Name: Solutions (Bonus 2 points)

# 18-648: Embedded Real-Time Systems

## Quiz #2

**Fall 2017**

**60 minutes**

**Instructions**

1. Be brief and to the point. Verbose answers that beat around the bush will be penalized.
2. Show all relevant work.
3. Partial credit may be given for some questions.
4. The use of a calculator is allowed.
5. The time limit will be strictly enforced.
6. **Watch the screen for any clarifications.**
7. Don't get stuck on any single question. Come back to any difficult questions later.
8. Please assume "traditional" systems unless specifically specified otherwise.
9. Empty pages are at the back in case you need some scratchpad space.

---

**For Graders' Use Only**

Name: __________ / 2 (Bonus)

1.__________ / 20

2.__________ / 30

3.__________ / 10

4.__________ / 15

**TOTAL. __________ / 75**

### *Question 1. Quick Take: True or False? (20 points)*

a. False — The "Compliant Kernel" approach benefits immediately from the Linux kernel contributions by the large open-source Linux community.

b. False — The deadline-monotonic scheduler (DMS) is optimal across all fixed-priority preemptive-scheduling policies for tasks with $D > T$.

c. False — A rate-monotonic scheduler is optimal across all fixed priority preemptive scheduling policies for tasks with $D > T$.

d. False — A rate-monotonic scheduler is optimal across all fixed-priority preemptive scheduling policies for tasks with $D \leq T$.

e. True — A deadline-monotonic scheduler is optimal across all fixed-priority preemptive scheduling policies for tasks with $D \leq T$.

f. True — When using hardware interrupts together with DMS, the hardware interrupt task can be treated as if it is causing priority inversion and thus causing a blocking time to all periodic tasks with shorter deadlines.

g. True — Interrupt handlers are (normally) unschedulable by the OS.

h. True — Hard real-time tasks cannot use the Linux API or Linux facilities in the *dual kernel* approach within Linux.

i. True — The resource reservation concept for temporal resources can be supported by either fixed-priority or dynamic-priority preemptive scheduling in resource kernels.

j. True — Deadline-monotonic scheduling and earliest deadline first policies are not *equivalent*.

## *Question 2. To Produce or Not to Produce? (30 points)*

A producer thread produces items into a buffer and executes the function strangely named `produce()`. A consumer thread consumes items that have been produced into the buffer and executes the function called, surprise, `consume()`.

The buffer is a "circular buffer" so that items can both be produced and consumed in First-In-First-Out fashion. The producer will produce items at the *tail* of the circular buffer, and the consumer will consume at the *head* of the buffer. The producer calls `AddToBuffer()` to add an item to the tail of the buffer if it has at least one empty slot. The consumer calls `RemoveFromBuffer()` to remove an item at the head of the buffer if there is at least one valid item in the buffer.

Assume that the circular buffer implementation is correct but hidden from you, and that the methods `full()` and `empty()` on a circular buffer object correctly return whether the buffer is full or empty.

You are right: a producer should *not* produce into a full buffer and a consumer should *not* consume from an empty buffer. Right again: the buffer is shared between the producer and consumer threads. Assume that all data types (such as circular_buffer_t and item_t) are defined and correct.

```
circular_buffer_t *circBuf;
pthread_mutex_t   buffer_mutex;     /*mutex to protect circBuf */

pthread_cond_t    buffer_full_cv;   /*condition variable for circBuf full*/

pthread_cond_t    buffer_empty_cv;  /*condition variable for circBuf empty*/

void produce (item_t *item)
{
    pthread_mutex_lock(&buffer_mutex); /* lock buffer */
    /* check for full buffer. The reason why we need a while loop here is
    subtle. It is necessary to use a while loop instead of a conditional if
    we consider the case where we have multiple producer threads. Consider
    the case where a producer thread, call it T1, is about to wake up, due
    to a call to pthread_cond_signal(), and another producer thread, call it
    T2, starts executing produce. Prior to T1 waking, T2 can acquire the
    mutex, see that the buffer is not full, and add an item, making the
    buffer full. After T2 releases the mutex, T1 returns from
    pthread_cond_wait(). At this point, the buffer is full, and  T1 should
    not add to the buffer. To know this, it should check the buffer state
    again. This can only be accomplished with a while loop; not a
    conditional. */.
    while (circBuf->full())
    {
          pthread_cond_wait(&buffer_full_cv, &buffer_mutex);
          continue;
    }
    /* "produce" item into buffer: "full" status is automatically updated */
    AddToBuffer(item);
   /* at least 1 item available; wake up any consumer blocked on buffer
    * one blocked consumer will be awakened after mutex is unlocked */
   pthread_cond_signal(&buffer_empty_cv);
   pthread_mutex_unlock(*buffer_mutex); /* unlock buffer */
   return(0);

}

item_t *consume()
{
       item_t *item;
       pthread_mutex_lock(&buffer_mutex); /* lock buffer */
       while (circBuf ->empty())   /* check for empty buffer */
       {
```

```
        /* buffer is empty; block for at least one buffer slot to fill
         * note that the mutex will be released automatically by call */
        pthread_cond_wait(&buffer_empty_cv, &buffer_mutex);
        /* when awakened by a producer's pthread_cond_signal() call,
         * the thread will also automatically get the mutex lock */
              continue;
        }

        /* "consume" item from buffer: 'empty' status will be updated */
        item = RemoveFromBuf(circBuf);

        /* item consumed resulting in at least one empty slot in buffer;
         * wake up any producer blocked on full buffer */
        pthread_cond_signal(&buffer_full_cv);
        pthread_mutex_unlock(*buffer_mutex); /*unlock buffer*/

        return(item);

}
```

```
Things to note:

1. The circular buffer is shared between the producer and consumer threads.
   Hence, a mutex (only one) is needed to ensure that no more than one
   thread accesses this shared data structure (aka critical section).

2. There are two conditions to monitor: "buffer is full" (when producers
   have to wait), and "buffer is empty" (when consumers have to wait).

3. For each cond_wait, there must be a corresponding cond_signal.
```

