Unidad Profesional Interdisciplinaria de Ingeniería
Campus Zacatecas IPN

Práctica 4 y 5. Exclusión Mutua

Rafael Martín Enciso Cuevas
Fernando Murillo Franco
Roberto Coronel Santiago

M. en C. Ramón Jaramillo Martínez

Optativa 4
Sistemas Operativos en Tiempo Real

Zacatecas, Zac.
30 de octubre de 2023

RESUMEN

Los sistemas operativos en tiempo real tienen aplicaciones extensas, desde sistemas de control industrial, dispositivos médicos, de seguridad, hasta vehículos autónomos. La garantía de que múltiples tareas o procesos puedan acceder a recursos compartidos de manera segura y sin conflictos es de vital importancia para garantizar su funcionamiento correcto y cumplir con estrictas restricciones temporales.

Uno de los conceptos clave en la administración de recursos compartidos es la exclusión mutua. Se refiere a la capacidad de asegurar que solo un proceso tenga acceso a una sección crítica de código en un momento dado. Es esta práctica, exploraremos diferentes algoritmos y métodos de exclusión mutua y su implementación en un entorno de sistemas operativos en tiempo real.

ÍNDICE DE CONTENIDO

1. INTRODUCCIÓN

En esta práctica abordamos un concepto fundamental de la programación concurrente. Esto es la exclusión mutua. Este concepto es esencial para evitar problemas de concurrencia, tales como condiciones de carrera y bloqueos inesperados, que pueden surgir cuando múltiples procesos intentan acceder a los mismos recursos compartidos simultáneamente.

En la presente, se explora la importancia de la exclusión mutua y se realizan ejercicios prácticos para comprender su aplicación en sistemas embebidos y sistemas en tiempo real. Para ello, se utilizan tarjetas de desarrollo ESP32 y el entorno de Arduino IDE, junto con sensores analógicos.

El objetivo principal de esta práctica es adquirir una comprensión de la exclusión mutua y los algoritmos que la implementan, así como su aplicación en situaciones prácticas. Se exploran diferentes enfoques y se analizan ventajas y desventajas para tomar decisiones informadas sobre cuál es el método más adecuado en un contexto particular.

A lo largo de este trabajo, se implementa un código básico de exclusión mutua, se exploran los algoritmos de Dekker y Peterson, y se analizan las diferencias entre ellos, identificando sus ventajas y desventajas. Además, se lleva a cabo una aplicación práctica que involucra la escritura en una base de datos. La práctica proporciona una sólida base teórica y práctica para comprender y aplicar la exclusión mutua en sistemas embebidos y en tiempo real.

1.1. ANTECEDENTES

Exclusión mutua: La administración de tareas de datos y recursos compartidos en un sistema involucra la implementación de la exclusión mutua para evitar conflictos entre procesos que compiten por acceder a la misma región de memoria o recurso. En sistemas de multiprogramación con un único procesador, los procesos se intercalan en el tiempo para simular, una ejecución simultánea. Sin embargo, el problema de compartir recursos conlleva riesgos, como conflictos en el valor de

variables globales cuando múltiples procesos realizan operaciones de lectura y escritura simultáneamente (Monroy, Sistemas operativos, 2014).

Secciones críticas de código: Son aquellas partes de los procesos concurrentes que no pueden ejecutarse de forma concurrente o, también, que desde otro proceso se ven como si fueran una única instrucción. Esto quiere decir que, si un proceso entra a ejecutar una sección crítica en la que se accede a unas variables compartidas, entonces otro proceso no puede entrar a ejecutar una región crítica en la que acceda a variables compartidas con el anterior (Programación Concurrente).

Características mínimas necesarias para ser considerado un proceso de exclusión: Seguridad: A lo sumo un proceso puede estar ejecutándose una vez en la SC. Supervivencia: Las peticiones para entrar y salir de la SC al final deben ser concedidas, esto implica la inexistencia de deadlocks e inanición. Ordenación: Si una petición para entrar en la SC ocurrió antes que otra, entonces la entrada a la SC se garantiza en ese orden (Wikipedia, 2018).

Algoritmos de exclusión mutua. (Dekker, Peaterson y otros):

Dekker: El algoritmo de Dekker es un algoritmo de programación concurrente para exclusión mutua, que permite a dos procesos o hilos de ejecución compartir un recurso sin conflictos.

FUNCIONAMIENTO

Si ambos procesos intentan acceder a la sección crítica simultáneamente, el algoritmo elige un proceso según una variable de turno. Si el otro proceso está ejecutando en su sección crítica, deberá esperar su finalización.

PRIMERA VERSIÓN

Garantiza la exclusión mutua, pero su desventaja es que acopla los procesos fuertemente, esto significa que los procesos lentos atrasan a los procesos rápidos.

SEGUNDA VERSIÓN

Problema interbloqueo. No existe la alternancia, aunque ambos procesos caen a un mismo estado y nunca salen de ahí.

TERCERA VERSIÓN

Colisión región crítica no garantiza la exclusión mutua. Este algoritmo no evita que dos procesos puedan acceder al mismo tiempo a la región crítica.

CUARTA VERSIÓN

Postergación indefinida. Aunque los procesos no están en interbloqueo, un proceso o varios se quedan esperando a que suceda un evento que tal vez nunca suceda (Sandoval, 2019).

Peaterson: El algoritmo de Peterson, también conocido como solución de Peterson, es un algoritmo de programación

concurrente para exclusión mutua, que permite a dos o más procesos o hilos de ejecución compartir un recurso

sin conflictos, utilizando sólo memoria compartida para la comunicación.

Peterson desarrolló en 1981 el algoritmo básico para dos procesos, como una simplificación del algoritmo de Dekker. El algoritmo básico puede generalizarse fácilmente a un número arbitrario de n procesos

¿En qué consiste?

-Cuando dos o más procesos secuenciales en cooperación ejecutan todos ellos asíncronamente y comparten datos comunes se produce el problema de la sección crítica.

-Cada proceso tiene un turno para entrar a la sección critica

-Si un proceso quiere entrar, debe activar su señal y puede que tenga que esperar a que llegue su turno (Glez, 2015).

ALGORITMO DE SEMÁFORO

Un semáforo es una variable especial (o tipo abstracto de datos) que constituye el método clásico para restringir o permitir el acceso a recursos compartidos (por ejemplo, un recurso de almacenamiento del sistema o variables del código fuente) en un entorno de multiprocesamiento (en el que se ejecutarán varios procesos

concurrentemente).

Ejemplo de uso

Los semáforos pueden ser usados para diferentes propósitos, entre ellos:

• Implementar cierres de exclusión mutua o locks

• Barreras

• Permitir a un máximo de N threads (hilos) acceder a un recurso, inicializando el semáforo en N

• Notificación. Inicializando el semáforo en 0 puede usarse para comunicación entre threads sobre la disponibilidad de un recurso (ramirez, 2015).

ALGORITMO DE MONITOR

En la programación paralela, los monitores son estructuras de datos abstractas destinadas a ser usadas sin peligro por más de un hilo de ejecución. La característica que principalmente los define es que sus métodos son ejecutados con exclusión mutua. Lo que significa, que, en cada momento en el tiempo, un hilo como máximo puede estar ejecutando cualquiera de sus métodos. Esta exclusión mutua simplifica el razonamiento de implementar monitores en lugar de código a ser ejecutado en paralelo.

Componentes

Un monitor tiene cuatro componentes: inicialización, datos privados, métodos del monitor y cola de entrada.

• Inicialización: contiene el código a ser ejecutado cuando el monitor es creado

• Datos privados: contiene los procedimientos privados, que solo pueden ser usados desde dentro del

monitor y no son visibles desde fuera

• Métodos del monitor: son los procedimientos que pueden ser llamados desde fuera del monitor.

• Cola de entrada: contiene a los hilos que han llamado a algún método del monitor, pero no han podido adquirir permiso para ejecutarlos aún (Wikipedi, 2021).

Diferencias en ambos algoritmos:

El algoritmo de Dekker es un algoritmo de programación concurrente para exclusión mutua, que permite a dos procesos o hilos de ejecución compartir un recurso sin conflictos. Fue uno de los primeros algoritmos de exclusión mutua inventados, implementado por Edsger Dijkstra.

Si ambos procesos intentan acceder a la sección crítica simultáneamente, el algoritmo elige un proceso según una variable turno. Si el otro proceso está ejecutando en su sección crítica, deberá esperar su finalización.

El algoritmo de Peterson es un algoritmo de programación concurrente para exclusión mutua, que permite a dos o más procesos o hilos de ejecución compartir un recurso sin conflictos, utilizando sólo memoria compartida para la comunicación.

Peterson desarrolló el primer algoritmo (1981) para dos procesos que fue una simplificación del algoritmo de Dekker para dos procesos. Posteriormente este algoritmo fue generalizado para N procesos (SITES, 2019).

Ventajas y desventajas de los procesos de exclusión:

Ventajas:

- Es aplicable a cualquier nº de procesos en sistemas con memoria compartida, tanto de monoprocesador como de multiprocesador.
- Es simple y fácil de verificar.
- Puede usarse para disponer de varias secciones críticas (Universidad de Valladolid).

Desventajas:

- Uno de los grandes problemas que nos podemos encontrar es que el hecho de compartir recursos está lleno de riesgos (Monroy, UAEH, s.f.).
- La espera activa consume tiempo del procesador.
- Puede producirse inanición cuando un proceso abandona la sección crítica y hay más de un proceso esperando.
- Interbloqueo: de procesos de baja prioridad frente a otros de prioridad mayor (Universidad de Valladolid).

