PRACTICA 4 : SISTEMAS OPERATIVOS EN TIEMPO REAL

El objetivo de la practica es comprender el funcionamiento de un sistema operativo en tiempo Real .

Para lo cual realizaremos una practica donde generaremos varias tareas y veremos como se ejecutan dividiendo el tiempo de uso de la cpu.

Introducción teórica

Multitarea en ESP32 con Arduino y FreeRTOS

A estas alturas, no es ningún secreto que el ESP32 es mi chip de referencia para fabricar dispositivos IoT. Son pequeños, potentes, tienen un montón de funciones integradas y son relativamente fáciles de programar.

Sin embargo, cuando se usa junto con Arduino, todo su código se ejecuta en un solo núcleo. Eso parece un poco desperdicio, así que cambiemos eso usando FreeRTOS para programar tareas en ambos núcleos.

¿Por qué?

Hay varios casos de uso para querer realizar múltiples tareas en un microcontrolador. Por ejemplo: puede tener un microcontrolador que lee un sensor de temperatura, lo muestra en una pantalla LCD y lo envía a la nube.

Puede hacer los tres sincrónicamente, uno tras otro. Pero, ¿qué sucede si está utilizando una pantalla de tinta electrónica que tarda unos segundos en actualizarse? **Responder en el informe**

Afortunadamente, la implementación de Arduino para ESP32 incluye la posibilidad de programar tareas con FreeRTOS. Estos pueden ejecutarse en un solo núcleo, muchos núcleos e incluso puede definir cuál es más importante y debe recibir un trato preferencial.

Creando tareas

Para programar una tarea, debe hacer dos cosas: crear una función que contenga el código que desea ejecutar y luego crear una tarea que llame a esta función.

Digamos que quiero hacer parpadear un LED encendido y apagado continuamente.

Primero, definiré el pin al que está conectado el LED y estableceré su modo OUTPUT. Cosas muy estándar de Arduino:

```
const int led1 = 2; // Pin of the LED

void setup(){
  pinMode(led1, OUTPUT);
}
```

A continuación, crearé una función que se convertirá en la base de la tarea. Utilizo digitalWrite()para encender y apagar el LED y usar **vTaskDelay** (en lugar de **delay()**) para pausar la tarea 500ms entre estados cambiantes:

```
void toggleLED(void * parameter){
  for(;;){ // infinite loop

    // Turn the LED on
    digitalWrite(led1, HIGH);

    // Pause the task for 500ms
    vTaskDelay(500 / portTICK_PERIOD_MS);

    // Turn the LED off
    digitalWrite(led1, LOW);

    // Pause the task again for 500ms
    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}
```

¡Esa es tu primera tarea! Un par de cosas a tener en cuenta:

Sí, creamos un for(;;)bucle infinito , y eso puede parecer un poco extraño. ¿Cómo podemos realizar múltiples tareas si escribimos una tarea que continúa para siempre? El truco es **vTaskDelay** que le dice al programador que esta tarea no debe ejecutarse durante un período determinado. El planificador pausará el ciclo for y ejecutará otras tareas (si las hay).

Por último, pero no menos importante, tenemos que informarle al planificador sobre nuestra tarea. Podemos hacer esto en la setup() función:

```
void setup() {
  xTaskCreate(
    toggleLED,    // Function that should be called
    "Toggle LED",   // Name of the task (for debugging)
    1000,            // Stack size (bytes)
    NULL,            // Parameter to pass
    1,               // Task priority
    NULL             // Task handle
  );
}
```

¡Eso es! ¿Quiere hacer parpadear otro LED en un intervalo diferente? Simplemente cree otra tarea y siéntese mientras el programador se encarga de ejecutar ambas.

Crear una tarea única También puede crear tareas que solo se ejecuten una vez. Por ejemplo, mi monitor de energía crea una tarea para cargar datos en la nube cuando tiene suficientes lecturas.

Las tareas únicas no necesitan un ciclo for interminable, sino que se ve así:

```
void uploadToAWS(void * parameter){
    // Implement your custom logic here

    // When you're done, call vTaskDelete. Don't forget this!
    vTaskDelete(NULL);
}
```

Esto parece una función normal de C ++ excepto por vTaskDelete(). Después de llamarlo, FreeRTOS sabe que la tarea ha finalizado y no debe reprogramarse. (Nota: no olvide llamar a esta función o hará que el perro guardián reinicie el ESP32).

```
xTaskCreate(
    uploadToAWS,    // Function that should be called
    "Upload to AWS",  // Name of the task (for debugging)
    1000,            // Stack size (bytes)
    NULL,            // Parameter to pass
    1,               // Task priority
    NULL             // Task handle
);
```

###Elija en qué núcleo se ejecutará### Cuando lo usa **xTaskCreate()**, el programador es libre de elegir en qué núcleo ejecuta su tarea. En mi opinión, esta es la solución más flexible (nunca se sabe cuándo puede aparecer un chip IoT de cuatro núcleos, ¿verdad?)

