

**Universidad Nacional de Córdoba.**  
**Facultad de Ciencias Exactas, Físicas y**  
**Naturales.**



**SISTEMAS OPERATIVOS II**

**Trabajo Práctico N° 4:**  
***Tiempo real - FreeRTOS***

**Alumno:**

- GOLDMAN, Nicolás.

**Fecha de entrega:**

06 / 06 / 2019

# INDICE

<b>Introducción</b>	<b>2</b>
<b>Objetivo</b>	<b>2</b>
<b>Desarrollo</b>	<b>2</b>
<b>Implementación</b>	<b>3</b>
<b>Conclusión</b>	<b>10</b>

# 1. Introducción

Toda aplicación de ingeniería que posea requerimientos rigurosos de tiempo, y que esté controlado por un sistema de computación, utiliza un Sistema Operativo de Tiempo Real (RTOS, por sus siglas en inglés). Una de las características principales de este tipo de SO, es su capacidad de poseer un kernel preemptive y un scheduler altamente configurable. Numerosas aplicaciones utilizan este tipo de sistemas tales como aviónica, radares, satélites, etc. lo que genera un gran interés del mercado por ingenieros especializados en esta área.

## 2. Objetivo

El objetivo del presente trabajo práctico es que el estudiante sea capaz de diseñar, crear, comprobar y validar una aplicación de tiempo real sobre un RTOS.

## 3. Desarrollo

Se pide que, sobre un sistema embebido de arquitectura compatible con FreeRTOS (ej: ARM Cortex M4):

1. Se instale y configure FreeRTOS en el sistema embebido seleccionado.
2. Crear un programa con dos tareas simples (productor/consumidor) y realizar un análisis completo del Sistema con Tracealyzer (tiempos de ejecución, memoria).
3. Diseñe e implemente una aplicación que posea dos productores y un consumidor. El primero de los productores es una tarea que genera strings de caracteres de longitud variable (ingreso de comandos por teclado). La segunda tarea, es un valor numérico de longitud fija, proveniente del sensor de temperatura del embebido. También que la primer tarea es aperiódica y la segunda periódica definida por el diseñador.

Por otro lado, el consumidor, es una tarea que envi´a el valor recibido a la terminal de una computadora por puerto serie (UART). Y nuevamente, realizar un an´alisis del sistema con Tracealyzer.

## 4. Implementaci3n

Para la soluci3n del cuarto trabajo pr´actico de la materia, se realiz3 primero una investigaci3n sobre los diferentes sistemas operativos de RT, su funcionamiento y sus ventajas sobre los sistemas operativos convencionales.

Una vez con conocimientos sobre el tema se comenz3 a implementar cada uno de los puntos en la consigna del trabajo.

Lo primero que se realiz3 fue la prueba del sistema operativo en la placa designada para el trabajo pr´actico (NXP - LPC1769). Para ello, se re-utiliz3 un proyecto proveniente de los ejemplos de la IDE **mcuxpresso**, utilizada en algunas materias de la carrera para poder realizar la programaci3n de las placas de NXP. El proyecto con el cual se comenz3 era un simple “blink led”. El mismo se basa en generar una simple tarea encargada de encender uno de los led de la placa y apagarlo la pr3xima vez que se ejecuta dicha tarea. Lo importante no es la funci3n del mismo sino el uso del FreeRTOS para su desarrollo.

Lo importante fue poder ver conceptos como la configuraci3n de la placa para poder utilizar el sistema operativo de TR, la creaci3n de las tareas mediante la funci3n **xTaskCreate** y otras funciones como **vTaskDelay** para establecer el tiempo entre que se ejecuta la tarea una y otra vez y **vTaskStartScheduler** para iniciar el funcionamiento del programador de dichas tareas.

Tan pronto como se implement3 dicho proyecto en la LPC y se comprob3 el funcionamiento, se comenz3 a desarrollar el c3digo correspondiente a la primer consigna del trabajo: programa con dos tareas simples (productor/consumidor). Se pensaron varias formas de lograrlo haciendo uso de variables globales que sean modificadas por ambas tareas. La decisi3n final fue la de hacer uso de una variable de

tipo `char[]` que representa un string al cual el productor le agrega elementos y el consumidor retira los mismos.