1.4. OBJETIVO

Adquirir una comprensión de los conceptos fundamentales y aplicar técnicas esenciales requeridas para la implementación de algoritmos de exclusión mutua en un contexto de sistemas operativos en tiempo real.

2. DESARROLLO

2.1. CÓDIGO BÁSICO DE EXCLUSIÓN MUTUA

2.1.1. Código en el entorno de Arduino IDE

Especificaciones:

- Dos tareas que compartan una sección crítica. La sección crítica corresponde al envío de un mensaje por UART. Cada proceso además de mandar un mensaje en común mandará un carácter que identifique que proceso ha utilizado la sección crítica.
- Genere un código básico que realice el proceso de exclusión.
- Ambas tareas deben tener la misma prioridad.

Código en Arduino:

```
2  bool semaforo = false;
3  void setup() {
4    // Inicializar la comunicación UART
5    Serial.begin(9600);
6  }
7  void loop() {
8    tarea1();
9    tarea2();
10 }
11 void tarea1() {
12   // Intentar adquirir el semáforo
13   if (!semaforo) {
14     // Sección crítica
15     semaforo = true;
16
17     // Enviar mensaje y carácter de identificación
18     Serial.print("Tarea 1 - Sección crítica");
19     Serial.println(" (Proceso A)");
20
21     // Liberar el semáforo
22     semaforo = false;
23     // Realizar otras acciones aquí fuera de la sección crítica
24     delay(1000);
25   }
26 }
27 void tarea2() {
```

Ilustración 1: Código en Arduino, parte 1(elaboración propia).

```
27  void tarea2() {
28    // Intentar adquirir el semáforo
29    if (!semaforo) {
30      // Sección crítica
31      semaforo = true;
32      // Enviar mensaje y carácter de identificación
33      Serial.print("Tarea 2 - Sección crítica");
34      Serial.println(" (Proceso B)");
35      // Liberar el semáforo
36      semaforo = false;
37      // Realizar otras acciones aquí fuera de la sección crítica
38      delay(1000);
39    }
40  }
```

Ilustración 2: Código en Arduino, parte 2(elaboración propia).

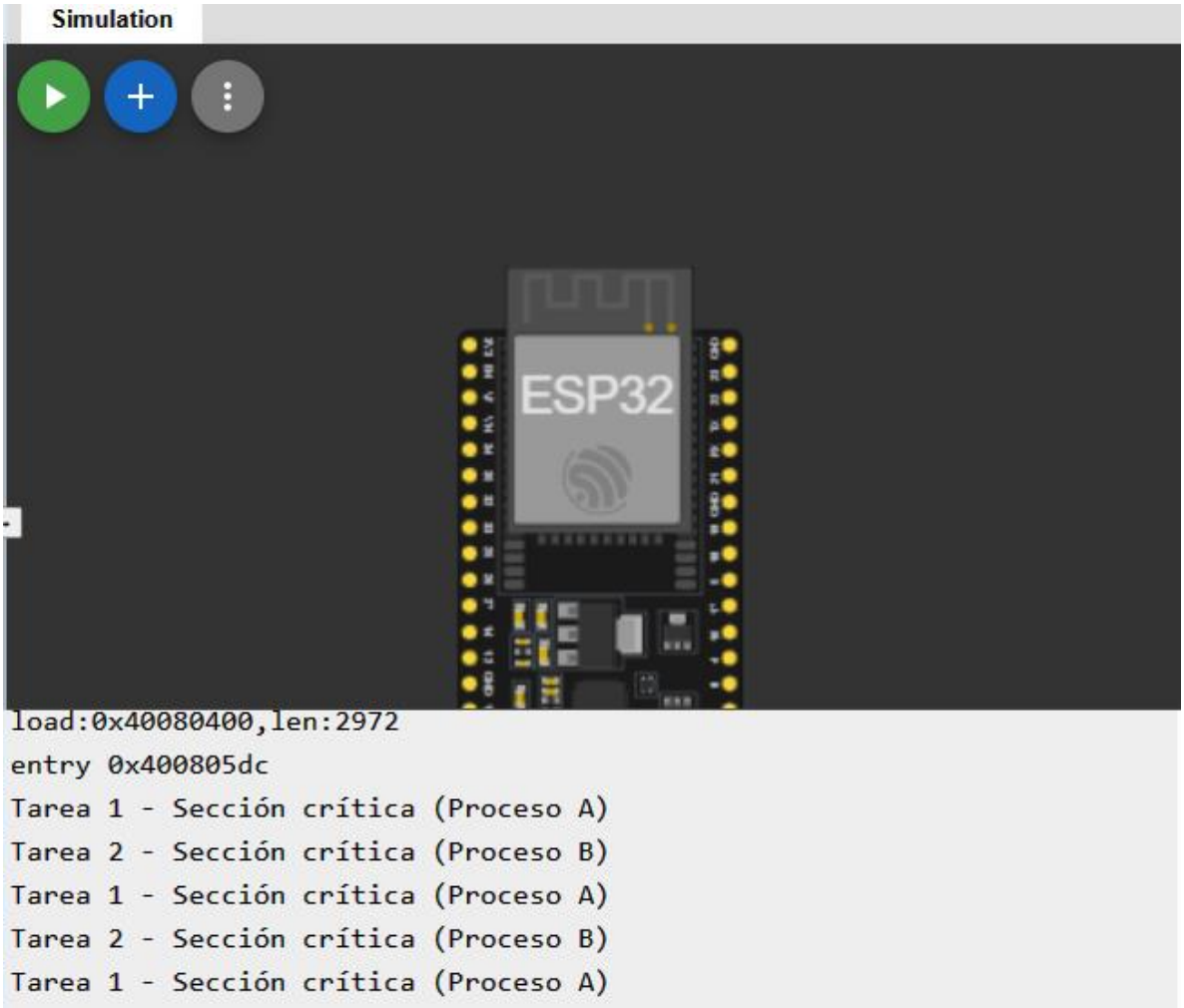

Ilustración 3: simulación en la plataforma de Wokwi (Elaboración propia).

2.1.2. Cuatro tareas compartes misma sección crítica

Especificaciones:

- Modifique el código anterior para utilizar 4 tareas que compartan la misma sección crítica.

Código modificado para cuatro tareas en Arduino:

```
// Inicialización de semáforo para exclusión mutua
bool semaforo = false;
void setup() {
  // Inicializar la comunicación UART
  Serial.begin(9600);
}
void loop() {
  tarea1();
  tarea2();
  tarea3();
  tarea4();
}
void tarea1() {
  // Intentar adquirir el semáforo
  if (!semaforo) {
    // Sección crítica
    semaforo = true;
    // Enviar mensaje y carácter de identificación
    Serial.print("Tarea 1 - Sección crítica");
    Serial.println(" (Proceso A)");
    // Liberar el semáforo
    semaforo = false;
    // Realizar otras acciones aquí fuera de la sección crítica
    delay(1000);
  }
}
```

Ilustración 4: Código en Arduino, parte 1(elaboración propia).

```
28  void tarea2() {
29    if (!semaforo) {
30      semaforo = true;
31      Serial.print("Tarea 2 - Sección crítica");
32      Serial.println(" (Proceso B)");
33      semaforo = false;
34      delay(1000);
35    }
36  }
37  void tarea3() {
38    if (!semaforo) {
39      semaforo = true;
40      Serial.print("Tarea 3 - Sección crítica");
41      Serial.println(" (Proceso C)");
42      semaforo = false;
43      delay(1000);
44    }
45  }
46  void tarea4() {
47    if (!semaforo) {
48      semaforo = true;
49      Serial.print("Tarea 4 - Sección crítica");
50      Serial.println(" (Proceso D)");
51      semaforo = false;
52      delay(1000);
53    }
54  }
```

