Cátedra de Sistemas Operativos II

Trabajo Práctico N° III

Piñero, Tomás Santiago

25 de Junio de 2020

Índice

1. Introducción

1.1. Objetivo

El objetivo del trabajo es diseñar, crear, comprobar y validar una aplicación de tiempo real sobre un RTOS.

1.2. Definiciones, Acrónimos y Abreviaturas

- *RTOS*: *Real Timer Operative System*. Sistema operativo en tiempo real.
- *FreeRTOS*: es un sistema operativo en tiempo real para microcontroladores y pequeños microprocesadores. [1]
- *qemu*: es un emulador y virtualizador de procesadores genérico. [8]
- *UART*: Transmisor-receptor universal asíncrono.

2. Descripción general

2.1. Esquema del proyecto

El proyecto está dividido en las siguientes carpetas:

- **FreeRTOS**: contiene los archivos necesarios para la ejecución de *FreeRTOS*. Los archivos fuentes de este último se encuentran en el subdirectorio `Source`, mientras que los archivos del trabajo práctico se encuentran en el subdirectorio `Demo/src`.
- **informe**: contiene los archivos correspondientes al informe.

2.2. Requisitos futuros

- Mejorar la estructura de los directorios.
- Realizar el trabajo práctico en una `LPC769`.

3. Diseño de solución

Como se mencionó en la sección anterior, se organizó el proyecto en varias carpetas (ver **2.1**).

Para realizar el práctico primero se consultó la documentación de *FreeRTOS* [2] y el libro *Mastering the FreeRTOS Real Time Kernel, A hands-on tutorial guide* [3]. Consecuentemente, se decidió utilizar el manejo de memoria dinámico `heap_4.c`[4], que utiliza el método *Firs Fit* para la asignación y es el recomendado.

La comunicación de las tareas se realiza mediante colas[5], por lo que habrá una única cola en todo el sistema.

3.1. Alnálisis del *scheduler*

La documentación de *FreeRTOS* explica que *scheduler* siempre elige la tarea con mayor prioridad que esté lista para ejecutarse (*Ready state*) y, en caso de existir dos tareas con la misma prioridad, utiliza el método *Round Robin* alternando entre ellas.

Para verificar esto, se utilizó la tarea *top* que muestra el estado del sistema con la misma prioridad que la tarea con máxaima prioridad. En el caso de este trabajo, las tareas no abandonan el procesador por sí solas, realizando que el *scheduler* funcione simplemente por nivel de prioridad.

A continuación, se muestra el estado del sistema con el consumidor con mayor prioridad sobre los productores:

```
------------------Task State------------------
Task            State   Prio    Stack   Num
----------------------------------------------
top             X       4       14      6
C1              R       4       2       1
P1              R       2       30      2
P2              R       2       30      3
P3              R       2       30      4
P4              R       2       30      5
IDLE            R       0       30      7
----------------------------------------------

----------------Runtime Stats-----------------
Task            ABS             %CPU
----------------------------------------------
top             74              <1%
C1              299626          99%
P2              0               <1%
P3              0               <1%
P4              0               <1%
P1              0               <1%
IDLE            0               <1%
----------------------------------------------
```

Figura 1: Ejecución del programa con prioridad máxima (4) para C1.

Como puede observarse en la Fig. **1**, el consumidor ocupa todo el tiempo del procesador, debido a que la tarea siempre está lista para ejecutarse y tiene la mayor prioridad, produciéndose así la inanición a los productores.

En caso de tener todas las tareas la misma prioridad, el *scheduler* reparte el tiempo del procesador entre los procesos con el método *Round Robin*.

```
----------------Runtime Stats-----------------
Task            ABS             %CPU
----------------------------------------------
C1              19970           20%
P1              19995           20%
P2              19959           19%
P3              19964           20%
top             11              <1%
P4              19945           19%
IDLE            0               <1%
----------------------------------------------

------------------Task State------------------
Task            State   Prio    Stack   Num
----------------------------------------------
C1              R       2       6       1
P1              R       2       4       2
P2              R       2       6       3
P3              R       2       6       4
top             X       2       14      6
P4              R       2       6       5
IDLE            R       0       30      7
----------------------------------------------
```

Figura 2: Ejecución del programa con todas las tareas con la misma prioridad.

Como se puede observar en la Fig. **2**, todas las tareas tuvieron un mismo tiempo de ejecución debido a que todas ellas comparten la misma prioridad.

