E4.52 | IA 3.35

Solutions 2008

FOUR Questions in 180 minutes => 45 min per question

Answer codes: A=analysis, B=bookwork, D=design, C= new application of learnt theory

Question 1.

This question tests whether the students understand some of the issues when writing application code for RTOS

a) RMA theorem guarantees that all deadlines will be met in a pre-emptive priority-scheduled system of jobs.Each job *i* is assumed to have fixed repetitive period Ti at end of which is the deadline, and have a fixed CPU time Ci, and there are *n* jobs.

Providing that jobs are ordered strictly with shorter periods having higher priority, and the RMA limit is met:

$$\sum_i \frac{C_i}{T_i} \le n\left(2^{\frac{1}{n}} - 1\right)$$

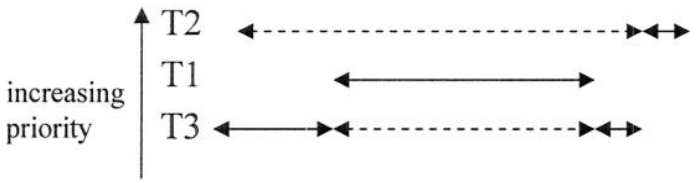
Then all deadlines are guaranteed to be met.

Extended RMA extends this to the case where jobs may block on semaphores etc, providing that a bound on the maximum blocking time B_i for each job is known. In this case $B_i + C_i$ replaces C_i in the above formula.

[4B]

b) From RMA, priority must be T2 > T1 > T3

The access to R_A is OK since the two tasks that do this have no intermediate priority tasks, however that for R_B suffers priority inversion where T2, blocking on the semaphore held by T3, must wait for T1.



Total blocking time (NB task cannot be blocked by a higher priority task except due to priority inversion). T2: t2+20us, T1: t1, T3: 0.

RMA: (20+t1)/200+ (10+t2+20)/100+100/400 = t1/200+t2/100+0.65 < 0.780 =>

t1+2t2 < 26us

[8C/A]

c) Possible reasons: deadlock, starvation or livelock. Can distinguish by first examining task status, if bad tasks are not always blocked must be livelock. Otherwise construct resource dependency graph to see whether there are any cycles. If there are, it is deadlock, otherwise it is starvation.

Since all three of these problems are schedule dependent, it is not possible to say whether the original application will continue to work under different circumstances. Thus it should be considered to be incorrect even if in fact it never exhibits problems in the original form.

[8A]

Question 2.

This question tests student's understanding of scheduling algorithms and analysis of real-time performance. The last two parts require innovative thinking.

a)

(i)

50<t1<250: Y>X>W>Z

250 < t1 <400: Y > W > X > Z

t1 > 400: Y > W > Z > X

(ii)

RMA: 50/t1+0.1+0.25+0.2= 50/t1+0.55 < 0.757 => 50/t1 < 0.207 => t1 > 242

If this inequality is met, then all deadlines met, otherwise cannot be sure.

[6C]

b)

(i) Priorities stay as before

(ii) Since Y > Z,W we do not need to consider blocking of Z,W by Y. So only change is 2us blocking of Y => utilisation increases by 0.2.

50/t1 < 0.007 => t1 > 7000us!

[4C]

c) Normally, EDF will allow 100% CPU utilisation. Where tasks are blocked the blocking time can be added to the job time to make modified utilisation. Thus we have 50/t1+0.25+102/400+52/250 < 1 => t1 > 174.2

[4C]

d)

In this case allowance must be made for possible blocking at any time, even near the deadline. Thus worst case all deadlines must be moved forward by the max blocking time for their job, and EDF implemented with the adjusted deadlines. However the blocking may be before the adjusted deadlines and hence CPU utilisation could be impacted by up to the sum of B_i/T_i : 1 - (2/400+2/250)=0.987

[4A]

Question 3

This question tests whether students understand the implementation of RTOSes by examining in detail the behaviour of some real (but not ideal, and hence now replaced) code.

a)

The message posted by task A will unblock task B, which will then pre-empt task A and execute.

Initially task B executes QueueReceive until line 640, when it yields to task A. Task A executes QueueSend 472-481 and 533-560 when it yields because task B is higher priority and has been unblocked.

QueueReceive then executes from 641, setting xReturn to pdPass on line 666 and returning this on line 688.

[8B]

b) Task A will not be pre-empted by the ready task B and will therefore continue to execute and receive the message it has just posted, making the queue empty again. When subsequently task A sleeps task B will run but return from QueueReceive with a pdFail value because the queue is still empty.

[4A]

c) this section of code is a critical section, so while it executes no ISR can execute. However the taskYield() will always cause the current task to suspend (unless taskResumeAll() has previously done this). After the context switch ISRs may occur, if the switched to task has interrupts enabled.

[4A]

d) The ISR can execute only when interrupts are on. There are three cases inside QueueSend():

1) Before the critical section surrounding the queue access 470-481

2) After the message is copied into the queue: 549-570

(Note that the queue is not full so 482-527 - parts of which also allow interrupts - will not execute)

In all cases the queue will end up with two messages, the order depends on where the ISR executes. In case 1 the ISR message m2 is queued first, in case 2 m1 is queued first. Both the ISR and task A QueueSend will wake up task C because it is the highest priority waiting task, so it will execute after the task A QueueSend yields at 558 or 560.

Task C will pick up the first of the two posted messages, run, and then pick up the second message. Note that Task B cannot do this since it will never be woken up unless a third message is posted, and in any case is lower priority than task C so will not run.

[6A]

4.