Ilustración 5: Código en Arduino, parte 2
Elaboración propia

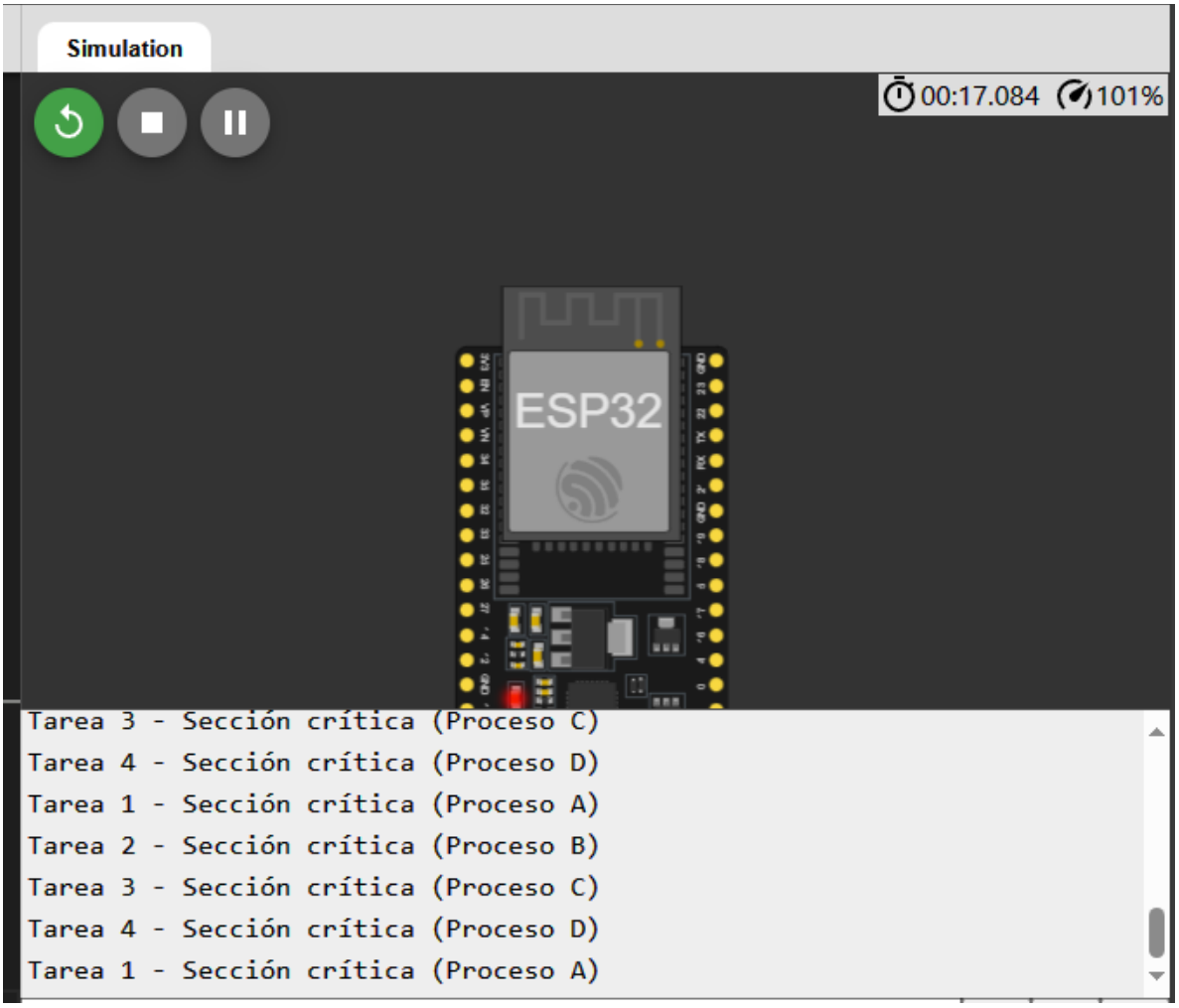

Ilustración 6: simulación en la plataforma de Wokwi
Elaboración propia

2.1.3. Análisis

- ¿Es posible crear un ciclo infinito?

Es posible que ocurra un ciclo infinito si no se utilizan mecanismos de exclusión mutua para controlar el acceso a la sección crítica. La falta de sincronización adecuada entre ambos procesos podría generar conflictos y condiciones impredecibles, lo que podría dar lugar a un ciclo infinito.

- ¿Qué ventajas y desventajas identifica?

Las ventajas de permitir que dos tareas compartan una sección crítica incluyen la simplificación del diseño del sistema al facilitar la comunicación y el intercambio de recursos entre las tareas. Además, esta práctica puede conducir a una mayor eficiencia al optimizar el uso de recursos, lo cual es especialmente beneficioso en entornos con restricciones de recursos.

Las desventajas incluyen el riesgo de condiciones de carrera si no se implementan adecuadamente mecanismos de exclusión mutua, lo que puede resultar en resultados impredecibles o incorrectos. La implementación de la exclusión mutua y la sincronización entre tareas puede aumentar la complejidad del código, lo que dificulta su mantenimiento y depuración.

2.2. ALGORITMO DE DEKKER Y PEATERSON

2.2.1. Código de Dekker

Especificaciones:

- Genere el código de Dekker en cualquiera de sus variantes y explique el funcionamiento (Utilice FreeRTOS).

```
// ALGORITMO DE DEKKER
// Sistemas Operativos en Tiempo Real
// Equipo de la muerte
// Octubre 2023

SemaphoreHandle_t mutex;

void Task1(void *pvParameters) {
  int id = 0;
  while (1) {
    xSemaphoreTake(mutex, portMAX_DELAY);
    // para la primer parte
    Serial.println("Actividad 1 está adentro paps");
    // sale de la exclusión
    xSemaphoreGive(mutex);
    // Dejar que haga las otras tareas
    vTaskDelay(pdMS_TO_TICKS(1000));
  }
}

void Task2(void *pvParameters) {
  int id = 1;
  while (1) {
    xSemaphoreTake(mutex, portMAX_DELAY);
    // segunda parte
    Serial.println("Actividad 2 está adentro rey");
    // sale de la exclusión
    xSemaphoreGive(mutex);
    // Dejar que haga otras tareas
    vTaskDelay(pdMS_TO_TICKS(1000));
  }
}

void setup() {
  Serial.begin(9600);
  // Un semaforo para exclusión mutua
  mutex = xSemaphoreCreateMutex();

  xTaskCreate(Task1, "Task1", 1000, NULL, 1, NULL);
  xTaskCreate(Task2, "Task2", 1000, NULL, 1, NULL);

  vTaskStartScheduler();
}

void loop() {
  // loop dejar vacio por recomendacion de profe
}
```

Ilustración 7. Algoritmo de Dekker
Elboración Propia

El código es una variante del Algoritmo de Dekker que utiliza variables globales. Usamos dos variables globales “want” y “turn” para lograr lo exclusión mutua entre dos procesos o huilos.

Want es un arreglo de dos elementos que indica si un proceso quiere acceder a la región crítica. Turn es una variable que indica a quien se le permite el acceso en ese momento.

La función “Algoritmo” se llama desde el loop para simular que se ejecutaría en cada uno de los dos procesos. Esta función toma un argumento “id” que identifica el proceso con un binario 0 o 1.

Antes de entra en la región crítica, un proceso establece que quiere entrar.

Tenemos un ciclo de espera para verificar si el otro proceso también quiere acceder con “other” o con “turn”. En caso de que los dos procesos quieran acceder al mismo tiempo, uno de ellos tiene que esperar.

Cuando el proceso ha termino de ejecutar su código crítico, establece falso en el binario identificador, lo que dice que ya no quiere acceder y deja permiso para que otro proceso tenga la oportunidad de acceder a la región crítica.

A continuación, vemos la implementación de este código cargado en una tarjeta de desarrollo electrónica tipo ESP32

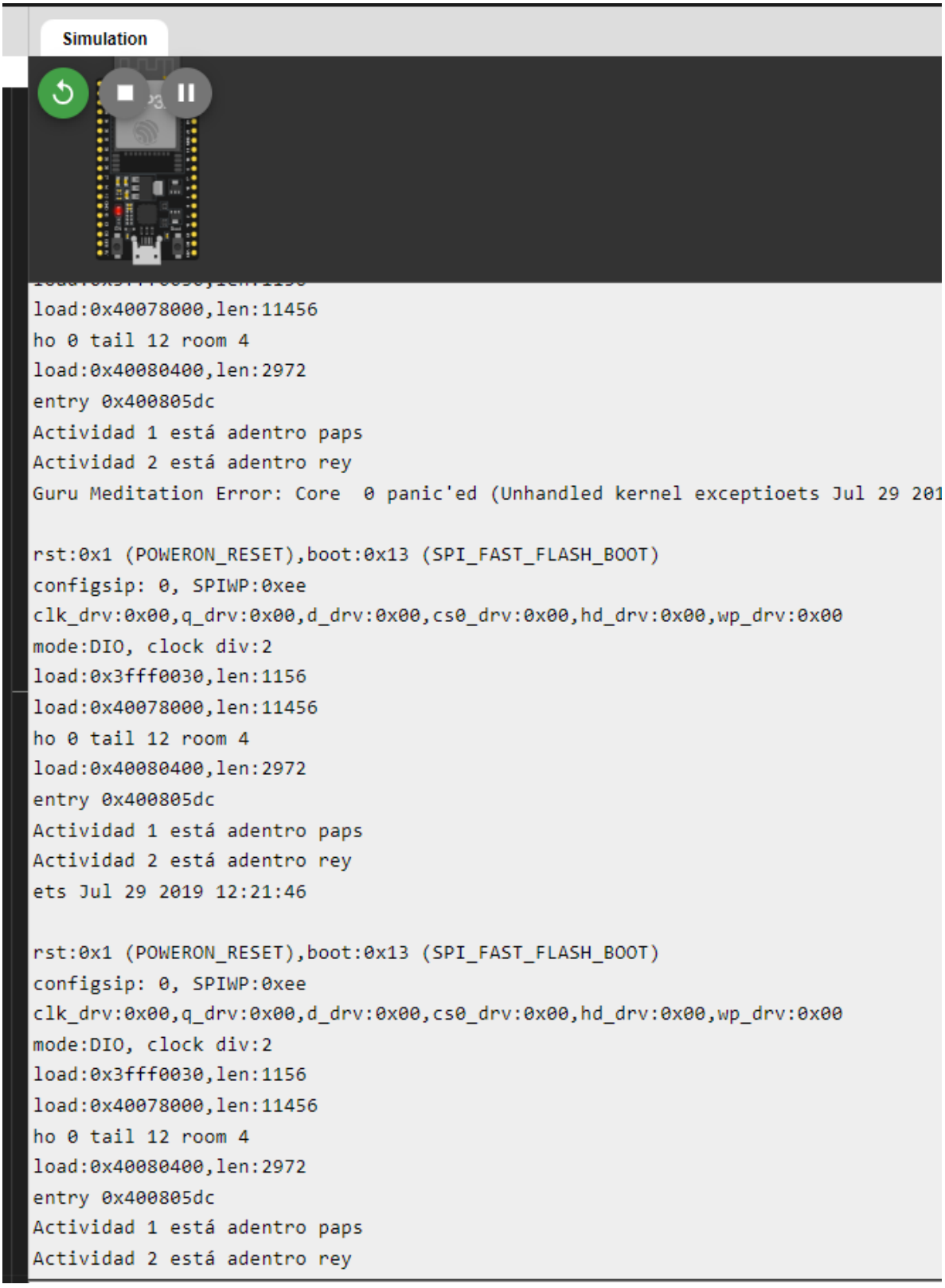

Ilustración 8. Simulación algoritmo Dekker
Elaboración Propia

2.2.2. Código de Peaterson

Especificaciones:

- Genere el código de Peaterson y explique el funcionamiento (Utilice FreeRTOS).

```
// código de Peaterson
#define N 2

volatile int turno = 0;
volatile bool want[N] = {false, false};

void tarea1(void *parameter) {
  while (1) {
    want[0] = true;
    turno = 1;
    while (want[1] && turno == 1) {
      // Esperar a que sea su turno
    }

    // Sección crítica
    Serial.println("Tarea 1 en la sección crítica");

    want[0] = false;
    // Sección no crítica
    vTaskDelay(1);
  }
}

void tarea2(void *parameter) {
  while (1) {
    want[1] = true;
    turno = 0;
    while (want[0] && turno == 0) {
      // Esperar a que sea su turno
    }
```

Ilustración 9: Algoritmo de Peaterson parte 1.

Elaboración propia

```
32
33      // Sección crítica
34      Serial.println("Tarea 2 en la sección crítica");
35
36      want[1] = false;
37      // Sección no crítica
38      vTaskDelay(1);
39    }
40  }
41
42  void setup() {
43    Serial.begin(115200);
44
45    xTaskCreatePinnedToCore(
46        tarea1, "Tarea1", 10000, NULL, 1, NULL, 0);  // Core 0
47    xTaskCreatePinnedToCore(
48        tarea2, "Tarea2", 10000, NULL, 1, NULL, 1);  // Core 1
49  }
50
51  void loop() {
52  }
53
```

Ilustración 10: Algoritmo de Peaterson parte 2.

Elaboración propia

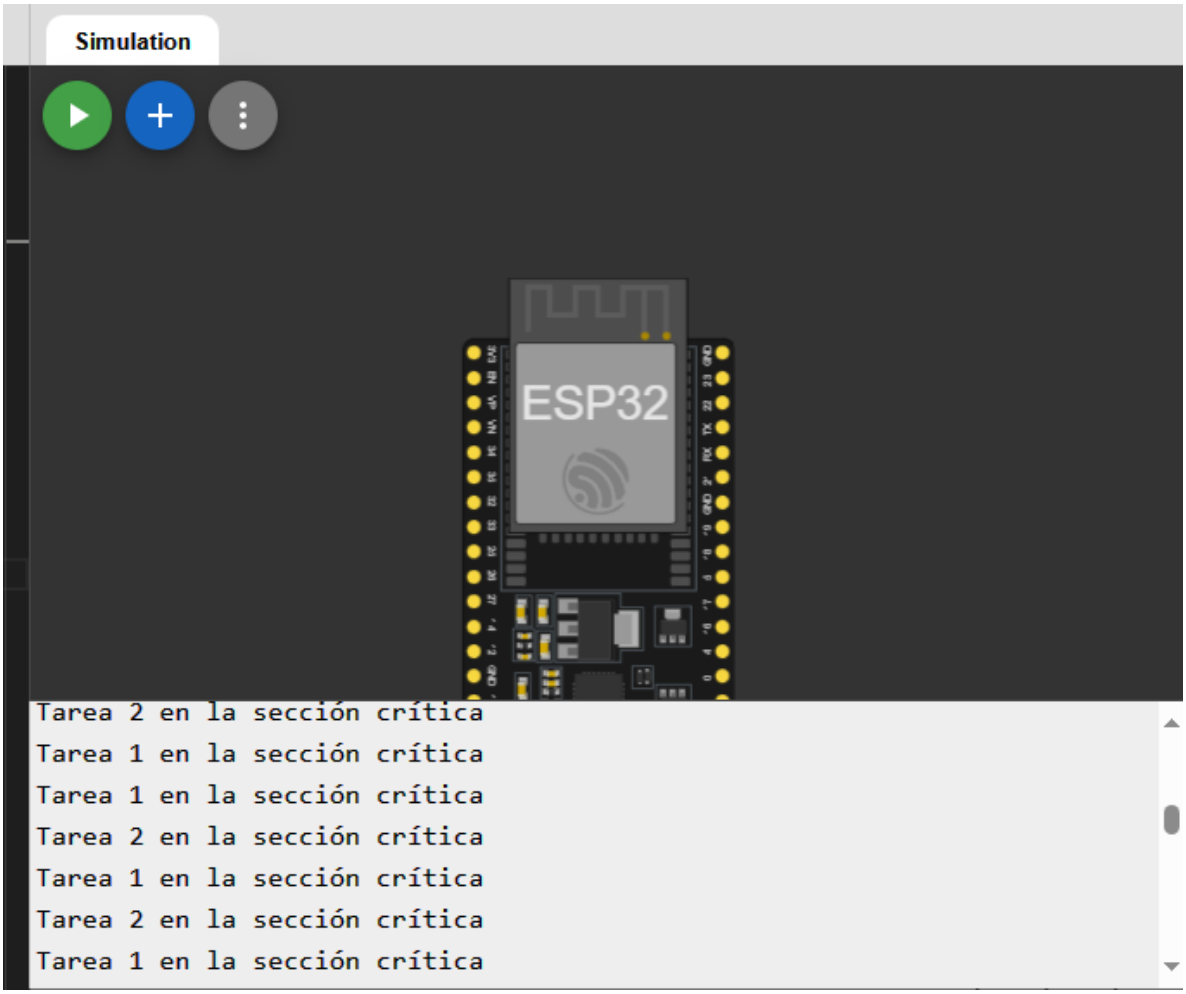

Ilustración 11: Simulación algoritmo Peaterson

Elaboración propia

En este código, las dos tareas tarea1 y tarea2 comparten una sección crítica utilizando el algoritmo de Peterson. El uso de las variables **turno** y **want** garantiza que los procesos no interfieran entre sí durante la ejecución de la sección crítica. Si un proceso está en la sección crítica, el otro proceso debe esperar su turno. Esto garantiza la exclusión mutua entre las dos tareas.

2.2.2. Análisis

1. ¿Qué ventajas y desventajas identifica?

Algoritmo de **Dekker**:

Ventajas:

- Implementación relativamente sencilla.
- No requiere hardware adicional y se puede implementar en software.
- Es adecuado para sistemas con dos procesos que necesitan compartir recursos críticos.

Desventajas:

- Puede experimentar inanición si uno de los procesos siempre se encuentra en la sección crítica.
- Puede producir resultados impredecibles si no se implementa correctamente.
- No es adecuado para sistemas con más de dos procesos que necesitan acceso a la sección crítica.

Algoritmo de **Peaterson**:

Ventajas:

- Implementación sencilla y comprensible.
- No requiere hardware adicional y es fácil de implementar en sistemas multiprocesador.
- Es adecuado para sistemas con dos procesos que necesitan compartir recursos críticos.

Desventajas:

- Puede sufrir inanición si uno de los procesos siempre se encuentra en la sección crítica.

- La eficiencia puede disminuir en sistemas con un gran número de procesos que necesitan acceso a la sección crítica.
- La complejidad aumenta en sistemas con más de dos procesos que necesitan acceso a recursos críticos.

2. ¿Qué diferencias observa entre ambos códigos y analice en qué condiciones es de utilidad o no?

Algoritmo de **Dekker**:

- Usa un mecanismo de cambio de turno explícito para decidir cuál de los dos procesos puede ingresar a la sección crítica en un momento dado.
- Utiliza la variable turn para indicar cuál de los procesos tiene prioridad para acceder a la sección crítica.
- Es adecuado para sistemas con dos procesos que necesitan acceso exclusivo a un recurso compartido. No es escalable para más de dos procesos.

Algoritmo de **Peterson**:

- Usa la variable turn al igual que Dekker, pero también utiliza un conjunto de variables booleanas interested para indicar el deseo de un proceso de acceder a la sección crítica.
- Proporciona una mayor claridad en la sincronización y la lógica de exclusión mutua al utilizar las variables interested.
- Al igual que Dekker, es adecuado para sistemas con dos procesos que necesitan acceso exclusivo a un recurso compartido. No se escala fácilmente para más de dos procesos.

Condiciones de utilidad:

- Ambos algoritmos son útiles en sistemas con un número limitado de procesos que compiten por el acceso a una sección crítica. Esto se aplica a situaciones donde se necesita la exclusión mutua, por ejemplo, para

garantizar que solo un proceso escriba o lea en un recurso compartido a la vez.