En las imágenes de abajo se muestran los códigos correspondientes a ambas tareas ejecutadas por el sistema operativo de tiempo real.

```

18 /* PRODUCTOR */
19 static void producer(void *pvParameters) {
20
21     int index = 1;
22     char result[5];
23
24     while (1) {
25
26         sprintf(result, "%d", index);
27         strcat (sharedBuffer, result);
28
29         index++;
30
31         printf("P -> %s\r\n", sharedBuffer);
32
33         vTaskDelay(configTICK_RATE_HZ);
34
35     }
36 }
37

```

```

38 /* CONSUMIDOR */
39 static void consumer(void *pvParameters) {
40
41     while (1) {
42
43         sharedBuffer[strlen(sharedBuffer)-1] = '\0';
44         sharedBuffer[strlen(sharedBuffer)-2] = '\0';
45         sharedBuffer[strlen(sharedBuffer)-3] = '\0';
46
47         printf("C -> %s\r\n", sharedBuffer);
48
49         /* About a 3s delay here */
50         vTaskDelay(3*configTICK_RATE_HZ);
51
52         /* About a 1Hz period */
53         //vTaskDelay(80 / portTICK_RATE_MS);
54     }
55 }

```

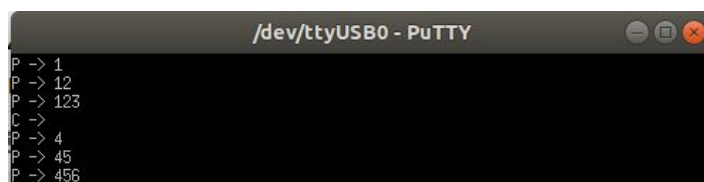
Figura 1. Productor y consumidor simple del trabajo práctico 1

En la **Figura 1** podemos ver el funcionamiento de ambos códigos. El mismo es muy simple, tenemos una tarea “producer” la cual se encarga de agregar un número de índice a la variable global *sharedBuffer* y luego manda por UART dicha variable para poder ver los resultados en una terminal. Esta tarea tiene un **delay** de 1 segundo:

```
#define configTICK_RATE_HZ      ( ( portTickType ) 1000 )
```

Por otra parte tenemos la tarea del consumidor. Para poder lograr una buena visualización del programa, se realizó una tarea encargada de encontrar los últimos tres valores del *sharedBuffer* y reemplazarlos por ‘\0’, “borrando” dichos elementos del arreglo de char.

Esta tarea tiene un **delay** del triple al productor, logrando que este último genere 3 elementos para el buffer siendo inmediatamente consumidos; enviando el resultado final por UART con una C en el comienzo para poder lograr el siguiente resultado:



```

/dev/ttyUSB0 - PuTTY
P -> 1
P -> 12
P -> 123
C -> 
P -> 4
P -> 45
P -> 456

```

Una vez que realizado dicho programa capaz de simular dos tareas y comprobado su correcto funcionamiento, se pasó a la segunda consigna del trabajo la cual tenía el agregado de un segundo productor. Para poder realizar esto, se utilizó un concepto muy importante a la hora de hablar de sistemas operativos de tiempo real. Este es el concepto de colas (o queues).

La cola fue el recurso que se utilizó en este concepto para poder lograr tener dos tareas “produciendo” al mismo tiempo sin que se interpongan entre ellas a la hora de mostrar los resultados con un solo consumidor. Los servicios entre productores y consumidores compartirán una estructura la cual contendrá dos elementos; uno que llevará el registro de temperatura y otro que lo hará con el del string de tamaño variable. Más adelante se explicará la forma en que se crean las diferentes estructuras por cada uno de los productores y consumidor, y cómo se utilizan en conjunto con las colas.

```
typedef struct {
    int temp;
    char * str;
} QueueData;
```

Las tres tareas, a diferencia de la primer consigna, especifican cuál debe ser la acción que debe realizar cada una de ellas. Se explicarán por separado en diferentes secciones.

### Productor 1. Generador de temperatura.

El primer productor, es el encargado de generar un entero aleatorio que simularía el sensado de temperatura en un ambiente.

```
33= static void temperatureGenerator() {
34
35     QueueData * temperatureQue;
36     temperatureQue = malloc(sizeof(QueueData *));
37     temperatureQue->str = malloc (sizeof (SIZE));
38     temperatureQue->str = NULL;
39
40     int temperature = 0;
41     while (1) {
42
43         //rand () % (N-M+1) + M; // Este está entre M y N
44         // temperature = rand() % 10 + 20; // Entre 24°C y 25°C
45
46         temperature += 1;
47         temperatureQue->temp = temperature;
48         xQueueSend(xQueue1, (QueueData *) &temperatureQue, 5000);
49         //printf("Temperatura de la funcion temp --> %02d \r\n", temperature);
50         vTaskDelay(5*configTICK_RATE_HZ);
51     }
52 }
53 }
54
```

Como se puede observar en la figura de arriba, lo primero que tenemos en nuestra tarea es la creación de un puntero a una estructura. Como se mostró previamente, dicha estructura almacenará dos variables; una de tipo entero para la temperatura y otra de tipo puntero a char para el string.