This question relates to the FreeRTOS task list package implementation, source code for which can be found in the booklet RTOS Exam Notes.

a)

add task to ready list

delete task from ready list

find highest priority task from ready list, or move to next task (round-robin) of set of tasks of identical priority in ready list.

[4B]

b)

add task to ready list - O(1)

delete task from ready list -

O(1)

find highest priority task from ready list, or move to next task (round-robin) of set of tasks of identical priority in ready list -

If highest priority task has just been deleted must search downwards for next highest priority. This search takes O(highest task priority - idle task priority) worst case. otherwise moving to the next task is O(1).

[8A]

c) The answer they give depends on whether they use an array of lists (replacing FreeRTOS doubly-linked list by singly-linked) or a single list.

	single list	array of lists
add	O(1) OR O(number of tasks) if sorted list	O(1)
delete	O(no of tasks)	O(no of tasks of equal priority)
find HPT	O(number of tasks) OR O(1) if sorted list	O(priority difference between HPT and idle task) + O(no of tasks of equal priority)

[8D/A]

5.

a) Detect deadlock via a timeout error on every semaphore take operation.

```
if (xSemaphoreTake( Sema, TIMEOUT)==pdFail) {
  /* stop using all resources and undo any changes*/
For all previously taken semaphores Semax do /* unlock resources */
  xSemaphoreGive(Semax, 0);
end;
taskDelay(random(100)); /* wait some application-dependent random backoff time to prevent repeated
        deadlocks */
/* Retry locking and using resources */
}
```

This will work providing whenever a resource is locked it is possible to stop using all resources and undo any changes made. this is trivially possible when resources are all locked sequentially before any resources are used.

[8B/D]

b)

(i)

```
#define TIMEOUT 1000 /* longer than expected waiting time in clock ticks */
/* in initial task or one of the two tasks before any resource access */
xQueuehandle Sema;
vSemaphoreCreateBinary(&Sema)

/* each Task */
if (xSemaphoreTake( Sema, TIMEOUT)==pdFail) {
  /* report timeout error */
}
/* access resource */
if (xSemaphoreGive(Sema, 0)==pdFail) {
  /* report multiple token error */
}
```

(ii)

```
vTaskSuspendAll()
access resource
vTaskResumeAll()
```

(iii)

```
taskENTER_CRITICAL()
access resource
taskEXIT_CRITICAL)
```

(i) is more complex and requires RAM resources for semaphore but only blocks tasks using the resource, and does not block interrupts or affect interrupt latency.

(ii) blocks all tasks, but does not impact interrupts hence does not affect interrupt latency

(iii) is simplest & fastest (on most architectures) but affects interrupt latency if section length is longer than max critical section length in application or kernel.

[6B]

c) If a cyclic resource graph is constructed (necessary for deadlock) the last link, which completes the cycle, can be denoted without loss of generality TN->R1 with other tasks and resources labelled as shown below as shown below:

R1->T1->R2->T2->...->RN->TN

PCP then controls whether TN can gain a lock on R1. This will be the case if the priority of TN is strictly greater than the priority ceiling of RN. By priority inheritance and the definition of priority ceiling, Rn* >=P(Tn)>=Rn+1*>=P(Tn+1). Therefore R1*>=P(TN) and the cycle cannot be constructed.

[6B]

6.

(a)

B has smaller interrupt latency so would normally be preferred. The average figures for latency and task switch time are not relevant to deadlines. The only case this would change is if there was a very high interrupt rate, so that the average interrupt latency contributed significantly to CPU utilisation, and if interrupt latency required from the application was much larger than that specified. In that case A would be preferred.

[4A]

(b)

```
ECB *ERCreate(char bitvalues)
void ERChangeBits( char bitsAffectedMask, char bitvalues)
ERWaitOnAllBits(char bitsToWaitOnMask, char bitvalues)
ERWaitOnAnyBits(char bitsToWaitOnMask, char bitvalues)

/* initialisation */
evr = ERCreate(0); /* create event register with all bits initially 0 */

Task()
{
/* start of barrier
ERChangeBits(Mask, 0x7); /* where mask = 0x1, 0x2, 0x4 for tasks 1,2,3 */
ERWaitOnAllBits( 0x7, 0x7); /* wait till all three bits are 1 */
/* end of barrier
}

Initialise()
  ERChangeBits( 0x7, 0); /* reset all bits */
}
```

[4D]

(c) (Many solutions are possible). For example: use the same three more bits of the event register to detect when all tasks have got past the barrier.

```
/* start of barrier
ERChangeBits(Mask, 0x7); /* where mask = 0x1, 0x2, 0x4 for tasks 1,2,3 */
ERWaitOnAllBits( 0x7, 0x7); /* wait till all three bits are 1 */
ERChangeBits(Mask , 0x0); /* where mask = 0x1, 0x2, 0x4 for tasks 1,2,3 */
ERWaitOnAllBits( 0x7, 0); /* wait till all three bits are 0 */
/* end of barrier
```

[4D]

(d) Adding the event register API will cost ROM (for the ER code) RAM proportional to the number of event registers in use, and a critical section of length 10us*number of waiting tasks. This latter is a significant cost if it is longer than the current longest critical section in the RTOS. If not there is no cost. If yes then it impacts interrupt latency globally. For the application above the maximum number of waiters is two (the last task to hit the barrier does not wait) so the interrupt disable time is 20us.

[4A]

(e) This implementation does not allow event flag changes from ISR, so it is not necessary to disable interrupts when performing event flag operations. Replace the interrupt disable critical section by a scheduler lock critical section and the operation will impact maximum task latency, but not maximum interrupt latency.

[4D]