Condiciones en las que pueden no ser útiles:

- En sistemas con un gran número de procesos que necesitan acceso a una sección crítica, la implementación de Dekker o Peterson puede volverse complicada y menos eficiente. En estos casos, podrían preferirse soluciones más escalables, como semáforos o mutexes proporcionados por sistemas operativos en tiempo real como FreeRTOS.
- Si los procesos no cooperan adecuadamente o si se produce un error en la implementación, tanto Dekker como Peterson pueden sufrir condiciones de carrera y ciclos infinitos. Por lo tanto, se requiere una implementación cuidadosa y pruebas exhaustivas para garantizar su correcto funcionamiento.

3. ¿Es posible caer en ciclos infinitos?

Sí, tanto el algoritmo de Dekker como el algoritmo de Peterson pueden potencialmente sufrir de ciclos infinitos si no se implementan correctamente. Ambos algoritmos dependen de la sincronización y la cooperación entre los procesos para garantizar la exclusión mutua y evitar condiciones de carrera. Si se producen conflictos en la sincronización o si los procesos no cooperan adecuadamente, es posible que se creen situaciones en las que los procesos no puedan avanzar y se queden atrapados en bucles infinitos, sin permitir que otros procesos accedan a la sección crítica.

2.3. APLICACIÓN

2.2.1. Código en entorno Arduino IDE

Especificaciones: Genere un código en el entorno Arduino IDE que considere los siguientes:

- Tres tareas comparten una sección crítica.
- La sección critica consiste en escribir en una base de datos y validar que se ha escrito correctamente el dato en la base de datos. (Lectura/Escritura)
- Tarea 1. Estará adquiriendo la información del sensor 1. (También estará realizando el filtrado de la señal)
- Tarea 2. Estará adquiriendo la información del sensor 2. (También estará realizando el filtrado de la señal)
- Tarea 3. Estará recibiendo información flotante a través del puerto serial y esta información se enviará a la base de datos. (Solo se envía la información cuando se recibe información, de lo contrario no se solicita entrar a la sección crítica).
- Algoritmo de exclusión que usted considere se adapta mejor al problema y justifique la razón.

A continuación, se presenta el código desarrollado para la aplicación de este apartado.

```
/*****************************************************************************
*******/
/*LIBRERIAS*/
#ifdef ESP32
#include <WiFi.h>
#else
#include <ESP8266WiFi.h>
#endif
#include <WiFiClientSecure.h>
#include <UniversalTelegramBot.h>
#include <ArduinoJson.h>
#include <DHT.h>
/*****************************************************************************
*******/
/*CONFIGURACIÓN BOT-TELEGRAM*/
```

```
const char* ssid = "IZZI-8404"; //M_Note10
const char* password = "HPszmzJ4bNRZXCmTsC"; //$41801202

#define BOTtoken "6916809119:AAG-YS0a2Rjheqf0wvjzWLJPEvl65Y6cs8s"
#define CHAT_ID "1441341276" //1579653718 //1175557590
/*************************************************************************
*******/
/*SensorAnalógico_LM35*/
#define LM35_pin 0 // ESP32 pin GPIO36 (ADC0) conectado al sensor LM35
/*************************************************************************
*******/
/*SensorDigital_DHT11*/
#define DHTPIN 4  //PIN DONDE CONECTAR LA SEÑAL DEL SENSOR
#define DHTTYPE DHT11 //Definimos el tipo de DHT a utilizar
DHT dht(DHTPIN, DHTTYPE);//Declaramos nuestra variable DHT
/*************************************************************************
*******/
/*COMUNICACIÓN Y ACCESO A TELEGRAM*/
#ifdef ESP8266
  X509List cert(TELEGRAM_CERTIFICATE_ROOT);
#endif
/*************************************************************************
*******/
WiFiClientSecure client;
UniversalTelegramBot bot(BOTtoken, client);

int botRequestDelay = 500;
unsigned long lastTimeBotRan;
/*************************************************************************
*******/
/*CREAMOS LAS TAREAS*/
TaskHandle_t Task1;
TaskHandle_t Task2;
TaskHandle_t Task3;
static void Tarea1( void * parameter);
static void Tarea2( void * parameter);
static void Tarea3( void * parameter);
/*************************************************************************
*******/
/*DECLARAMOS BARIABLES PARA LOS TURNOS DE LAS TAREAS*/
volatile bool tarea[3] = {false,false,false};
int turno = 0;

bool valorI = LOW;
bool valorS = LOW;
```

```
/*************************************************************************
*******/
/*PINES DE LOS LED's*/
int LEDS = 18; //LM35  -- D12
int LEDI = 16; //DHT11 --
char mensaje;
int LEDU = 17;


//ASIGNAMOS LA TAREA PARA EL SENSOR  DHT11
static void Tarea1( void * parameter) {
  while(1){
    //SOLICITUD DE ENTRADA A LA SECC. CRITICA
    tarea[0] = true;
    turno = 0;
    while (tarea[1] || tarea[2] || (turno != 0)){
      //AQUI LAS TAREAS ESPERAN QUE SEA SU TURNO
      vTaskDelay(250);
    }
    //AQUI SE EJECUTA LA SECCIÓN CRITICA
    tarea[0]=false;
    vTaskDelay(1000 / portTICK_PERIOD_MS); //wait for a second
  }
}
//ASIGNAMOS LA TAREA DEL SENSOR LM35
static void Tarea2( void * parameter) {
  while(1){
    //SOLICITUD DE ENTRADA A LA SECC. CRITICA
    tarea[1] = true;
    turno = 1;
    while (tarea[0] || tarea[2] || (turno != 1)){
      //AQUI LAS TAREAS ESPERAN QUE SEA SU TURNO
      vTaskDelay(250);
    }
    //AQUI SE EJECUTA LA SECCIÓN CRITICA
    tarea[1]=false;
    vTaskDelay(1000 / portTICK_PERIOD_MS); //wait for a second
  }
}
static void Tarea3( void * parameter) {
  while(1){
    //SOLICITUD DE ENTRADA A LA SECC. CRITICA
    tarea[2] = true;
    turno = 2;
    while (tarea[0] || tarea[1] || (turno != 2)){
```

```
      //AQUI LAS TAREAS ESPERAN QUE SEA SU TURNO
      vTaskDelay(250);
    }
    //AQUI SE EJECUTA LA SECCIÓN CRITICA
    tarea[2]=false;
    vTaskDelay(1000 / portTICK_PERIOD_MS); //wait for a second
  }
}

//PROGRAMAMOS LA SECC. CRITICA
void handleNewMessages(int numNewMessages) {
  Serial.println("handleNewMessages");
  Serial.println(String(numNewMessages));

  for (int i=0; i<numNewMessages; i++) {

    String chat_id = String(bot.messages[i].chat_id);
    if (chat_id != CHAT_ID){
      bot.sendMessage(chat_id, "Unauthorized user", "");
      continue;
    }

    String text = bot.messages[i].text;
    Serial.println(text);

    String from_name = bot.messages[i].from_name;

    if (text == "/start") {
      String welcome = "Elija opción, " + from_name + ".\n";
      welcome += "Elige alguno de estos comandos para ejecutar una tarea del
ESP32.\n\n";
      welcome += "/led_DHT11 para ejecutar la tarea del sensor DHT11\n";
      welcome += "/led_LM35 para ejecutar la tarea del sensor LM35 \n";
      welcome += "/led_UART para ejecutar la tarea de la UART \n";
      bot.sendMessage(chat_id, welcome, "");
    }

    if (text == "/led_DHT11") {
      bot.sendMessage(chat_id, "Sección sensor DHT11", "");
    valorI = digitalRead(DHTPIN);
    digitalWrite(LEDI, valorI);
    delay(1500);
    digitalWrite(LEDI, LOW);
    }
```

```
    if (text == "/led_LM35") {
      Serial.print(LM35_pin);
      bot.sendMessage(chat_id, "Sección sensor LM35", "");
    valorS = digitalRead(LM35_pin);
    digitalWrite(LEDS, valorS);
    delay(1500);
    digitalWrite(LEDS, LOW);
    }

    if (text == "/led_UART") {
      bot.sendMessage(chat_id, "Sección UART", "");
      bot.sendMessage(chat_id, "Introducir: M --> encender led", "");
      if (Serial.available() > 0){
        char mensaje = Serial.read();
         if (mensaje == 'M') {
          digitalWrite(LEDU, LOW);
          delay(1500);
          digitalWrite(LEDU, HIGH);
          }
         else  digitalWrite(LEDU, LOW);

      }
    }
  }
}

void setup() {
  Serial.begin(115200);

  #ifdef ESP8266
    configTime(0, 0, "pool.ntp.org");
    client.setTrustAnchors(&cert);
  #endif

  pinMode(LEDI, OUTPUT);
  pinMode(LEDS, OUTPUT);
  pinMode(LEDU, OUTPUT);
  pinMode(DHTPIN, INPUT);
  pinMode(LM35_pin, INPUT);
  digitalWrite(LEDI, valorI);
  digitalWrite(LEDS, valorS);
  digitalWrite(LEDU, LOW);

  vTaskDelay(1000 / portTICK_PERIOD_MS); //wait for a second
  xTaskCreatePinnedToCore(Tarea1,
```

```
                          "Tarea01",
                          2048,
                          NULL,
                          1,
                          &Task1,
                          NULL);
  xTaskCreatePinnedToCore(Tarea2,
                          "Tarea02",
                          2048,
                          NULL,
                          1,
                          &Task2,
                          NULL);
  xTaskCreatePinnedToCore(Tarea3,
                          "Tarea03",
                          2048,
                          NULL,
                          1,
                          &Task3,
                          NULL);

  // Connect to Wi-Fi
  //Serial.println("Conectando al Servidor");
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  #ifdef ESP32
    client.setCACert(TELEGRAM_CERTIFICATE_ROOT);
  #endif
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Estableciendo conexión..");
  }

  Serial.println(WiFi.localIP());
}

void loop() {
  if (millis() > lastTimeBotRan + botRequestDelay)  {
    int numNewMessages = bot.getUpdates(bot.last_message_received + 1);

    while(numNewMessages) {
      Serial.println("got response");
      handleNewMessages(numNewMessages);
      numNewMessages = bot.getUpdates(bot.last_message_received + 1);
    }
```

```
    lastTimeBotRan = millis();
  }
}
```

A continuación, se muestra el armado físico en una placa de pruebas electrónica de los dos sensores y los leds que advierten las entradas a las secciones críticas.

