29	
4	18	
Total	100	

This exam is <u>open book</u>, <u>open notes</u>. Calculators are allowed. Internet access is not permitted. Show all work for full credit. Partial credit is also assigned.

1) [16 pts] The function `lookup()` 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. Is this function reentrant, non-reentrant, or reentrant under certain conditions? **Label** and **explain** all the locations in `lookup()` that cause you to suspect non-reentrancy.

```
void lookup(int32_t i, int32_t *p)
{
    static int32_t t[3] = {420, 5032, 220356};
    if (i >=0 && i < 3) {
        *p = t[i];
    }
}
```

This function has two locations that arouse worries about non-reentrancy:

- **The access to the <u>static</u> array `t[]`. However, there is only a <u>read</u> access (`t[]` is used as a constant). Therefore this does not create a shared data issue.**
- **The access to an unknown location by pointer `p`. There is only a single <u>write</u> access to `*p`. This does not create a shared data issue because a write to `int32_t` is <u>atomic</u> on this 32-bit CPU.**

Therefore this function is <u>reentrant</u>.

2) [37 pts] Your real-time system contains 3 periodic tasks specified below.

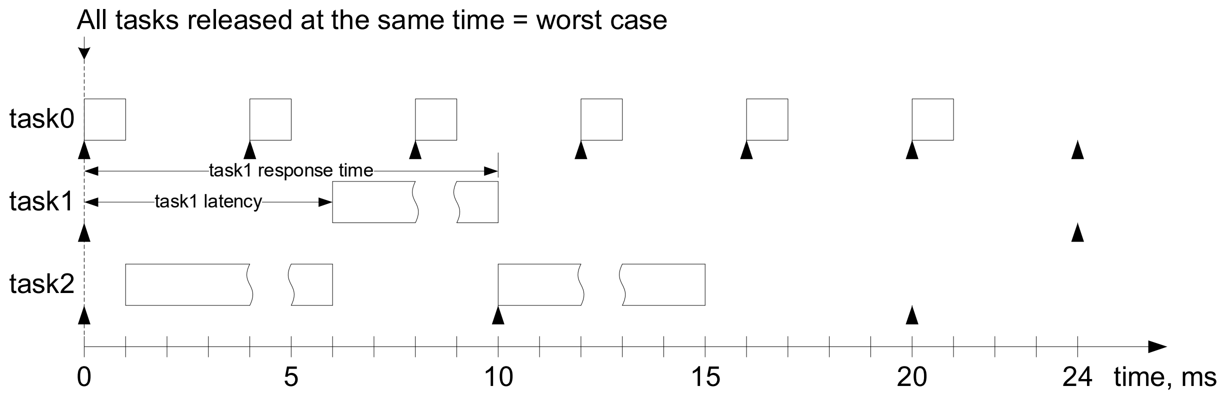
Task	Period	Execution time
task0	4 ms	1 ms
task1	24 ms	3 ms
task2	10 ms	4 ms

a) [8 pts] 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 = 1ms/4ms + 3ms/24ms + 4ms/10ms = 0.775 = 77.5%
RMS CPU utilization bound = n*(2^(1/n) - 1) = 3*(2^(1/3) - 1) = 0.7798 = 77.98%
CPU utilization (0.775) < RMS CPU utilization bound (0.7798)
Therefore the CPU utilization guarantees that the system is schedulable by RMS.

b) [20 pts] 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	4 ms	1 ms	**high**	**0 ms**	**1 ms**	**Yes**
task1	24 ms	3 ms	**low**	**6 ms**	**10 ms**	**Yes**
task2	10 ms	4 ms	**mid**	**1 ms**	**6 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

(problem continues on the next page)

c) [9 pts] We have learned that the “no synchronization” RMS assumption is not likely to be satisfied in a typical real-time system. Briefly explain **why** it is not likely to be satisfied, how this modifies the RMS **schedule**, and which **tasks** may start missing deadlines because of this.

Why: Tasks need to <u>share data</u> among them. Synchronization is necessary to ensure <u>mutually exclusive</u> access to shared data.