Como en este caso nos encontramos en la producción de temperatura, se setea el valor de la variable \*char en NULL (se explicará el por qué en la sección del consumidor). Una vez que ya tenemos este puntero a la estructura creado, entramos al ciclo que se encarga de generar un numero random entre 20 y 30 que simularán esa misma cantidad de grados centígrados (en la imagen de arriba, dicha línea está comentada por estar haciendo pruebas de UART).

En este punto estamos en condiciones de enviar los datos a la cola del consumidor.

```
temperatureQue->temp = temperature;
xQueueSend(xQueue1, (QueueData *) &temperatureQue, 5000);
```

Como se puede ver en la imagen de arriba, tenemos la primer línea en la que almacenamos el valor random generado (temperature), dentro del int de la estructura QueueData. Inmediatamente se envía dicha estructura a la cola con la sentencia:

```
xQueueSend(created_queue, adress_of_data_parsed, time_to_block).
```

Como el envío de dicha información se debe realizar de forma periódica, se estableció un tiempo de 5 segundos de delay para que se vuelva a ejecutar dicho task (delay modificado a la hora de usar Tracelyzer).

## **Productor 2. Generador de strings de longitud variable.**

El segundo caso del productor consiste en algo muy parecido al recién explicado, con la diferencia de que ahora lo que se genera es un string de longitud variable como si se estuviera simulando una entrada de teclado al consumidor.

Por esta misma razón es que no se entra tanto en detalle con los puntos explicados con anterioridad, y se detalla la forma en que se generó dicha cadena.

```

58 static void stringGenerator() {
59
60     QueueData * stringQue;
61     stringQue = malloc (sizeof (QueueData *));
62     stringQue->str = malloc (SIZE);
63     stringQue->temp = NULL;
64
65     srand((unsigned int)(time(NULL)));
66
67     char charVocabulary[] = "abcdefghijklmnopqrstuvwxyz0123456789";
68     int stringLength = 0;
69     float randTime = 0;
70     char finalStr[SIZE] = "";
71
72     while(1){
73
74         stringLength = rand() % 5 + 2; //Entre 2 y 6 caracteres
75         randTime = rand() % 2 + 1;
76         for (int i = 0; i < stringLength; i++){
77             finalStr[i] = charVocabulary[rand() % (sizeof (charVocabulary) - 1)];
78         }
79
80         finalStr[stringLength] = '\0';
81
82         strcpy(stringQue->str, finalStr);
83
84         xQueueSend(xQueue1, (QueueData *) &stringQue, 5000);
85
86         for (int i = 0; i < stringLength; i++){
87             finalStr[i] = '\0';
88         }
89
90         vTaskDelay(randTime * configTICK_RATE_HZ);
91
92     }
93 }
94
95 }

```

La imagen de arriba muestra el código correspondiente a la tarea generadora de strings. En ella podemos ver el comienzo de la misma manera que el caso anterior, en el que declaramos un puntero a la estructura que contendrá los datos, los comandos para alocar memoria y el seteo de temperatura en este caso como NULL.



La forma en que se decidió formar la palabra es primero generando un entero aleatorio que será el límite del bucle **for** en el cual se agregan letras al final de la palabra. Estos caracteres que se van agregando lo hacen seleccionando de manera aleatoria alguno de los elementos del arreglo de chars *charVecabulary*, contenedor de todos los caracteres deseados.

Una vez que salimos del for, en **finalStr** tenemos una palabra de longitud variable creada a partir de los caracteres del vocabulario; y hacemos un **strcpy** para pasar dicho valor al puntero de la estructura. Finalmente lo enviamos a la cola y borramos finalStr para la próxima vez que se ejecute dicha tarea, que será en un tiempo random generado de la misma forma que los anteriores casos de aleatoriedad.

### Consumidor. Envío de los datos del productor a UART.

La última tarea que se debió realizar era aquella que reciba correctamente los datos de las colas llenadas por los productores y los envíe a la terminal de la computadora a través de UART.