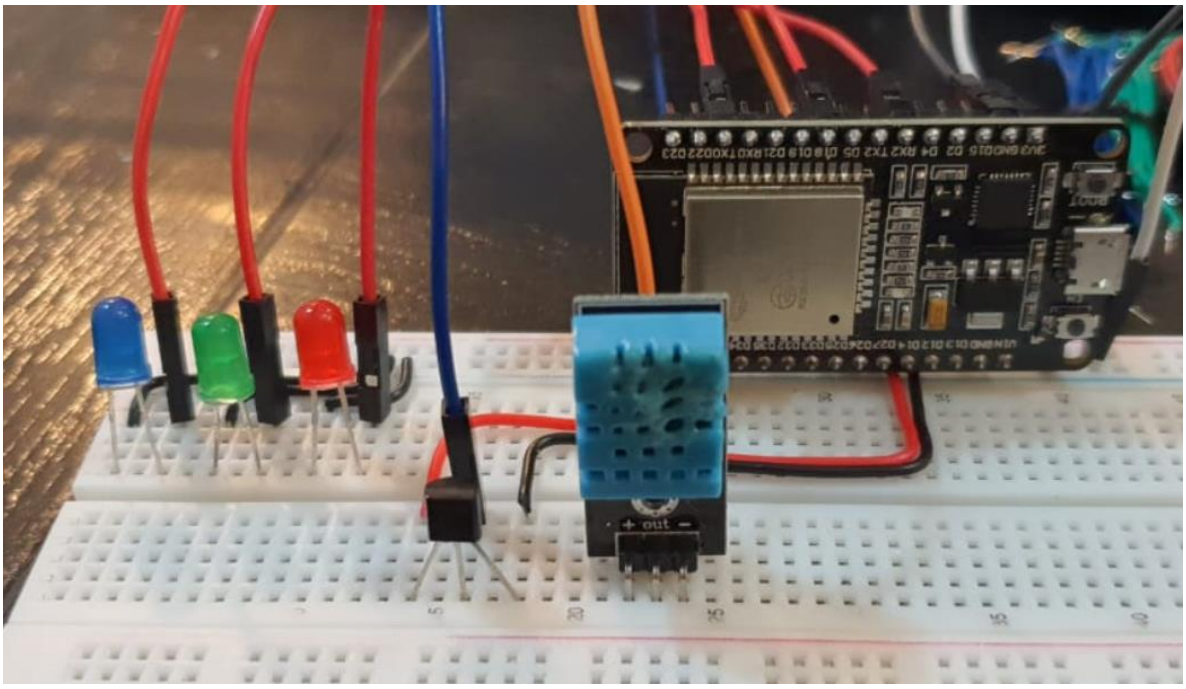

Ilustración 12. Armado físico de sensores

Elaboración propia

3. ANÁLISIS DE RESULTADOS

3.1. CÓDIGO BÁSICO DE EXCLUSIÓN MUTUA

Para esta parte del proyecto, el objetivo es garantizar que varias tareas compartan una sección crítica de manera segura, evitando condiciones de carrera y asegurando la consistencia de los datos. Los resultados se dividen en dos partes:

Dos tareas que comparten una sección crítica: Se implementa un código básico que permite permite que dos tareas compartan una sección crítica. Cada tarea envía un mensaje por UART y un carácter que identifica qué tarea ha utilizado la sección crítica. Los aspectos clave a considerar son:
Se utilizan dos tareas con la misma prioridad.
Se implementa un mecanismo de exclusión mutua para garantizar que solo una tarea a la vez pueda acceder a la sección crítica.
Cada tarea envía un mensaje y un carácter identificador cuando accede a la sección crítica, lo que permite verificar cuál tarea está operando en ese momento.

Cuatro tareas comparten la misma sección crítica: se modifica el código anterior para permitir que cuatro tareas compartan la misma sección crítica. Esto es un avance en la complejidad de la implementación, ya que ahora se deben gestionar más tareas y garantizar una exclusión mutua efectiva. Se añade dos tareas adicionales para un total de cuatro tareas compartiendo la sección crítica y modificamos el código para manejar esta mayor concurrencia.

En el análisis en equipo pudimos observar una implementación más compleja pero eficiente que permite a cuatro tareas compartir la sección crítica de manera segura. Vemos que es un sistema escalable y capaz de gestionar sus tareas concurrentes sin conflictos o desbordamientos.

3.2. ALGORITMO DE DEKKER Y PEATERSON

Pudimos observar ciertas ventajas, desventajas y algunas consideraciones usando el algoritmo de Dekker y Peaterson.

Algoritmo de Dekker:

Ventajas:
Implementación relativamente sencilla.
No requiere hardware adicional y se puede implementar en software.
Adecuado para sistemas con solo dos procesos que necesitan compartir recursos críticos.

Desventajas:
Puede experimentar inanición si uno de los procesos siempre se encuentra en la sección crítica.
Puede producir resultados impredecibles si no se implementa correctamente.
No es adecuado para sistemas con más de dos procesos que necesitan acceso a la sección crítica.

Algoritmo de Peterson:

Ventajas:
Implementación sencilla y comprensible.
No requiere hardware adicional y es fácil de implementar en sistemas multiprocesador.
Adecuado para sistemas con dos procesos que necesitan compartir recursos críticos.

Desventajas:

Puede sufrir inanición si uno de los procesos siempre se encuentra en la sección crítica.
La eficiencia puede disminuir en sistemas con un gran número de procesos que necesitan acceso a la sección crítica.
La complejidad aumenta en sistemas con más de dos procesos que necesitan acceso a recursos críticos.

Diferencias entre ambos códigos:

Ambos códigos se utilizan para lograr la exclusión mutua en sistemas con dos procesos que desean acceder a una sección crítica, y ambos utilizan una variable "turn" para decidir cuál de los dos procesos tiene la prioridad.
La diferencia principal radica en el uso de las variables "interested" en el algoritmo de Peterson, que proporciona una mayor claridad en la sincronización y la lógica de exclusión mutua al indicar el deseo de un proceso de acceder a la sección crítica.
En la implementación de Peterson, se utiliza un conjunto de variables booleanas "interested" para cada proceso, lo que permite una representación más explícita de las intenciones de los procesos.

Condiciones de utilidad:

Ambos algoritmos son útiles en sistemas con un número limitado de procesos que compiten por el acceso a una sección crítica, donde se requiere la exclusión mutua para garantizar la integridad de los datos compartidos.

Condiciones en las que no son tan útiles:

En sistemas con un gran número de procesos que necesitan acceso a una sección crítica, la implementación de Dekker o Peterson puede volverse complicada y menos eficiente. En estos casos, podrían preferirse soluciones más escalables, como semáforos o mutexes proporcionados por sistemas operativos en tiempo real.
Si los procesos no cooperan adecuadamente o si se produce un error en la implementación, tanto Dekker como Peterson pueden sufrir condiciones de carrera

y ciclos infinitos. Por lo tanto, se requiere una implementación cuidadosa y pruebas exhaustivas para garantizar su correcto funcionamiento.

Nota: es posible que ambos algoritmos caigan en ciclos infinitos si no se implementan correctamente o si los procesos no cooperan adecuadamente. Estos ciclos infinitos pueden surgir cuando los procesos se bloquean mutuamente y no pueden avanzar. Para evitar ciclos infinitos, es esencial que los procesos sigan las reglas de cooperación y se implementen correctamente los algoritmos. Los ciclos infinitos son situaciones no deseadas que deben evitarse en aplicaciones en tiempo real.

3.3. APLICACIÓN

El código desarrollado en esta parte es una aplicación que aborda un escenario en el que tres tareas comparten una sección crítica. Las tareas realizan adquisiciones de datos de sensores y procesamiento de señales, y la sección crítica implica la escritura en una base de datos y la validación de que se ha escrito correctamente. También implementamos un sistema de control de acceso que garantiza que las tareas se ejecuten en exclusión mutua.

A continuación, analizamos las partes más relevantes del código:

Librerías y configuración del bot de Telegram: El código comienza incluyendo las librerías necesarias y configurando la conexión WiFi y el bot de Telegram. Esto permite la comunicación con el ESP32 a través de mensajes de Telegram.

Configuración de sensores y pines: Se configuran los pines y se inicializan los sensores DHT11 y LM35. También se configuran los pines para los LEDs que indican las entradas a las secciones críticas.

Definición de las tareas: Se definen tres tareas utilizando la biblioteca FreeRTOS. Cada tarea tiene su propia función asociada: Tarea1, Tarea2 y Tarea3. Estas tareas son las que adquieren datos de sensores y ejecutan las secciones críticas.