(a) Do you see any problem(s) with the above solution? If so, point them out. Else, say why not.

*(5 points)*

There are multiple problems:

1. The shared buffer data structure is not protected and two or more concurrent threads can end up accessing it concurrently causing the structure to become incorrect/inconsistent.

2. The producer "busy-waits" (or polls continuously) when the buffer is full causing a ton of wasted CPU cycles.

3. The consumer "busy-waits" (or polls continuously) when the buffer is empty also causing a ton of wasted CPU cycles.

(b) Demonstrate your knowledge of POSIX primitives with (very rough) syntax, and fix any and all problem(s) you see. Approximate pseudo-code is fine in terms of parameters. Default POSIX object attributes can be assumed. Annotate directly in code segments above. *(20 points)*

Please see highlighted segments in code.

(c) Does your solution work for any number of producer threads and consumer threads that call `produce()` and `consume()` respectively? Why or why not? *(5 points)*

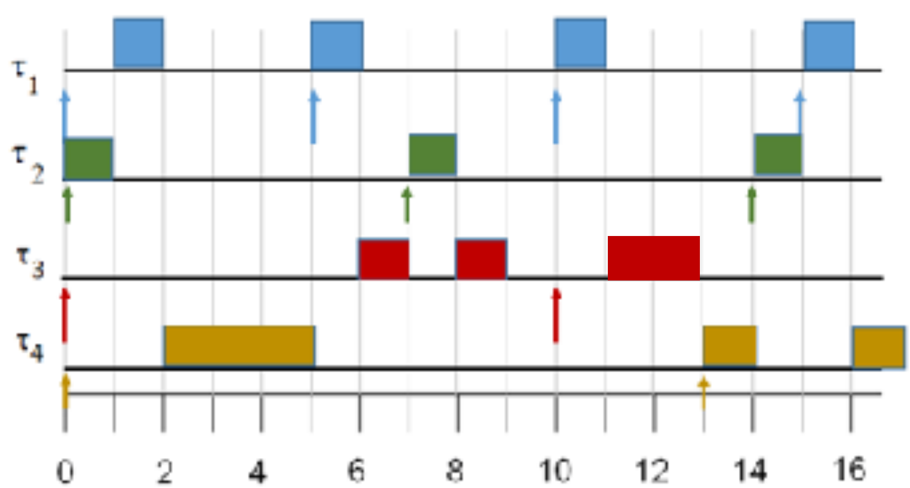
Yes, it does. The buffer mutex will ensure that no more than one thread (either producer or consumer) is manipulating the buffer (critical section) at any one time. The "buffer full" condition variable will suspend any producer trying to add an item to the buffer when the latter is full. The "buffer empty" condition variable will suspend any consumer trying to pull an item from the buffer when the latter is empty. When an item is produced, a suspended consumer will be awakened. When an item is consumed, a suspended producer will be awakened. *(Condition variables are neat – but have to be used correctly in conjunction with a mutex).*

### *Question 3. Is it* always so *monotonic out here? (10 points)*

Draw the time-line diagram (from t= 0 to t = 16) for the following taskset under deadline-monotonic scheduling. Assume that all tasks arrive at their respective critical instants. Recall that a critical instant of a task τ is its relative phasing where τ encounters its worst-case response time.

| Task | $C_i$ | $T_i$ | $D_i$ | **Priority** |
|---|---|---|---|---|
| $\tau_1$ | 1 | 5 | 3 | 2nd Highest |
| $\tau_2$ | 1 | 7 | 2 | Highest |
| $\tau_3$ | 2 | 10 | 9 | L |
| $\tau_4$ | 3 | 13 | 6 | 3rd Highest |

Under DMS, tasks with shorter (relative) deadlines get higher priority values as shown above. Hence, the timeline looks as follows:



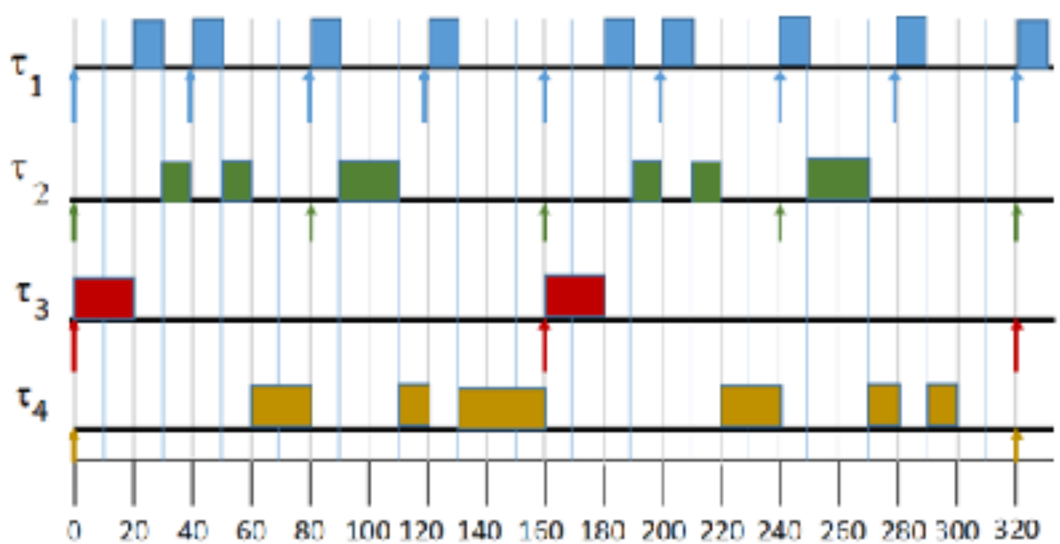
### *Question 4. Why do you keep interrupting? (15 points)*

Consider the following taskset where task $\tau_3$ represents an interrupt handler executing at higher priority than the other "user space" tasks. Assume that $D_i = T_i$ for all tasks.

| Task | $C_i$ | $T_i$ | *Use as you see fit*<br>***Priority*** | *Use as you see fit*<br>***Blocking Term*** |
|---|---|---|---|---|
| $\tau_1$ | 10 | 40 | 2$^{nd}$ Highest | 20 |
| $\tau_2$ | 20 | 80 | 3$^{rd}$ Highest | 20 |
| $\tau_3$ | 20 | 160 | Highest | 0 |
| $\tau_4$ | 100 | 320 | Lowest | 0 |

(a) Draw the timeline for the above taskset when all tasks are released at time 0. (5 points)

Since $\tau_3$ is an interrupt task, it has the highest priority. The other tasks are ordered according to RMS as shown in the penultimate column above. The timeline then looks as follows




(b) *Analytically*, show whether the above taskset is schedulable or not. (10 points)

We first note that RMS and DMS are one and the same in this case since $D_i = T_i$.

Secondly, Under RMS, $\tau_3$ should have lower priority $\tau_1$ than and $\tau_2$, but as an interrupt handler it has the highest priority. Hence, from an RMS/DMS perspective, $\tau_3$ introduces priority inversion to $\tau_1$ and $\tau_2$. Since there can be at most one instance of $\tau_3$ within the shorter periods of $\tau_1$ or $\tau_2$, the worst-case priority inversion durations for $\tau_1$ and $\tau_2$ are 20 and 20 respectively.

We next note that the taskset is harmonic with periods of 40, 80, 160 and 320. Hence, under RMS, the schedulable utilization bound is 100%.

Each task must be checked for schedulability.

The schedulability condition for $\tau_1$ is given by

$(C_1 + B_1) / T_1 \leq 1.0$ ?

$(10 + 20) / 40 \leq 1.0$ ? Yes, $0.75 \leq 1.0 \rightarrow$ Task $\tau_1$ is schedulable.

The schedulability condition for $\tau_2$ is given by

$C_1 / T_1 + (C_2 + B_2) / T_2 \leq 1.0$ ?

$10/40 + (20+20)/80 \leq 1.0$ ? Yes, $0.75 \leq 1.0 \rightarrow$ Task $\tau_2$ is schedulable.

The schedulability condition for $\tau_3$, which is the highest-priority task, is given by

$C_3 / T_3 \leq 1.0$ ?

$20 / 160 \leq 1.0$ ? Yes, $0.125 \leq 1.0 \rightarrow$ Task $\tau_3$ is schedulable.

The schedulability condition for $\tau_4$ is given by

$U_{total} = C_1 / T_1 + C_2 / T_2 + C_3 / T_3 + C_4 / T_4 \leq 1.0$ ?

$10/40 + 20/80 + 20/160 + 100/320 \leq 1.0$ ?

Yes, $300/320 \leq 1.0 \rightarrow$ Task $\tau_4$ is schedulable.

Hence, the entire taskset is schedulable even in the worst case.