Para ello se desarrolló el siguiente pedazo de código:

```

102 static void consumidor(void *pvParameters) {
103
104     QueueData * recQue;
105     recQue = malloc (sizeof (QueueData *));
106     recQue->str = malloc (SIZE);
107
108     while (1) {
109
110         if(xQueueReceive(xQueue1, &recQue, 5000)){
111
112             if(recQue->str != NULL){
113                 printf("String-> %s\r\n", recQue->str);
114             }
115             if(recQue->temp != NULL){
116                 printf("Temperature -> %2d\r\n", recQue->temp);
117             }
118         }
119     }
120 }
121 }

```

En la figura de arriba podemos ver la forma en que se implementó el consumidor. Cuenta con un puntero a la estructura de la misma forma que los dos productores, y simplemente implementa un **xQueueReceive** detallando como buffer donde se recibirán los datos dicho puntero a la estructura QueueData.

Si quisiéramos resolver el problema sin hacer uso de las colas, la respuesta sería la siguiente:



```
/dev/ttyUSB0 - PuTTY
String -> H7T/G
String -> xUgc
String -> J;mTemperatureiff
-> 25[°C]
String -> TMdTemperaturep5f
-> 24[°C]
String -> nrVTemperaturecfQ
-> 25[°C]
String -> PB4TemperatureNeH
-> 24[°C]
```

Como se explicó anteriormente, cuando se entraba a alguna de las tareas de los productores, el valor correspondiente al restante productor se seteaba en NULL. Esto es así para que en la sección del consumidor se pueda saber con precisión cuál fue el último en producir un dato y enviarlo a la cola simplemente con detectar cual de los dos valores de la estructura no es nulo.

De esta forma, una vez que sabemos cuál fue el último productor, enviamos a UART el valor producido junto con un prefijo que nos afirmará correspondientemente si es un dato de temperatura o de string aleatorio.

```

/dev/ttyUSB0 - PuTTY
Temperature -> 21 °C
String-> tfv6d1cwm42nouewab
Temperature -> 20 °C
Temperature -> 25 °C
String-> nj114a9rcwvdp7uele0t
Temperature -> 26 °C
Temperature -> 28 °C
Temperature -> 22 °C
Temperature -> 27 °C
Temperature -> 22 °C
String-> x6jhg9vuelqlg7
Temperature -> 26 °C
Temperature -> 29 °C
String-> uiwvaxfi6nt
Temperature -> 28 °C
Temperature -> 27 °C
String-> 1piy9lt0qe2
Temperature -> 27 °C
Temperature -> 28 °C
Temperature -> 28 °C
String-> oul8ysokqc4duldr658k

```

Por último, se muestra en la siguiente imagen el código correspondiente a la creación de las tareas en el **main**:

```

xTaskCreate(temperatureGenerator, (signed char *) "vTaskP1",
            configMINIMAL_STACK_SIZE, NULL, (tskIDLE_PRIORITY + 3UL),
            (xTaskHandle *) NULL);

xTaskCreate(stringGenerator, (signed char *) "vTaskP2",
            configMINIMAL_STACK_SIZE, NULL, (tskIDLE_PRIORITY + 2UL),
            (xTaskHandle *) NULL);

xTaskCreate(consumidor, (signed char *) "vTaskC1",
            configMINIMAL_STACK_SIZE, NULL, (tskIDLE_PRIORITY + 1UL),
            (xTaskHandle *) NULL);

```

En ella podemos ver que se crea una tarea de tiempo real para cada uno de los productores y consumidor, y se setean las prioridades en caso de tener dos al mismo tiempo intentando ejecutarse.