Control de acceso a la sección crítica: Cada tarea solicita entrada a la sección crítica utilizando el arreglo "tarea" y la variable "turno". Si otra tarea ya está en la sección crítica o si no es su turno, la tarea actual espera. Una vez que se le permite el acceso, la tarea ejecuta la sección crítica y luego libera la exclusión mutua.

Manejo de mensajes de Telegram: La función handleNewMessages se encarga de manejar los mensajes de Telegram. Dependiendo del mensaje recibido, se realizan acciones como encender LEDs o enviar información de sensores a la base de datos.

Configuración inicial: En la función setup, se configuran las tareas, los pines y la conexión WiFi. También se inicializa la comunicación con el bot de Telegram.

Bucle principal: El bucle principal loop verifica y maneja los mensajes entrantes de Telegram y ejecuta las tareas relacionadas con los sensores y la sección crítica.

El código implementa un control de acceso a la sección crítica que se asemeja al algoritmo de Dekker, aunque la implementación es ligeramente diferente. En lugar de utilizar banderas "want" y "turn" como en el algoritmo de Dekker, se utilizan las variables "tarea" y "turno" para lograr la exclusión mutua.

El algoritmo de Dekker es adecuado para situaciones en las que se deben compartir recursos críticos entre dos procesos, y aquí se adaptó para su uso con tres tareas. Este algoritmo es relativamente sencillo y puede implementarse en software sin requerir hardware muy especializado. Además, es adecuado para sistemas con un número limitado de tareas que necesitan acceso exclusivo a una sección crítica.

Debido a que nosotros tenemos codigo dentro del void loop, se llegaba a desbordar la uart al momento de mandar cierto número de instrucciones y se queda en un ciclo infnito en donde no hacía nada.

Por recomendaciones del profesor, nos pidió dejar el void lopp vacio, y el codigo que está dentro del void loop debería de estar en una tarea aparte para posteriormente mandarla llamar cada que los sensores lo requirieran. Intemaomos hacerlo, pero se perdía la comunicación con le bot, esta es un área de oportunidad de nuestro código que se puede mejorar. El código lo dejamos como se muestra en la parte 2.2.1

Se hizo el cambio de la selección de la placa por el ESP32-Dev-Module en lugar de usar el ESP32 Wrover Kit All Version, ya que de esta manera podemos trabajar directamente con el chip ESP32, así se evitó el error en el bot al momento de conectarlo al servidor. Como plus, al trabajar con esta placa, no tenemos que hacer el proceso de reset con los botones de la placa al cargar el programa.

A continuación, se muestran algunas ilustraciones de los resultados de esta sección:

Ilustración 13. Monitor Serial con instrucciones recibidas del bot de telegram

Elaboración propia

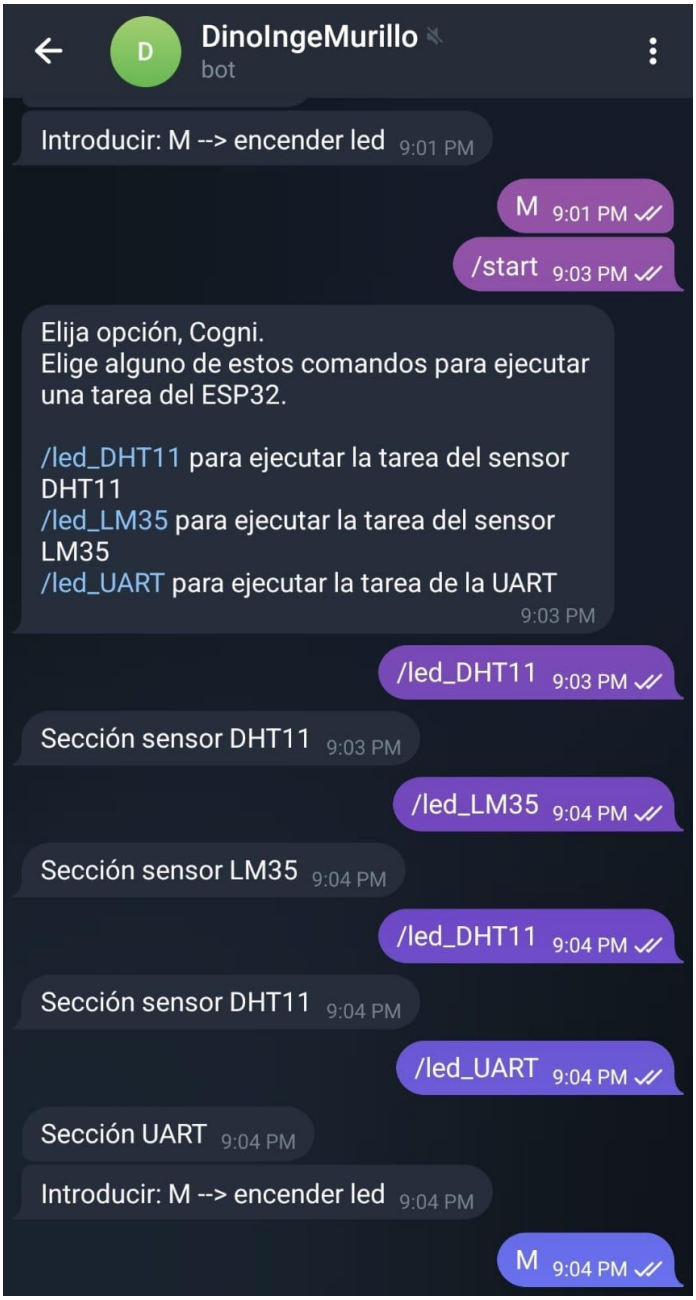

Ilustración 14. Menú de interacciónes Telegram Bot

Elaboración propia

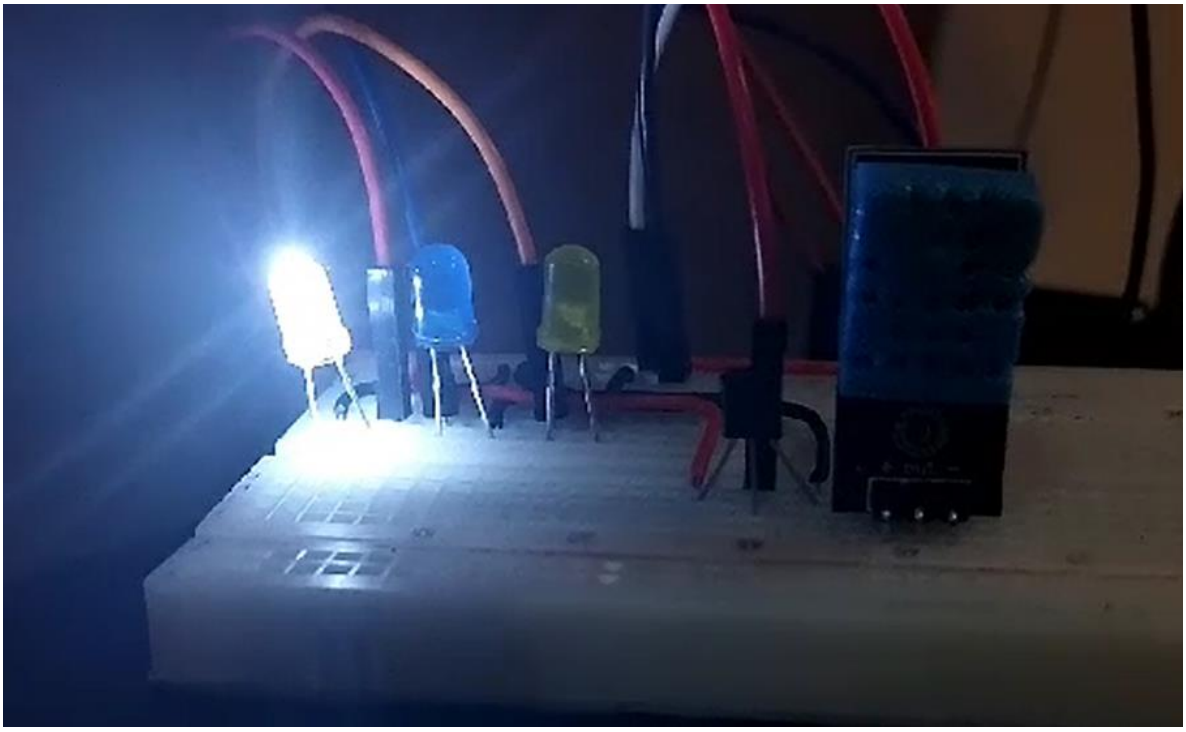

Ilustración 15. UART cuando le mandamos el carácter M lo compara con el carácter.

