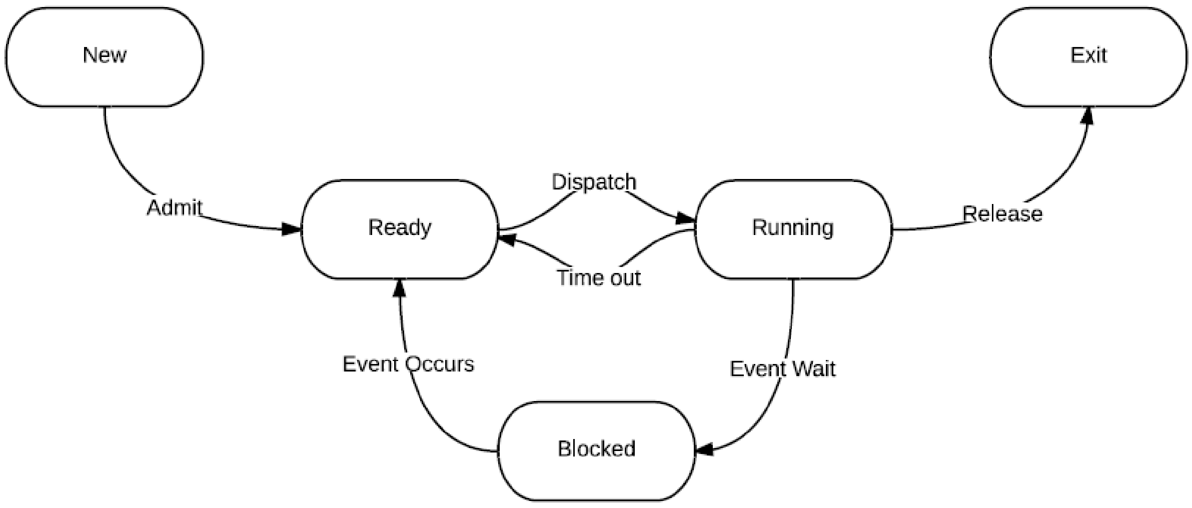
CG2271 Real-Time Operating Systems

Tutorial 4 Suggested Solutions

In this tutorial, we are going to cover many of the important aspects of MultiTasking through the Lab 6 Manual. Almost all the answers for the Lab Manual (except for the last part) are covered here. The objective is for you to have a very clear understanding of how a multi-threaded program works and to be able to analyse it.

Q1. Let's first look back at the State Transition Diagram.



a. Task A is Running and chooses to give up the CPU voluntarily, what state does it go to?

Answer:

It goes to the Ready State

b. Task A is Running and a higher priority Task B becomes Ready. What will happen?

Answer:

Task A will go to the Ready State and Task B will go to the Running State.

c. After some time, Task B requests for some resource and is unable to acquire it. It is unable to proceed without this resource. What happens?

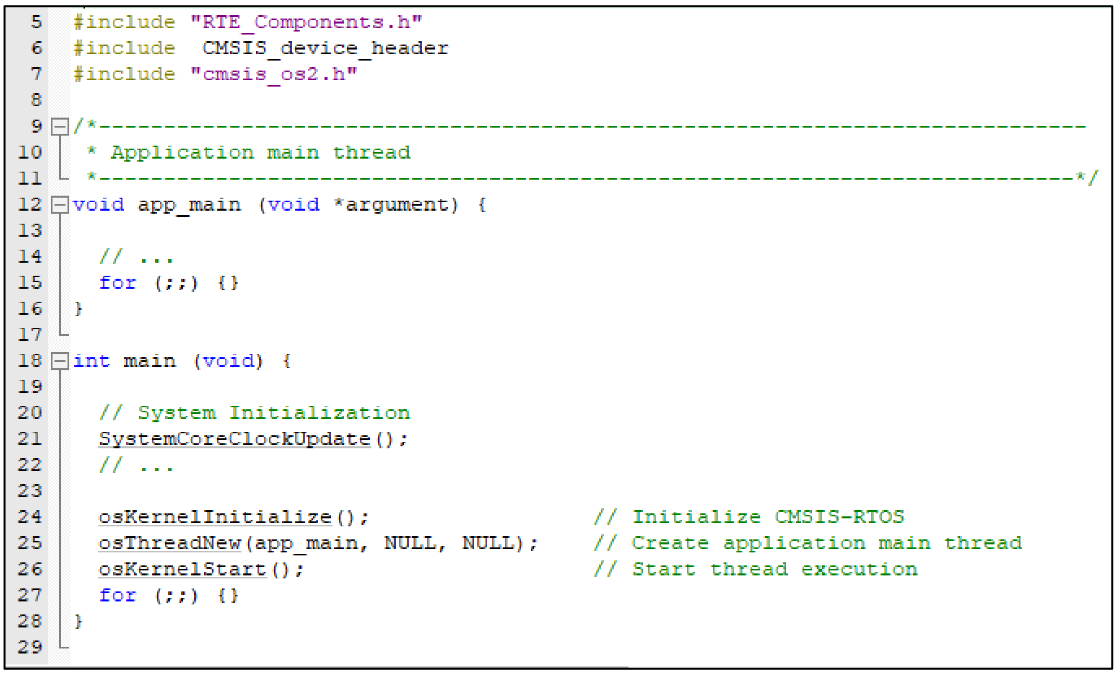
Answer:

Task B goes to the Blocked State and Task A goes to the Running State.

d. After 5ms, the resource required by Task B is available. What happens?

Answer: Task B goes from Blocked State to Ready State. If there is no other higher priority task running, it transitions to the Running State and Task A goes to the Ready State.

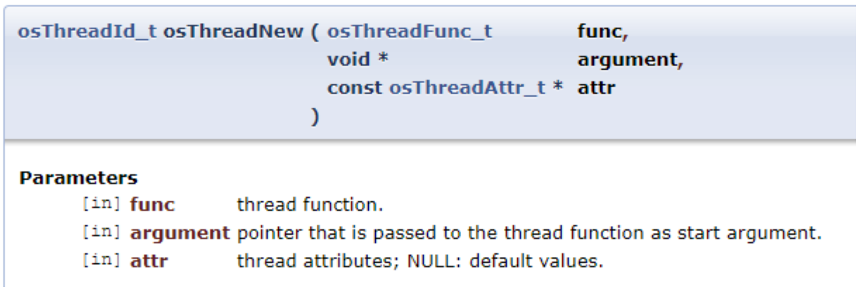
Q2. The following code snippet shows the way in which a task is created in RTX.



```
#include "RTE_Components.h"
#include  CMSIS_device_header
#include "cmsis_os2.h"

/*----------------------------------------------------------------------------
 * Application main thread
 *---------------------------------------------------------------------------*/
void app_main (void *argument) {

  // ...
  for (;;) {}
}

int main (void) {

  // System Initialization
  SystemCoreClockUpdate();
  // ...

  osKernelInitialize();                 // Initialize CMSIS-RTOS
  osThreadNew(app_main, NULL, NULL);    // Create application main thread
  osKernelStart();                      // Start thread execution
  for (;;) {}
}

```

a. The OS call, osThreadNew() takes in three parameters. What are they?

Answer:



```
osThreadId_t osThreadNew ( osThreadFunc_t        func,
                           void *                argument,
                           const osThreadAttr_t * attr
                         )
```

Parameters

[in]	**func**	thread function.
[in]	**argument**	pointer that is passed to the thread function as start argument.
[in]	**attr**	thread attributes; NULL: default values.

b. When will app_main() be called?

Answer:

Once the osKernelStart() function is called, the multi-threaded environment has started. At that time, there can be many tasks that are in the Ready state. The one with the highest priority will run.