3.2. *Stack* mínimo para cada tarea

Mediante la función `uxTaskGetStackHighWatermark` se puede conocer la cantidad de espacio de la pila que no fue utilizado durante la ejecución de la tarea. Este valor se encuentra expresado en palabras (*words*)[6], por lo que en un sistema de 32 bits, una palabra significan 4 bytes de pila no utilizada.

Se iniciará con un valor de 280 bytes (70 en palabras) para cada tarea y se utilizará la función para ajustarlo según la tarea.

3.3. *Top Task*

La tarea tipo *top* de *Linux* puede realizarse con la función `vTaskGetRunTimeStats`, que genera y muestra una tabla con la información sobre el uso de la CPU de cada tarea. Esta tarea se ejecutará cada 5 segundos, pero este tiempo puede ser modificado al momento de ejecución modificando la variable `mainCHECK_DELAY`, cuyo valor se encuentra expresado en milisegundos.

4. Implementación y resultados

Toda muestra de información se realiza por medio del periférico *UART*, que se muestra en consola utilizando la *flag* `-serial mon:stdio` de *qemu*.

Se utilizó la prioridad máxima por defecto `configMAX_PRIORITIES`, cuyo valor es 5 y es utilizada como base para la prioridad de todas las tareas, por lo que la prioridad de la tarea puede ir de 0 a 4.

4.1. Productor

Los productores utilizan el Código **1**, en el que simplemente produce (pone en la cola) su nombre. Su prioridad es 2 (`configMAX_PRIORITIES-3`).

Código 1: Función '*vProducerTask*'.

```
static void vProducerTask( void *pvParameters )
{
  UBaseType_t uxHighWaterMark;
  uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );
  char *pcMessage = pcTaskGetName( NULL );

  for( ;; )
  {
    /* Send message to the queue. */
    if(xQueueSend( xPrintQueue, &pcMessage, 0 ) != pdTRUE)
    {
      ;
      // vUARTPrint(pcMessage);
      // vUARTPrint(" cannot send message.\n\r");
    }

    uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );
  }
}
```

4.2. Consumidor

El consumidor muestra qué valor de la cola ha consumido cada 1 segundo con el siguiente formato:

```
C1 -> 'nombreProductor'
```

Su código es el siguiente:

Código 2: Función '*vConsumerTask*'.

```
static void vConsumerTask( void *pvParameters )
{
  UBaseType_t xStatus;
  char *pcMessage;
  char *pcTaskName = pcTaskGetName( NULL );

```

```
7
8   for( ;; )
9   {
10    // vTaskDelay(1000);
11    /* Wait for a message to arrive. */
12    if(xQueueReceive( xPrintQueue, &pcMessage, 0 ) == pdFALSE)
13    {
14      ;//vUARTPrint("Empty queue.\n\r"); portMAX_DELAY
15    }
16    else
17    {
18      /* Print Consumer name -> Producer name */
19      vUARTPrint(pcTaskName);
20      vUARTPrint(" -> ");
21      vUARTPrint(pcMessage);
22      vUARTPrint("\r\n");
23    }
24  }
25 }
```

4.3. *Top*

La tarea top se llevo a cabo utilizando las funciones *vTaskList* y *vTaskGetRunTimeStats* provistas por *FreeRTOS*. La primera muestra el estado de las tareas del sistema con el siguiente formato:

```
Tarea     Estado     Prioridad     Stack     Numero de tarea
```

La segunda función, muestra el tiempo que cada tarea estuvo utilizando el procesador, tanto en valor absoluto como en porcentaje, de la siguiente forma:

```
Tarea     ABS     %CPU
```

La tarea *top* se ejecuta cada 5 segundos y su código es el siguiente:

Código 3: Función '*vTopTask*'.

```
1  static void vTopTask( void *pvParameters )
2  {
3    TickType_t xLastExecutionTime;
4    char *buffer = pvPortMalloc(sizeof(char));
5
6    /* Initialise xLastExecutionTime so the first call to vTaskDelayUntil()
7    works correctly. */
8    xLastExecutionTime = xTaskGetTickCount();
9
10   for( ;; )
11   {
12     /* Perform this check every mainCHECK_DELAY milliseconds. */
13     vTaskDelayUntil( &xLastExecutionTime, mainCHECK_DELAY );
14
15
16     /* Imprimir 'top' cada 5 segundos */
```

```
17      vUARTPrint("\n----------------Runtime Stats----------------\n\r");
18      vUARTPrint("Task\t\tABS\t\t%CPU\n\r");
19      vUARTPrint("-----------------------------------------------\n\r");
20
21      vTaskGetRunTimeStats(buffer);
22      vUARTPrint(buffer);
23      vUARTPrint("-----------------------------------------------\n\r");
24
25      vUARTPrint("\n-----------------Task State-----------------\n\r");
26      vUARTPrint("Task\t\tState\tPrio\tStack\tNum\n\r");
27      vUARTPrint("-----------------------------------------------\n\r");
28      vTaskList(buffer);
29      vUARTPrint(buffer);
30      vUARTPrint("-----------------------------------------------\n\r");
31    }
32  }
```