Elaboración propia

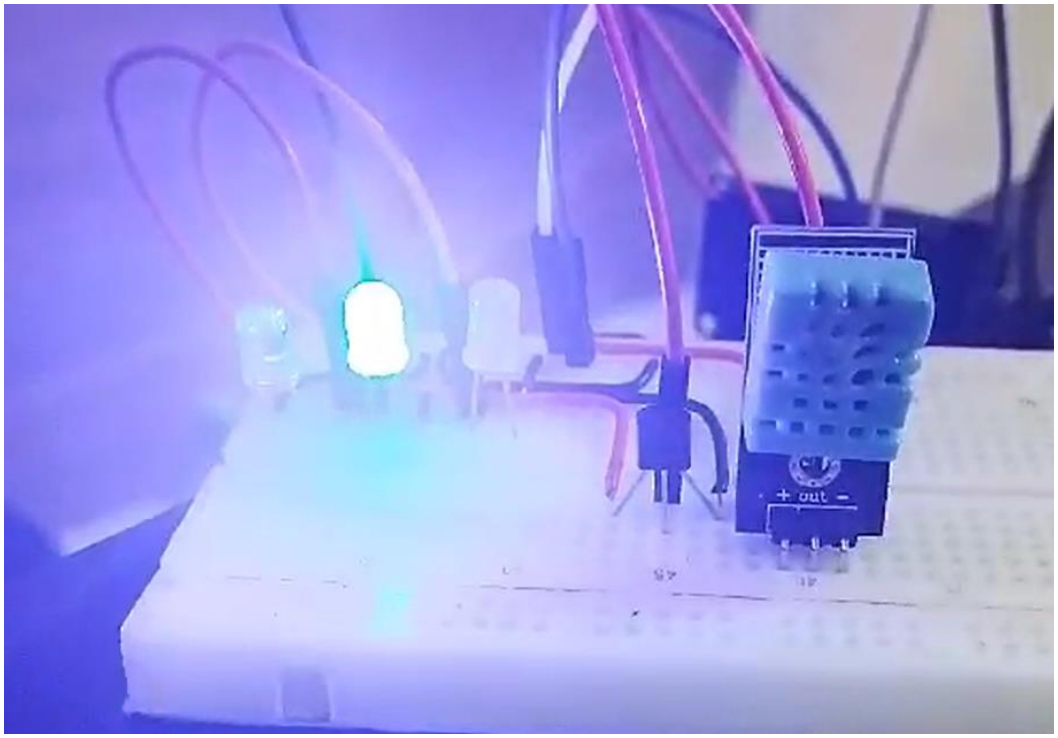

Ilustración 16. Luz de DH11
Elaboración propia

Esta se activa cuando a partir del bot seleccionamos el led DHT 11 si el bot recibe datos del sensor, el led encenderá un periodo de tiempo

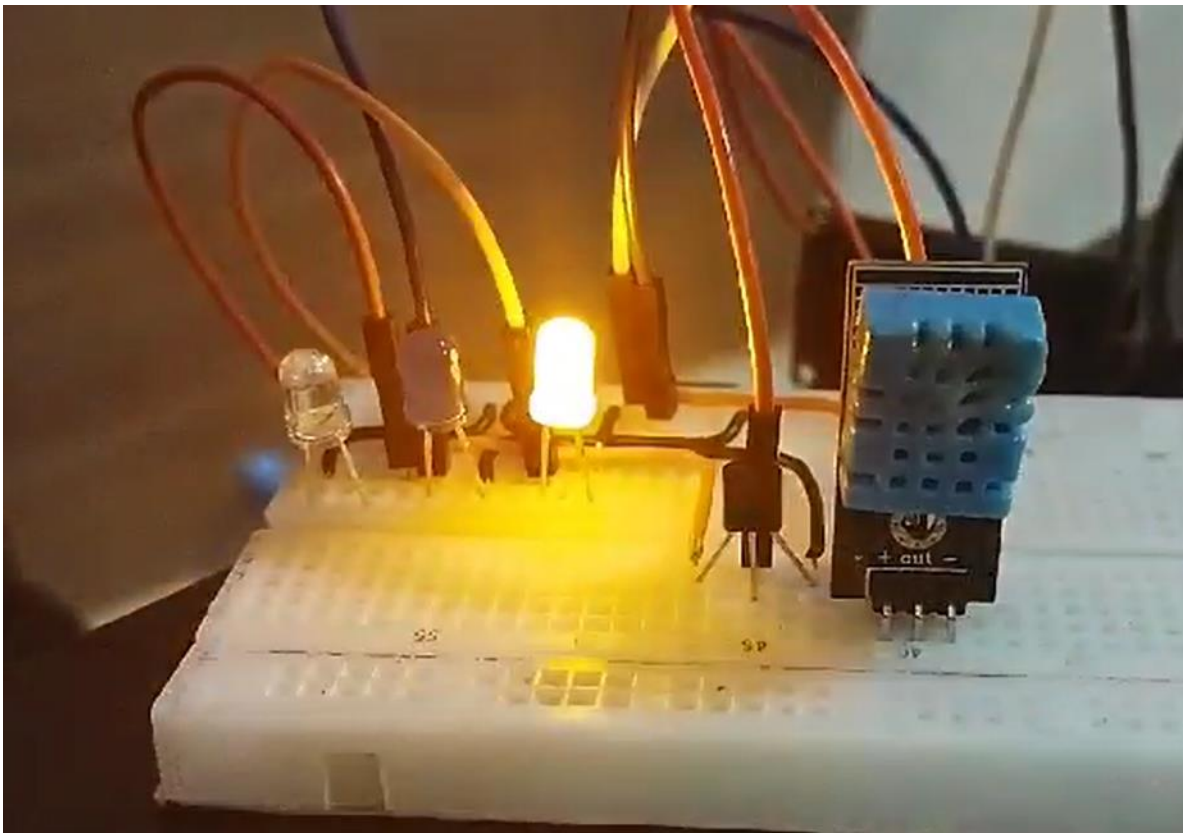

Ilustración 17. Luz de LM35

Elaboración propia

CONCLUSIONES

La práctica logra sus objetivos al proporcionar una comprensión sólida de los conceptos y técnicas fundamentales necesarios para la implementación de algoritmos de exclusión mutua. Abordamos ejemplos de código que demuestran la eficacia de estos algoritmos en situaciones de varias tareas compartiendo recursos críticos, subrayando la escalabilidad y las ventajas de aplicar este tipo de algoritmos a sistemas operativos en tiempo real.

Pudimos probar y evaluar los algoritmos de Dekker y Peterson, resaltando sus ventajas y desventajas en situaciones con procesos compartiendo recursos críticos. Ambos son útiles en sistemas con un número limitado de procesos, pero pueden enfrentar algo de problemas en entornos con un gran número de procesos compitiendo.

En la parte de aplicación se presenta un sistema completo de exclusión mutua en un sistema operativo en tiempo real. Se señala la necesidad de mejorar la organización de tareas y la comunicación con el bot de Telegram para evitar problemas como el desbordamiento en la UART.

La exclusión mutua es esencial en la programación concurrente. La elección del algoritmo de exclusión mutua y la plataforma de base de datos dependerá de las necesidades y las consideraciones específicas de la aplicación. Aplicando un sistema de exclusión de tareas podemos evitar caer en posibles ciclos infinitos o desbordamientos de código en el sistema.

REFERENCIAS

Ahmad, Z. (2006). *Principles of corrosion engineering and corrosion control.* Amsterdam: IChem; Elsevier.

Glez, B. (2015, Sep 18). *Prezi*. Retrieved from https://prezi.com/rcm4pmdmxrht/algoritmo-de-peterson/

Iturriaga de la Fuente, G. (1999). *México Patente n° 2268902.*

Monroy, M. e. (2014, Sep). *Sistemas operativos*. Retrieved from http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro26/exclusin_mutua.html

Monroy, M. e. (n.d.). *UAEH*. Retrieved from http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro26/index.html

Programación Concurrente. (n.d.). Retrieved from https://www.fing.edu.uy/tecnoinf/mvd/cursos/so/material/teo/so07-concurrencia.pdf

ramirez, j. (2015, Sep 22). *Prezi*. Retrieved from https://prezi.com/_nlbeeepa3oa/algoritmo-de-semaforo/

Sandoval, C. (2019, Sep 19). *Prezi*. Retrieved from https://prezi.com/wc43ow7ghjgr/algoritmo-de-dekker/

Sheila, W., Bisson, C., & Duffy, A. P. (2012). Applying a behavioural and operational diagnostic typology of competitive intelligence practice: empirical evidence from the SME sector in Turkey. *Journal of Strategic Marketing, 20*(1), 19-33. doi:http://dx.doi.org/10.1080/0965254X.2011.628450

SITES. (2019, Marzo 19). Retrieved from https://sites.google.com/site/concurrente9/introduccion?authuser=0

Song, Y., Rampley, C. P., & Chen, X. (2019). Application of bacterial whole-cell biosensors in health. In Y. Song, C. P. Rampley, & X. Chen, *Handbook of Cell Biosensors* (pp. 1-17). Springer Nature Switzerland AG.

Universidad de Valladolid. (n.d.). Retrieved from https://www.infor.uva.es/~fjgonzalez/apuntes/Tema6.pdf

Wikipedi. (2021, Dic 21). Retrieved from https://es.wikipedia.org/wiki/Monitor_(concurrencia)

Wikipedia. (2018, Mayo 26). Retrieved from https://es.wikipedia.org/wiki/Exclusi%C3%B3n_mutua_(inform%C3%A1tica)#Requisitos

ANEXOS

ANEXO A - Material Necesario

- Tarjeta de desarrollo ESP32.
- Conexión a Internet.

Arduino IDE.

- Sensor analógico y sensor digital.
- Elementos electrónicos básicos para acondicionamiento de señal de sensores.

ANEXO B – Repositorio Digital

Se adjunta la dirección electrónica del repositorio de códigos desarrollados por el equipo en esta práctica:

https://github.com/DonBerraco/RTOS