Effect on schedule: Synchronization adds <u>latency</u> and/or <u>prolongs response time</u> of tasks. For example if interrupts are disabled in critical sections, latency is extended by the duration of the longest such critical section.

Which tasks miss deadlines: <u>Highest priority</u> tasks are most likely to miss deadlines due the added latency or extended response time because their <u>relative deadlines</u> are the <u>shortest</u>.

3) [29 pts] A timer ISR keeps track of time in milliseconds. Functions have been written to allow main() to read and adjust the time. This code runs on a **32-bit** CPU: 32-bit reads and writes are atomic.

```
volatile uint32_t time = 0;

void TimerISR(void) {            // periodic ISR, period = 1 ms
    <clear timer interrupt flag>; // <...> delineates pseudo-code
    time++;
}
uint32_t GetTime() {             // called from main()
    return time;
}
void AdjustTime(uint32_t adj) { // called from main()
    time += adj;
}
```

a) [10 pts] AdjustTime() is known to intermittently corrupt the `time`. Why does this happen? What is the worst case deviation from the correct time due to a single occurrence of the shared data issue? Exclude the special case when `time` wraps around.

The read-modify-write operation AdjustTime() performs on the `time` shared variable is not atomic on this CPU architecture. If interrupted after the read, but before the write, the ISR increments `time`. After returning from the ISR, AdjustTime() overwrites `time` with a modified local copy from before the ISR call. Thus, one `time` increment is lost, and the expected time deviation due to one occurrence of the shared data issue is 1 ms (one timer period). Two or more timer interrupts may occur if there are other ISRs with high CPU load in the system. In that case the time error may be several ms.

b) [5 pts] Why do shared data issues not occur in GetTime()?

GetTime() only reads the `time` shared variable. This is an atomic operation on a 32-bit CPU architecture.

(problem continues on next page)

c) [14 pts] Fix AdjustTime() such that it never causes even a small `time` error. You may introduce new variables and modify TimerISR(). AdjustTime() may wait for an interrupt. If you cannot figure this out, fix the shared data bug by disabling interrupts for [8 pts] (you may give both answers; the highest scoring one will be counted).

One possible fix is for AdjustTime() to write to a global to indicate time adjustment, but for the ISR to do the actual adjustment. AdjustTime() also needs to check if the last adjustment has gone into effect. For this part to work, the global variable `adjust` must be `volatile`.

```
volatile uint32_t time = 0;
volatile uint32_t adjust = 0; // time adjustment shared variable

void TimerISR(void)
{
    <clear timer interrupt flag>;
    time += adjust + 1;  // include adjustment in increment
    adjust = 0;          // reset adjustment to 0
}

void AdjustTime(uint32_t adj)
{
    while (adjust != 0); // wait until the ISR resets adjust to 0
    adjust = adj;        // write to shared variable: atomic
}
```

Here is the solution that disables interrupts to fix the shared data bug:

```
void AdjustTime(uint32_t adj)
{
    IntMasterDisable();
    time += adj;
    IntMasterEnable();
}
```

4) [18 pts] Short answers.

a) [6 pts] Why are the normal TI-RTOS Hwi (hardware interrupt) service routines called by the OS, rather than directly by the interrupt controller?

Because these Hwi <u>can make OS calls</u> that unblock tasks. The RTOS needs to keep track of Hwi entry and exit. If the Hwi unblocks a task, any <u>task switch</u> is delayed until the <u>return from Hwi to Task code</u>.

b) [6 pts] Suppose an interrupt controller limitation requires interrupts to be globally disabled within all ISRs. You may assume interrupts are not disabled anywhere outside of ISRs, and ISR priority is configurable. What is the shortest worst-case latency a specific ISR can achieve in this system?

An ISR achieves its best latency when assigned the <u>highest priority</u>. Since it cannot preempt other ISRs, its shortest worst-case latency is the <u>execution time of the longest of the other ISRs in the system</u>.

c) [6 pts] To protect critical sections that access data shared with ISRs, it is typical to **globally** disable interrupts. Give two reasons for doing this over disabling only the specific higher-priority ISR with which we are sharing data.

1. **Priority inversion.**
2. **Error-prone.**
3. **Less portable (hardware-specific interrupt manipulation code).**