4.4. Resultados

```
C1 -> P1
C1 -> P2
C1 -> P3
C1 -> P4
C1 -> P1
C1 -> P2

---------------Runtime Stats----------------
Task            ABS             %CPU
-----------------------------------------------
top             13              <1%
P4              3095            3%
P1              3040            3%
P2              3102            3%
P3              3104            3%
IDLE            0               <1%
C1              85032           87%
-----------------------------------------------

-----------------Task State-----------------
Task            State   Prio    Stack   Num
-----------------------------------------------
top             X       4       18      6
P1              R       2       4       2
P2              R       2       4       3
P3              R       2       4       4
P4              R       2       4       5
IDLE            R       0       30      7
C1              B       4       1       1
-----------------------------------------------
C1 -> P1
C1 -> P2
```

Figura 3: Ejecución del programa.

La Fig. **3** muestra la ejecución del programa, donde se puede observar el orden en que los productores lograron acceder a la cola. Pasados los 5 segundos, se imprimen las estadísticas de las tareas del sistema. La única ejecutándose es la tarea *top*, ya que para obtener estas estadísticas, *FreeRTOS* bloquea la tarea Consumidor, que tiene la misma prioridad, mientras que las demás están listas para ejecutarse.

5. Conclusiones

Utilizando la función `uxTaskGetStackHighWatermark` y la tarea *top* se pudo definir el tamaño de pila necesario para cada tarea, si bien la documentación oficial recomienda no cambiar el valor por defecto, los cambios realizados no afectaron el funcionamiento del programa.

Respecto al trabajo práctico, lo que más dificultad generó fue la tarea *top*, dado que las funciones leídas de la documentación no funcionaban como uno esperaba. Su implementación llevó tiempo debido a esto y a que se tuvo que llegar a conocer la placa con la que se estaba trabajando.

ANEXO A Guía paso a paso

Para poder ejecutar el programa deben ejecutarse los siguientes comandos (la ejecución se detiene presionando `Ctrl+A X`):

A.1 *Github*

```
$ git clone https://github.com/Sistemas-Operativos-II-2020/so2-
   tp4-rtos-tomasspi.git
$ cd so2-tp1-rtos-tomasspi/FreeRTOS/Demo/src
$ make
$ qemu-system-arm -machine lm3s811evb -cpu cortex-m3 -nographic
    -serial mon:stdio -kernel gcc/RTOSDemo.axf -gdb tcp::3333
```

A.2 Archivo comprimido

Cambiar la letra n por la ñ en 'Pinero'.

```
$ tar -zxvf TP4_TPinero_39445871.tar.gz
$ cd TP4_TPinero_39445871/FreeRTOS/Demo/src
$ make
$ qemu-system-arm -machine lm3s811evb -cpu cortex-m3 -nographic
    -serial mon:stdio -kernel gcc/RTOSDemo.axf -gdb tcp::3333
```

ANEXO B *Debugging*

Para realizar un *debugging* se deben ingresar los siguientes comandos desde el directorio `src`:

```
$ gdb-multiarch gcc/RTOSDemo.axf
$ target remote localhost:3333
```

A partir de ese momento, pueden utilizarse los comandos de *gdb*[9].

Referencias

[1] *FreeRTOS*,
`Link to FreeRTOS`

[2] *FreeRTOS, API Reference*,
`Link to FreeRTOS API Reference`

[3] *Richard Barry, Mastering the FreeRTOS Real Time Kernel, A hands-on tutorial guide*,
`Link to Mastering FreeRTOS`

[4] *FreeRTOS, Memory Management*,
`Link to Memory Management`

[5] *FreeRTOS, Queues*,
`Link to Queues`

[6] *FreeRTOS, Documentation*,
`Link to uxTaskGetStackHighWaterMark`

[7] *FreeRTOS, Documentation*,
`Link to uxTaskGetSystemState`

[8] *qemu*,
`Link to qemu`

[9] *rkubik, gdb cheatsheet*,
`Link to gdb cheatsheet`