Trabalho Prático 2 - Parte 2
Um Estudo sobre o FreeRTOS e suas APIs

Guilherme Philippi
Campus Blumenau
Universidade Federal de Santa Catarina
UFSC
guilherme.philippi@grad.ufsc.br

25 de junho de 2019

Sumário

1 Sincronização de processos utilizando filas

A implementação que se segue esboça um problema clássico de sincronização de processos, o produtor/consumidor. Teve-se como objetivo implementar um grupo de tasks produtoras de conteúdo que é desenhado em uma tela por outra task consumidora. Note que esta é uma implementação prática, feita na IDE Arduino, utilizando um Arduino Mega, um display OLED GM009605 e um Joystick bidimensional modelo KY-023.

Segue o exemplo da solução deste problema utilizando a linguagem C:

```
1 #include <Arduino_FreeRTOS.h>
2 #include "U8glib.h"
3 #include "queue.h"
4
5 void Update( void *pvParameters );
6 void UpdateBullet( void *pvParameters );
7 void Draw( void *pvParameters );
8 QueueHandle_t mutex, queue;
9 struct Data
10 {
11     float x, y;
12     int id;
13 };
```

```
14
15
16 U8GLIB_SSD1306_128X64 u8g(U8G_I2C_OPT_NO_ACK); // model of display
17  const uint8_t bullet[] U8G_PROGMEM = {
18   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
19   0x00, 0x3c, 0x00, 0xff, 0xff, 0xc0, 0xff, 0xff, 0xc0,
20   0x00, 0x3c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
21   0x00, 0x00, 0x00,
22 };
23
24 const uint8_t smile[] U8G_PROGMEM = {
25   0x07, 0xfe, 0x00, 0x1e, 0x07, 0x80, 0x3c, 0x03, 0xc0,
26   0x70, 0x00, 0xe0, 0x60, 0x00, 0x60, 0xe0, 0x00, 0x70,
27   0xc6, 0x06, 0x30, 0x87, 0x0e, 0x10, 0x86, 0x06, 0x10,
28   0x80, 0x00, 0x10, 0x80, 0x00, 0x10, 0x88, 0x01, 0x10,
29   0x8c, 0x03, 0x10, 0xce, 0x07, 0x30, 0xe7, 0x0e, 0x70,
30   0x63, 0xfc, 0x60, 0x70, 0x00, 0xe0, 0x3c, 0x03, 0xc0,
31   0x1e, 0x07, 0x80, 0x07, 0xfe, 0x00,
32 };
33
34 void setup() {
35   mutex = xQueueCreate(1, sizeof(int));
36   if(mutex == 0) return;
37   queue = xQueueCreate(10, sizeof(struct Data*));
38   if(queue == 0) return;
39
40   // Now set up two Tasks to run independently.
41   xTaskCreate(
42     Update
43     , (const portCHAR *)"Produtor" // A name just for humans
44     , 128  // This the stack size
45     , NULL
46     , 1  // Priority, with 3 being the highest, and 0 being the lowest.
47     , NULL);
48
49   xTaskCreate(
50     UpdateBullet
51     , (const portCHAR *)"Produtor" // just for humans
52     , 128  // This the stack size
53     , NULL
54     , 1  // Priority
55     , NULL);
56
57
58   xTaskCreate(
59     Draw
60     , (const portCHAR *)"Consumidor"// just for humans
61     , 128  // This the stack size
62     , NULL
63     , 3  // Priority
64     , NULL);
65 }
66
67 void loop()
68 {
69   // Empty. Things are done in Tasks.
70 }
71
```

```
72 void UpdateBullet(void *pvParameters __attribute__((unused)))
73 {
74   /*
75     UpdateBullet
76     Wait for a mutex to fire the bullet
77   */
78   int count = 0;
79   for (;;){
80     if (xQueueReceive(mutex, NULL, (TickType_t)10) == pdTRUE){ //bullet walk
81       count+=5;
82       struct Data data, *pdata;
83       data.x = ((float)analogRead(0)/1024)*64-4;
84       data.y = ((float)analogRead(1)/1024)*128+count;
85       data.id = 2;
86
87       pdata = &data;
88       // try send or block
89       while(xQueueSend(queue, (void*)&pdata, (TickType_t)0) == errQUEUE_FULL)
90         vTaskDelay(1);
91     }else count = 0;
92     vTaskDelay(1);  // One tick delay (15ms) in between reads for stability
93   }
94 }
95
96 void Update(void *pvParameters __attribute__((unused)))
97 {
98   /*
99     Update
100    Reads a digital input on pin 2 for fire the bullet
101    and reads analogical inputs A0 and A1 for control of smiles position
102  */
103  uint8_t button = 2;
104
105  // make the button's pin an input:
106  pinMode(button, INPUT_PULLUP);
107
108  for (;;)
109  {
110    if (!digitalRead(button)){
111      //bullet walk
112      xQueueSend(mutex, NULL, (TickType_t)0);
113    }
114    //smile :)
115    struct Data data, *pdata;
116    data.x = ((float)analogRead(0)/1024)*64-4;
117    data.y = ((float)analogRead(1)/1024)*128-4;
118    data.id = 1;
119
120    pdata = &data;
121    // try send or block
122    while(xQueueSend(queue, (void*)&pdata, (TickType_t)0) == errQUEUE_FULL)
123      vTaskDelay(1);
124    vTaskDelay(1);  // one tick delay (15ms) in between reads for stability
125  }
126 }
127
128 void Draw(void *pvParameters __attribute__((unused)))
129 {
```

```
130   /*
131     Draw
132     Reads the buffer of images description in queue and draw it in the display
133   */
134   // picture loop
135   u8g.firstPage();
136   do{
137     struct Data *data;
138     // try receive or block
139     while (xQueueReceive(queue, &data, (TickType_t)10) == pdFALSE) vTaskDelay(1);
140
141     if(data->id == 1)
142       u8g.drawBitmapP(data->y, data->x, 3, 20, smile);
143     else if(data->id == 2){
144       u8g.drawBitmapP(data->y, data->x, 3, 10, bullet);
145     }
146
147     vTaskDelay(1);
148   } while(u8g.nextPage());
149 }
```