## 5. Análisis en Tracealyzer

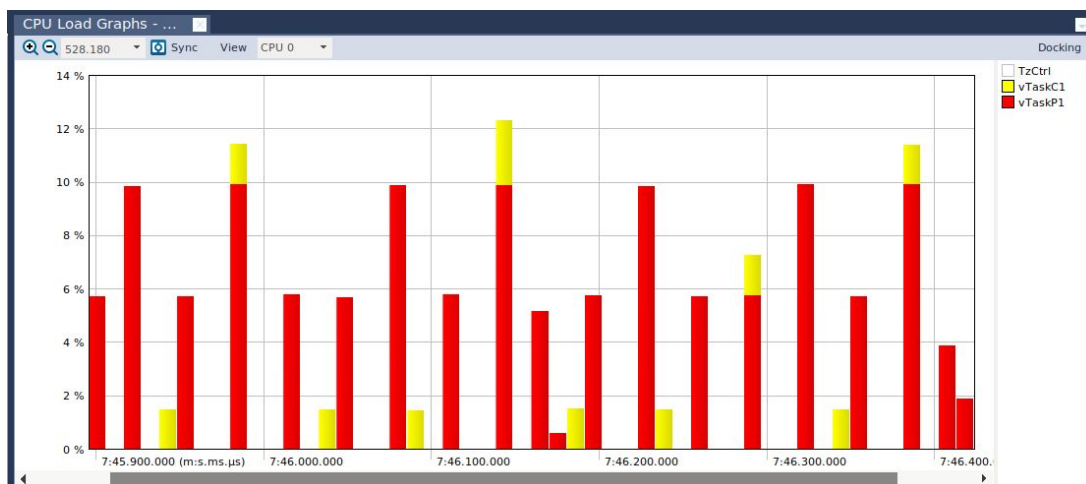
Una vez que comprobado el funcionamiento de ambos programas, se decidió pasar a la parte de análisis de tiempos de ejecución y performance mediante el software *Tracealyzer*. Para ello, previamente se debieron agregar y modificar algunos archivos de configuración del proyecto.

Además se debieron modificar todos los tiempos entre tareas ya que cuando no se realizaba el análisis, se deseaba visualizar de forma “estética” los resultados por UART, mientras que en Tracealyzer con tiempos de 1 segundo o más, no se llegan a ver los cambios de contexto.

### Análisis 1 productor - 1 consumidor.

Para poder entender correctamente el funcionamiento y resultados devueltos por el Tracealyzer, se hicieron varias pruebas, variando entre ellas cuestiones como el **delay**, **prioridades**, **periodicidad de las tareas**, etc. Se muestra una de ellas para cada programa.

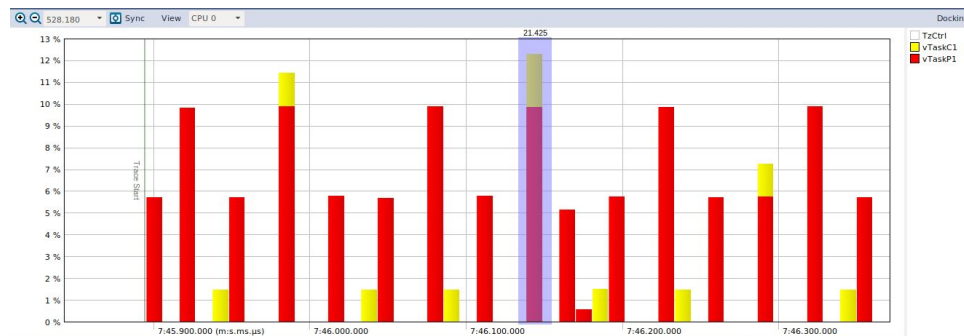
La primer prueba realizada fue con 30 [ms] para el productor y 40 [ms] para el consumidor. Los resultados obtenidos fueron los siguientes:



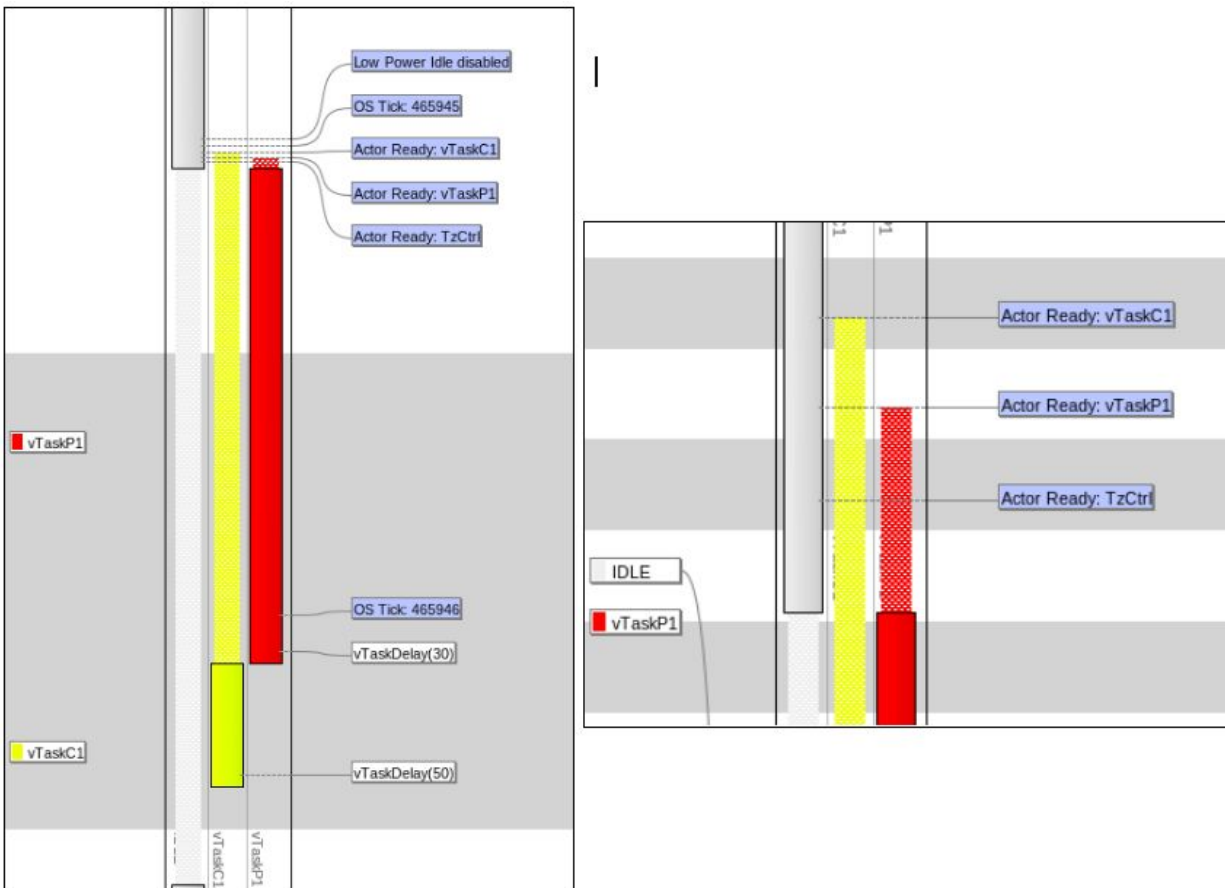
Lo primero que deseamos analizar es el **uso del CPU** a medida que va corriendo el programa. Podemos ver ambas tareas diferenciadas por su color, siendo el las amarillas correspondientes al consumidor y las rojas al productor.

Podemos notar un uso mayor del CPU por parte del productor. Esto se debe a las funciones que se ejecutan dentro del mismo. Mientras el productor debe llevar la cuenta de un índice, agregar este último al final de una variable temporal y luego copiar dicha variable temporal en otra global de tipo `char[]`, el consumidor únicamente realiza un **memset** para borrar todos los elementos de dicha variable.

A su vez, nos interesa analizar los tiempos de ejecución y las prioridades de nuestras tareas. Para hacerlo, encontramos algún caso en que tengamos las dos tareas intentado tomar al procesador (instancia 9/18 en P y 5/10 en C) y explicaremos lo que sucede en



dicha situación.