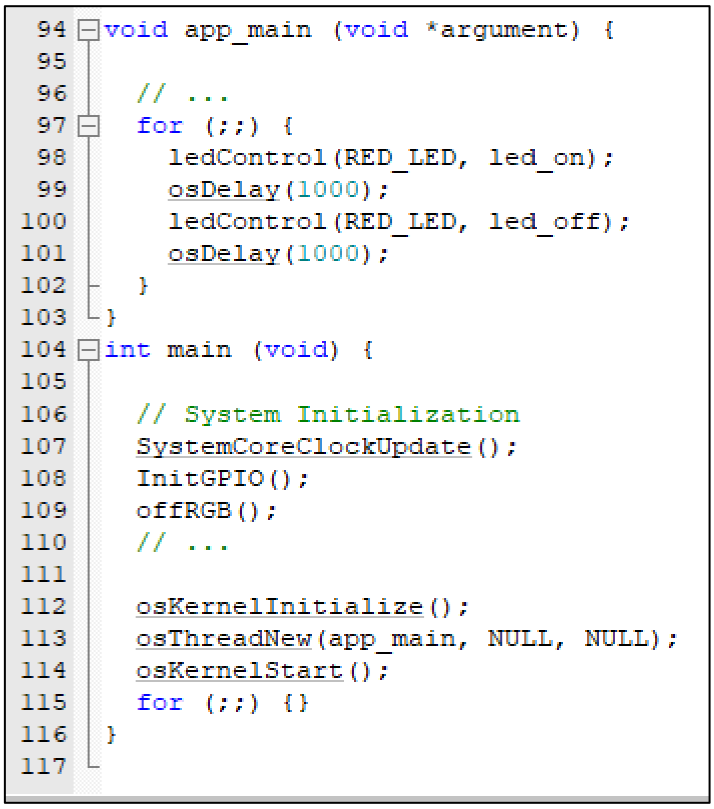
c. Why is there a need for the "for(;;) { }" loop in the app_main().

Answer:

In EOS, generally, most tasks are written in forever-loops. The state of the tasks will determine when they get to run.

Q3. Exploring the Blinky Function

Examine the following code snippet.



```
void app_main (void *argument) {

  // ...
  for (;;) {
    ledControl(RED_LED, led_on);
    osDelay(1000);
    ledControl(RED_LED, led_off);
    osDelay(1000);
  }
}
int main (void) {

  // System Initialization
  SystemCoreClockUpdate();
  InitGPIO();
  offRGB();
  // ...

  osKernelInitialize();
  osThreadNew(app_main, NULL, NULL);
  osKernelStart();
  for (;;) {}
}

```

a. When we call osDelay() what happens to the app_main() task?

Answer:

It transitions to the Blocked state for the duration of 1000 ticks. By default, the system is configured such that 1 tick is 1ms. So it will translate to a delay of 1ms.

b. What will the CPU execute during that delay time?

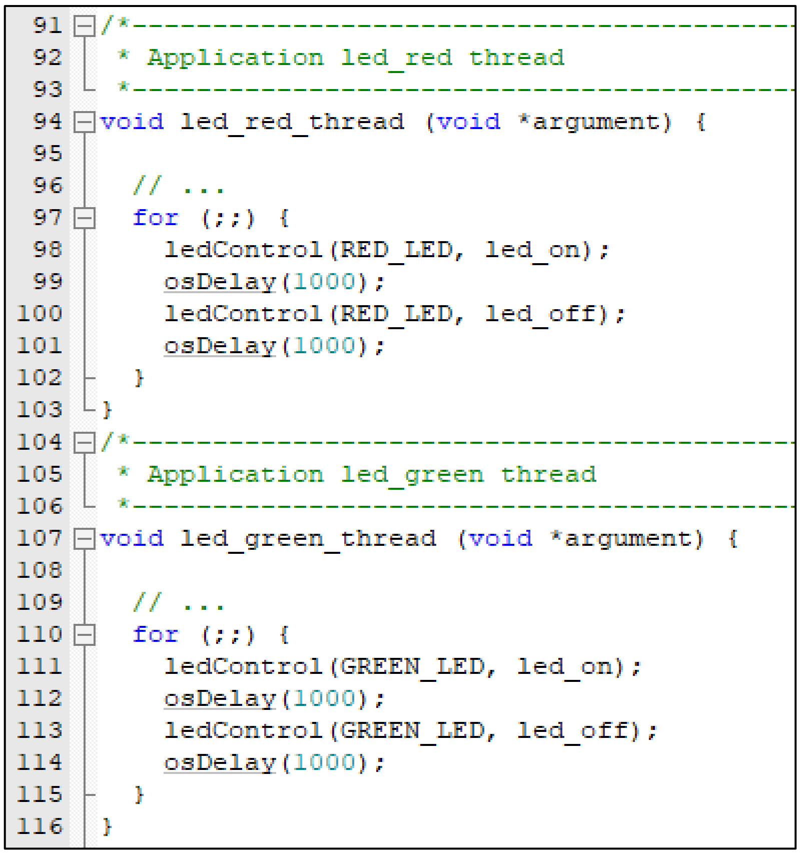
Answer: The OS has its own Idle Thread that is not shown to the developer. This thread will be run if there are no other threads that can make use of the CPU.

c. If we use a normal delay() routine like what you have been doing so far, will we see the same effect?

