Bugs found in FreeRTOS verification

February 9, 2014

FreeRTOS employs *prioritized pre-emptive* (or *non pre-emptive*) *scheduling policy*. The scheduler always schedules highest priority *ready* task. The user can configure FreeRTOS scheduler as pre-emptive or non pre-emptive. In the pre-emptive mode a task entering the *ready* state or having its priority altered will always pre-empt the *running task* if the running task has a lower priority. Low numeric priority values denote low-priority tasks, with priority 0 being the lowest priority possible [1].

1 Bug in Priority Inheritance scheme

A Mutex is used to control access to a resource that is shared between two or more tasks. The mutex can be thought of as a token that is associated with the resource being shared. For a task to access the resource legitimately, it must first successfully take the token (be the token holder). When the token holder has finished with the resource, it must give the token back. Only when the token has been returned can another task successfully take the token and then safely access the same shared resource. A task is not permitted to access the shared resource unless it holds the token [1].

1.1 Priority Inversion

A higher priority task may have to wait for a lower priority task to give up control of a mutex. A higher priority task being delayed by a lower priority task in this manner is called *priority inversion*.

1.2 Priority Inheritance

Priority inheritance is a scheme that minimizes the negative effects of priority inversion. Priority inheritance works by temporarily raising the priority of the mutex holder to that of the highest priority task that is attempting to obtain the same mutex. The low priority task that holds the mutex inherits the priority of the task waiting for the mutex. The priority of the mutex holder is reset automatically to its original value when it gives the mutex back.

Hence priority inheritance temporarily increases the priority of a low priority task (mutex holder) to minimize/bound the amount of time that the higher priority task has to wait for getting a mutex.

According to page number 58 of [1], "when more than one task blocked on a queue say Q to complete an operation, only one task will be unblocked when the operation is enabled on Q. *The task that is unblocked will always be the highest priority task that*

is waiting for the operation. If there are more than one blocked task with the highest priority, then the longest waiting task will be unblocked."

The implementation of the priority inheritance scheme in FreeRTOS fails to meet its goal when the task (whose priority is required to be inherited from a higher priority task) is blocked.

An application program demonstrating this bug is given in the file "main-1.c". The output of "main-1.c" is in the file "trace-1.txt".

The cause of the bug is that the function `vTaskPriorityInherit` directly modifies the value of a node in the event queue using the macro `listSET_LIST_ITEM_VALUE` (without changing its position in the list), thereby violating the ordering property of the queue.

We have verified the implementation of `vTaskPriorityInherit` with the following modification to fix the above bug. Replaced the call to `listSET_LIST_ITEM_VALUE` by calls to `vListRemove`,`listSET_LIST_ITEM_VALUE`, and then `vListInsert`.

2 Bug found in the API - *vTaskPrioritySet*

The `vTaskPrioritySet` API function can be used to change the priority of any task after the scheduler has been started.

In a similar way to the bug given in sec. 1, the `vTaskPrioritySet` API function fails to work correctly when it is used to change the priority of a blocked task.

An application program demonstrating this bug is given in "main-2.c" and its output trace is in "trace-2.txt".

Again the cause of the bug is that the function `vTaskPrioritySet` directly modifies the value of a node in the event queue using the macro `listSET_LIST_ITEM_VALUE` (without changing its position in the list), thereby violating the ordering property of the queue.

We have verified the implementation of `vTaskPrioritySet` with the following modification to fix the above bug. Replaced the call to `listSET_LIST_ITEM_VALUE` by calls to `vListRemove`,`listSET_LIST_ITEM_VALUE`, and then `vListInsert`.

3 Bug found in the API - *xTaskGenericCreate*

FreeRTOS violates the convention of scheduling the longest waiting highest priority task when the scheduler is started.

An application program demonstrating this bug is shown in "main-3.c" and its output trace is shown in the file "trace-3.txt".

The reason is that the function `xTaskGenericCreate` checks if the *pxcurrentTCB*'s (pointer to the task to be scheduled when the scheduler is initialised) priority is *less-than-or-equal-to* the newly created task's priority, in which case it makes the newly created task the *pxcurrentTCB*. In either case the newly created task is inserted at the end of its ready list. This is probably being done to maintain the invariant that the pxcurrentTCB is always at the *end* of the corresponding ready list.

An application program demonstrating this bug is given in "main-3.c" and its output trace is in "trace-3.txt".

The cause of the bug is that the guard for updating the pointer *pxcurrentTCB* in the function `xTaskGenericCreate` uses the comparison operator $\leq$ instead of $<$.

We have verified the implementation of `xTaskGenericCreate` by replacing the comparison operator $\leq$ by $<$ and doing a *rotate* of the ready queue corresponding to the top ready priority so that the *pxcurrentTCB* is at the end of its ready queue. The latter is changed in the port specific code.

4 Bug found in the APIs - *vTaskSuspend* and *vTaskResume*

The FreeRTOS functions `vTaskSuspend` and `vTaskResume` are used to suspend and resume a task in the system. Any task with access to the task handle of task t can suspend/resume t.

The implementation of suspend and resume in FreeRTOS just moves the suspended task to a suspended queue, and resume moves the task from here to the *ready queue*. Thus a suspended and resumed task loses its context (if, for example, it was suspended while it was in the delayed queue or event queue, on resumption it goes straight to ready).

An application program demonstrating this bug is given in "main-4.c" and its output trace is in "trace-4.txt".

We have verified the implementations of `vTaskSuspend` and `vTaskResume` by adding a precondition to the function `vTaskSuspend` to restrict the user to suspend a task only if its state is *ready*.

Permitting task suspend only when a task is ready is not a problem as suspending a blocked/delayed task can be implemented virtually by maintaining a boolean flag in the task control block. This flag can be set (or cleared) if user requests *suspend* (or *resume*) operation while a task is in blocked/delayed list. The scheduler should suspend the task if this flag is set when the delay/block period is over.

5 Other issues

The FreeRTOS function *vTaskDelay* is used to delay the current task for a given period. The time to delay is given by an argument to this function, which represents *the number of tick interrupts that the calling task should remain in the Blocked state before being transitioned back into the Ready state* (page number 29 of [1]). Thus a task is guaranteed to be blocked/delayed for the given period.

FreeRTOS fails to satisfy the above guaranty if the delayed task is suspended and resumed within the delay period. This is again caused by the bug given sec. 4.

The variable `uxTopReadyPriority` is meant to have the priority of the top ready task. This invariant is not maintained by the code, particularly when a task is moved out from ready. This does not lead to a problem as `vTaskSwitchContext` is usually called soon after, and restores the invariant. The invariant that is effectively maintained is that `uxTopReadyPriority` is an upper bound on the priority of the top ready task. It seems like it would be slightly more efficient to maintain the earlier invariant for `uxTopReadyPriority` (namely that it is always the priority of the top ready task).

References

[1] Richard Barry. **Using the FreeRTOS^{TM} Real Time Kernel- A Practical Guide**.