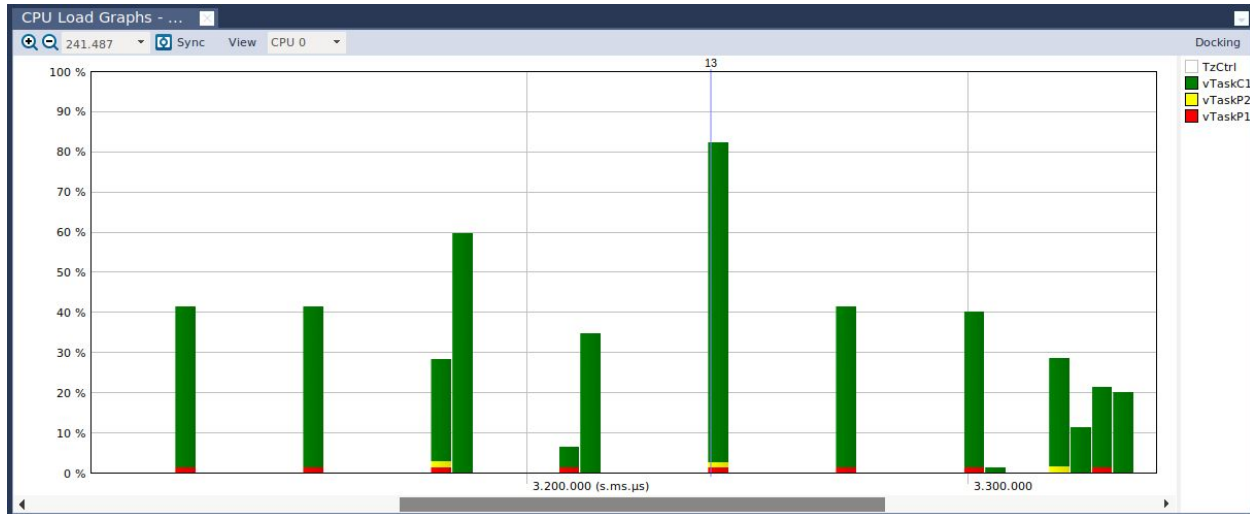


En las imágenes de arriba, tenemos los resultados de dicho análisis. En ellas podemos ver que nuestro procesador viene en estado de IDLE hasta el Tick de SO 465945. Es aquí cuando nos llega el Actor Ready de la tarea del consumidor. Aproximadamente 10 [μs] después, llega el mismo mensaje pero ahora de la tarea del productor.

Aquí es cuando vemos las prioridades. Como el productor tiene seteada una prioridad más alta que el consumidor, entonces el procesador asigna la tarea P1, mientras que deja en espera a la del consumidor. Una vez finalizada la tarea del productor, ejecuta el vTaskDelay y pone a correr la tarea del consumidor.

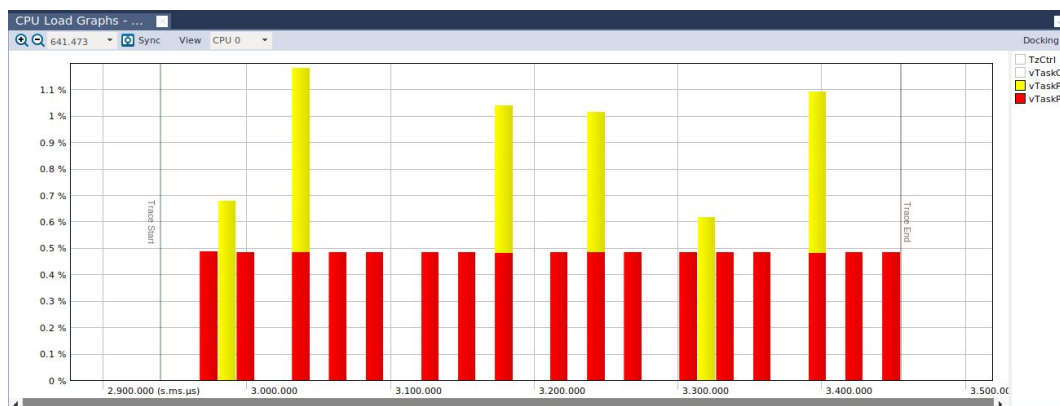
## 2 productores - 1 consumidor.