Answer: In terms of the output, yes, for this simple case. However, the significant difference is that with osDelay() the CPU is freed up to do other important activities. With your normal delay(), you are still making use of the CPU to execute code in order to generate the required delay.

Q4. Double Blinky

The following code snippet shows you TWO tasks each controlled a single colour of the led.



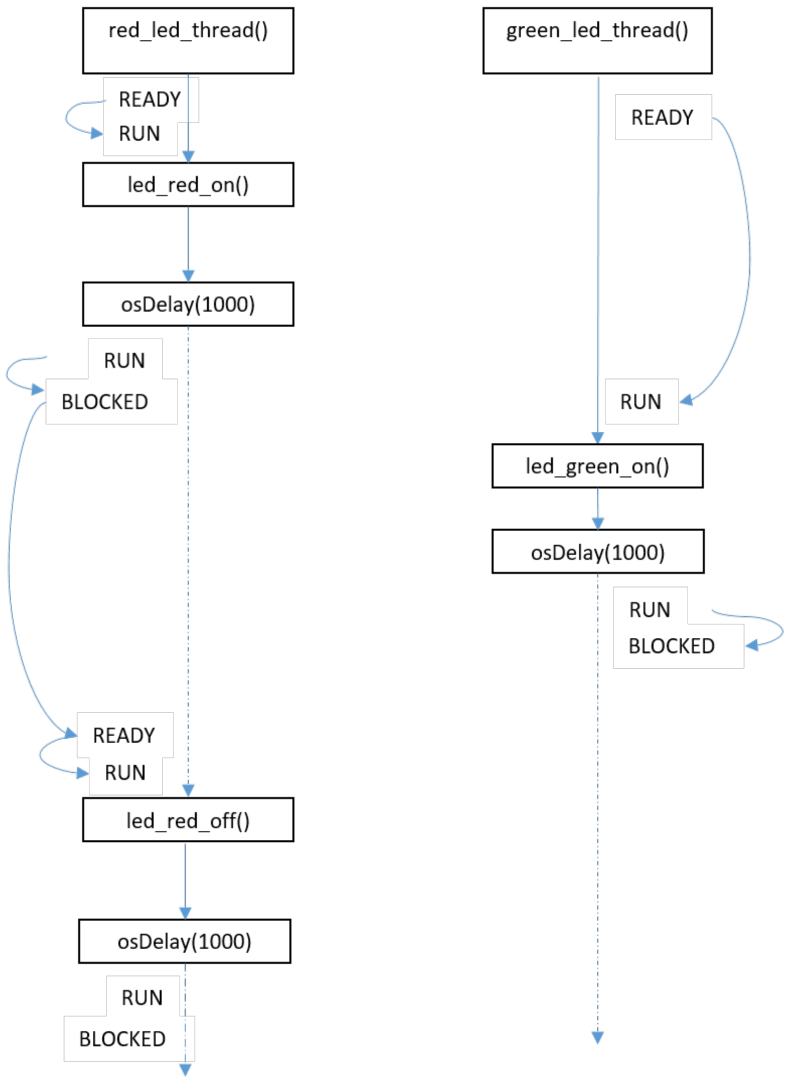
```c
/*----------------------------------------------
 * Application led_red thread
 *----------------------------------------------
void led_red_thread (void *argument) {

  // ...
  for (;;) {
    ledControl(RED_LED, led_on);
    osDelay(1000);
    ledControl(RED_LED, led_off);
    osDelay(1000);
  }
}
/*----------------------------------------------
 * Application led_green thread
 *----------------------------------------------
void led_green_thread (void *argument) {

  // ...
  for (;;) {
    ledControl(GREEN_LED, led_on);
    osDelay(1000);
    ledControl(GREEN_LED, led_off);
    osDelay(1000);
  }
}
```

a. What would be the expected behaviour?

Answer:

Both the RED and GREEN LED’s would light up together (for most part of the time). This is because of the context switching that happens the moment we call the osDelay().

b. Can you draw a timeline to show what happens. Your timeline must clearly show the state of the tasks as they are executing.


red_led_thread()
READY
RUN
led_red_on()
osDelay(1000)
RUN
BLOCKED
READY
RUN
led_red_off()
osDelay(1000)
RUN
BLOCKED
green_led_thread()
READY
RUN
led_green_on()
osDelay(1000)
RUN
BLOCKED