2 Mecanismos de escalonamento

Nesta seção iremos analisar o comportamento do escalonador do FreeRTOS. Para melhor compreendê-lo, executou-se o código visto abaixo duas vezes, donde, na segunda vez, modificou-se o valor de prioridade de apenas uma das tasks para 2.

```
1 #include <Arduino_FreeRTOS.h>
2 void vTask1( void * pvParameters ){
3   for( ;; ){
4     Serial.println("*");
5   }
6 }
7 void vTask2( void * pvParameters ){
8   for( ;; ){
9     Serial.println("+");
10  }
11 }
12
13 void setup(){
14   Serial.begin(115200);
15   xTaskCreate( vTask1, "Periodic", 128, NULL, 1, NULL );
16   xTaskCreate( vTask2, "Periodic", 128, NULL, 1, NULL );
17 }
18
19 void loop(){
20 }
```

Com esses testes pode-se verificar que, quando as tasks tem a mesma prioridade, tendem a executar na mesma proporção, porém, do contrário, quando uma task tem mais prioridade que as demais, fica evidente o efeito de *starvation*, ou seja, a task com maior prioridade não para de ser executada e a task com menor prioridade nunca entra em execução.

Segundo a documentação oficial do FreeRTOS [1]: "The FreeRTOS scheduler ensures that tasks in the Ready or Running state will always be given processor

(CPU) time in preference to tasks of a lower priority that are also in the ready state. In other words, the task placed into the Running state is always the highest priority task that is able to run.", isto é, em tradução livre, a tarefa que entra em execução é sempre a tarefa com maior prioridade que esteja preparada para executar. Pode-se concluir então que o FreeRTOS implementa um escalonador simples baseado em *Escalonamento por Prioridade*.

3 Tarefas Periódicas

Comumente necessita-se garantir a periodicidade de certas tarefas em sistemas de tempo real, como a leitura de um determinado sensor que desempenha o controle crítico de uma válvula de pressão. Para que se tenha essa garantia, é necessário um estudo sobre as diferentes formas de controle temporal no FreeRTOS.

Pode-se perceber que, nos exemplos anteriores, nós só havíamos utilizado o método vTaskDelay(), o qual tem o objetivo de manter uma tarefa em modo de espera até que uma certa quantidade de batidas do relógio do processador sejam efetuadas. Pode-se calcular o tempo que a tarefa ficará aguardando utilizando, por exemplo, a constante portTICK_PERIOD_MS que retorna o período de cada batida do relógio.

Infelizmente essa não é uma boa solução quando se quer garantir um período constante para a execução de uma tarefa. Perceba que, no fim deste método, apenas é habilitado que a task volte a ser executada. Nada garante que ela realmente entrará em execução naquele momento. Pior ainda, na verdade, garante-se que ela sempre demorará a mais o tempo que o kernel necessita para coloca-la em execução. Note que uma grande quantidade de repetições gera um erro acumulado significativo.

Uma solução alternativa para este problema é a utilização da função vTaskDelayUntil(), que, diferentemente da sua prima já comentada, esta retorna a ser executada em um tempo exato passado como parâmetro. Na verdade, ela recebe dois parâmetros: pxPreviousWakeTime, que diz quando foi a ultima vez que ela foi chamada; e xTimeIncrement, que é daqui a quantos ciclos ela será executada. Ou seja, com esses dois parâmetros pode-se garantir que a tarefa volte a ser executada exatamente em pxPreviousWakeTime+xTimeIncrement.

Segundo a documentação oficial [1]: "Whereas vTaskDelay() specifies a wake time relative to the time at which the function is called, vTaskDelayUntil() specifies the absolute (exact) time at which it wishes to unblock.", isto é, em tradução livre, diferentemente da vTaskDelay(), vTaskDelayUntil() especifica o tempo exato em que se quer executar.

Segue exemplo de implementação de uma tarefa periódica no FreeRTOS.

```
1 #include <Arduino_FreeRTOS.h>
2
3 void vTask( void * pvParameters ){
4   for( ;; ){
5     vTaskDelay(1);
6   }
7 }
8
9 // Perform an action every 200ms.
10 void vTaskFunction( void * pvParameters )
11 {
```

```
12    TickType_t xLastWakeTime;
13    const TickType_t xFrequency = 200 / portTICK_PERIOD_MS;
14
15    // Initialise the xLastWakeTime variable with the current time.
16    xLastWakeTime = xTaskGetTickCount();
17    for( ;; )
18    {
19      // Wait for the next cycle.
20      vTaskDelayUntil( &xLastWakeTime, xFrequency );
21      Serial.println("Tick");
22    }
23  }
24
25  void setup(){
26    Serial.begin(2000000);
27    xTaskCreate( vTaskFunction, "Periodic", 128, NULL, 1, NULL );
28    xTaskCreate( vTask, "*", 128, NULL, 2, NULL );
29  }
30
31  void loop(){
32  }
```

Referências

[1] Richard Barry. *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Limited, 2009.