Sin embargo, es posible anclar una tarea a un núcleo específico con **xTaskCreatePinnedToCore**. Es como **xTaskCreatey** toma un parámetro adicional, el núcleo en el que desea ejecutar la tarea:

```
xTaskCreatePinnedToCore(
    uploadToAWS,      // Function that should be called
    "Upload to AWS",    // Name of the task (for debugging)
    1000,               // Stack size (bytes)
    NULL,               // Parameter to pass
    1,                  // Task priority
    NULL,               // Task handle
    0,          // Core you want to run the task on (0 or 1)
);
```

###Compruebe en qué núcleo se está ejecutando La mayoría de las placas ESP32 tienen procesadores de doble núcleo, entonces, ¿cómo sabe en qué núcleo se está ejecutando su

tarea?

Simplemente llame xPortGetCoreID()desde dentro de su tarea:

```
void exampleTask(void * parameter){
  Serial.print("Task is running on: ");
  Serial.println(xPortGetCoreID());
  vTaskDelay(100 / portTICK_PERIOD_MS);
}
```

Cuando tenga suficientes tareas, el programador comenzará a enviarlas a ambos núcleos.

Detener tareas

Ahora, ¿qué pasa si agregó una tarea al programador, pero desea detenerla? Dos opciones: borra la tarea desde dentro o usa un identificador de tareas. Terminar una tarea desde adentro ya se discutió antes (uso vTaskDelete).

Para detener una tarea desde otro lugar (como otra tarea o su bucle principal), tenemos que almacenar un identificador de tarea:

```
// This TaskHandle will allow
TaskHandle_t task1Handle = NULL;

void task1(void * parameter){
   // your task logic
}

xTaskCreate(
    task1,
    "Task 1",
    1000,
    NULL,
    1,
    task1Handle            // Task handle
);
```

Todo lo que tuvimos que hacer fue definir el identificador y pasarlo como último parámetro de xTaskCreate. Ahora podemos matarlo con vTaskDelete:

```
void anotherTask(void * parameter){
  // Kill task1 if it's running
  if(task1Handle != NULL) {
    vTaskDelete(task1Handle);
  }
}
```

Prioridad de la tarea

A la hora de crear tareas, tenemos que darle prioridad. Es el quinto parámetro de xTaskCreate. Las prioridades son importantes cuando dos o más tareas compiten por el tiempo de la CPU. Cuando eso suceda, el programador ejecutará primero la tarea de mayor prioridad. ¡Tiene sentido!

En FreeRTOS, un número de prioridad más alto significa que una tarea es más importante . Encontré esto un poco contra-intuitivo porque para mí una "prioridad 1" suena más importante que una "prioridad 2", pero ese soy solo yo.

Cuando dos tareas comparten la misma prioridad, FreeRTOS compartirá el tiempo de procesamiento disponible entre ellas.

Cada tarea puede tener una prioridad entre 0 y 24. El límite superior está definido por configMAX_PRIORITIESen el archivo FreeRTOSConfig.h .

Utilizo esto para diferenciar las tareas primarias de las secundarias. Tome el medidor de energía de mi casa: la tarea de mayor prioridad es medir la electricidad (prioridad 3). Actualizar la pantalla o sincronizar la hora con un servidor NTP no es tan crítico para su funcionalidad principal (prioridad 2).

Ejercicio Practico 1

Programar el siguiente codigo

```
void setup()
{
Serial.begin(112500);
/* we create a new task here */
xTaskCreate(
anotherTask, /* Task function. */
"another Task", /* name of task. */
10000, /* Stack size of task */
NULL, /* parameter of the task */
1, /* priority of the task */
NULL); /* Task handle to keep track of created task */
}

/* the forever loop() function is invoked by Arduino ESP32 loopTask */
void loop()
{
Serial.println("this is ESP32 Task");
delay(1000);
}

/* this function will be invoked when additionalTask was created */
void anotherTask( void * parameter )
{
/* loop forever */
for(;;)
{
Serial.println("this is another Task");
delay(1000);
}
```

```
/* delete a task when finish,
this will never happen because this is infinity loop */
vTaskDelete( NULL );
}
```

1. Descibir la salida por el puerto serie
2. Expkicar el funcionamiento

Ejercicio Practico 2

A realizar como ejercicio en casa

1. Realizar un programa que utilice dos tareas una enciende un led y otra lo apaga dichas tareas deben estar sincronizadas

sugerencias utilizar un semaforo

https://circuitdigest.com/microcontroller-projects/arduino-freertos-tutorial-using-semaphore-and-mutex-in-freertos-with-arduino

https://techtutorialsx.com/2017/05/11/esp32-freertos-counting-semaphores/

referencias

https://github.com/uagaviria/ESP32_FreeRtos

https://www.freertos.org/