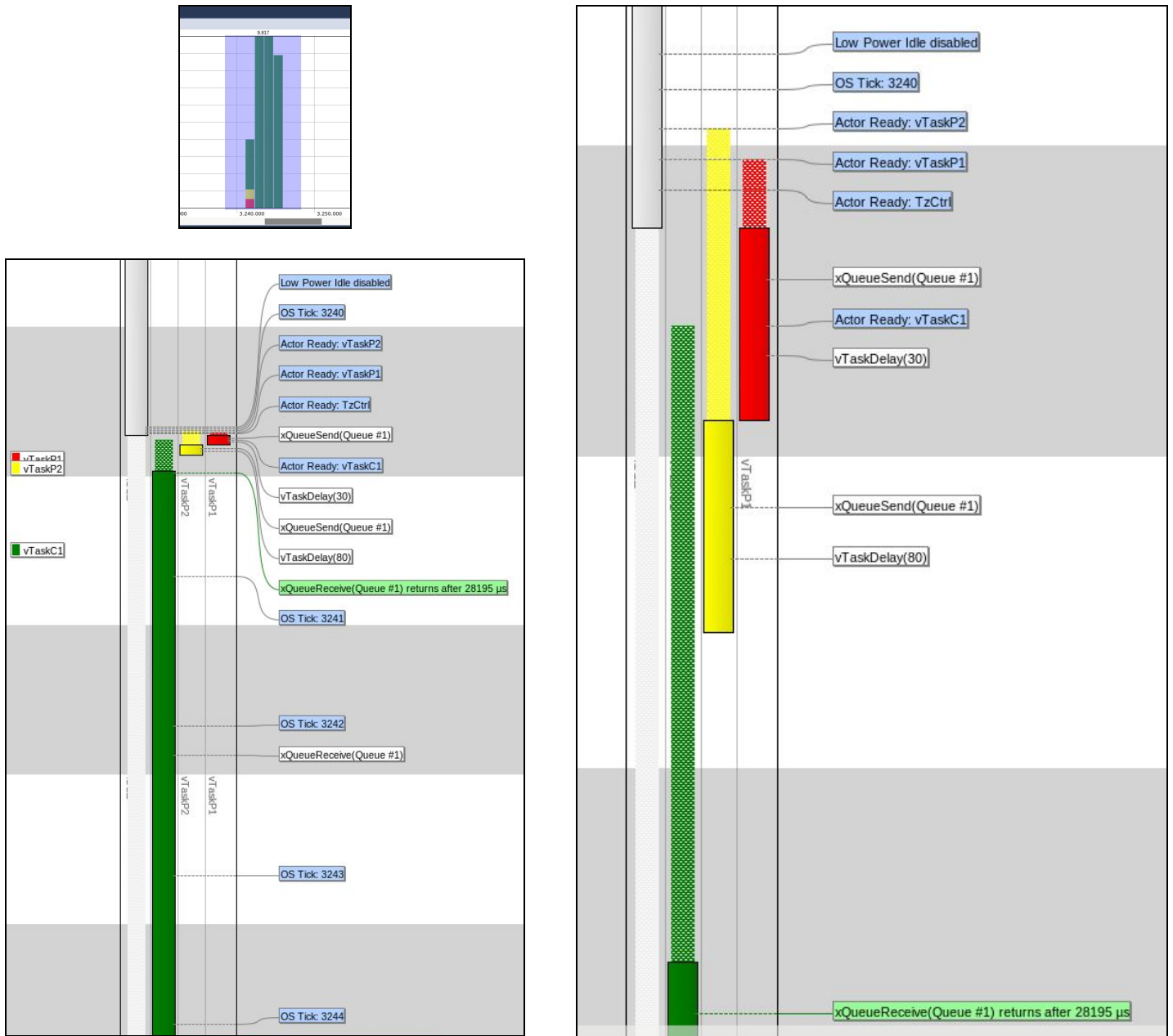
De la misma forma que para el productor-consumidor simple, comenzamos analizando el uso de CPU de las tareas que ejecutamos en nuestro programa.



En la imagen de arriba podemos observar como la tarea que mayor uso de CPU requiere en la ejecución, es la del *consumidor*. Esta tarea se ejecuta al mismo tiempo que alguno de los productores, y requiere mayor uso del CPU cuando recibe un dato de **vTaskP2** siendo esta, la tarea que genera el string (de mayor longitud al entero de temperatura).

Para poder ver con mayor claridad los usos entre los dos productores tenemos la imagen de abajo, que nos muestra las dos tareas; una que requiere la misma cantidad de procesamiento cada vez que se ejecuta (debido a que es un entero que no varía su tamaño entre ejecuciones) y es a su vez periódica; y otra tarea aperiódica cuyo uso de cpu varía con los cambios de tamaño de string del generador de comandos.





En las imágenes de arriba podemos ver uno de los casos en los que las tres tareas solicitaron al cpu para ejecutarse y la prioridad de cada una de ellas determinó el orden.

Se puede ver que la primer tarea en pasar a estado “*ready*” es la del productor de strings, sin embargo, antes de que el procesador le entregue los recursos, llega la tarea del productor de temperatura con mayor prioridad y toma el cpu para ejecutarse.



Por último llega la tarea del consumidor, encargada de recibir los valores de las colas e imprimirlos por UART en la terminal de la computadora y se ejecuta sin problemas.

## 6. Conclusión

El cuarto trabajo práctico de la materia me permitió comprender y observar el funcionamiento de un sistema operativo de tiempo real, darme una idea de cómo funciona su scheduler, dándole importancia a la previsibilidad y no tanto a la rapidez, y a generar tareas periódicas y aperiódicas. Además se pudo hacer uso de una herramienta para el diagnóstico y seguimiento como es Tracealyzer

6.

## Conclusión