PL3 ~ 2024-25:

PROYECTO CARRERA

PRIMER CUATRIMESTRE

Universidad de Alcalá

Sistemas Operativos – Julia María Clemente Párraga - 8/1/2025

Luis Miguel Herrá Alpuente – luis.herra@edu.uah.es – 06612001F

Marcos Marín Morales – marcos.marinm@edu.uah.es - 03187230M

Índice

Consideraciones Previas

El comportamiento de CarreraHilos.c y su implementación sigue lo establecido en el enunciado de la PL3. Las funcionalidades prestadas por el código serán las siguientes:

- Creación de hilos
- Simulación de "carrera" con sus respectivas posiciones de llegada
- Finalizar la ejecución de dichos hilos

Los integrantes han intercambiado los roles de editor y programador a lo largo de todo el desarrollo del trabajo. A continuación, el código de la respectiva clase.

Carreras Hilos.h

Para mejorar la organización y claridad del código, hemos creado este archivo de cabecera. Aquí se definen las operaciones que se implementarán en CarrerasHilos.c, un factor clave fue el poder añadir los include únicamente aquí.

```c
#ifndef CARRERAHILOS_H
#define CARRERAHILOS_H

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define N_COCHES 8

// Declaración de la estructura coche_t
typedef struct
{
    int id;
    char *cadena;
} coche_t;

// Declaración de variables globales
extern coche_t Coches[N_COCHES];
extern volatile int clasificacionFinal[N_COCHES];
extern volatile int posicion;
extern pthread_mutex_t mutex;

// Declaración de funciones
void *funcion_coche(void *arg);
void mostrarMensajeAscii();

#endif // CARRERAHILOS_H
```

Carreras Hilos.c

Funciones:

```c
#include "CarreraHilos.h"

// Definición de variables globales
coche_t Coches[N_COCHES];
volatile int clasificacionFinal[N_COCHES];
volatile int posicion = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Definición de la función
void *funcion_coche(void *arg)
{
    coche_t *pcoche = (coche_t *)arg;
    int aleatorio;
    unsigned int semilla = (pcoche->id) + pthread_self();
    printf("Salida de %s %d\n", pcoche->cadena, pcoche->id);
    fflush(stdout);

    aleatorio = rand_r(&semilla) % 10;
    sleep(aleatorio);

    printf("Llegada de %s %d\n", pcoche->cadena, pcoche->id);

    pthread_mutex_lock(&mutex);
    clasificacionFinal[posicion] = pcoche->id;
    posicion++;
    pthread_mutex_unlock(&mutex);

    pthread_exit(NULL);
}

void mostrarMensajeAscii()
{
    printf("                      _____________________\n");
    printf("                   //|          |       \\\\\n");
    printf("                 //  |          |         \\\\\n");
    printf("      ___________//___|__________|_________()\\\\_________________\n");
    printf("    /_________________|_=________|_=__________|_________________{}\n");
    printf("    [           _____ |          | .          | ==  ______      { }\n");
    printf("  __[__        /##  ##\\\\|          |            |    /##  ##\\\\    _{# }_\n");
    printf(" {_____)_____|##    ##|__________|____________|___|##    ##|__(_____}\n");
    printf("            /  ##__##                            /  ##__##        \\\\\n");
    printf("-------------------------------------------------------------------------------\n");
    printf("                         ¡Bienvenido a la carrera de coches!\n");
}
```

Función Main:

```
int main(void)
{
    mostrarMensajeAscii();
    pthread_t hilosCoches[N_COCHES];
    int i;

    printf("Se inicia proceso de creación de hilos...\n\n");
    printf("SALIDA DE COCHES\n");

    for (i = 0; i < N_COCHES; i++)
    {
        Coches[i].id = i;
        Coches[i].cadena = "Coche";

        if (pthread_create(&hilosCoches[i], NULL, funcion_coche, (void
*)&Coches[i]) != 0)
        {
            perror("Error al crear hilo");
            exit(EXIT_FAILURE);
        }
    }

    printf("Proceso de creación de hilos terminado\n\n");

    for (i = 0; i < N_COCHES; i++)
    {
        if (pthread_join(hilosCoches[i], NULL) != 0)
        {
            perror("Error al esperar el hilo");
            exit(EXIT_FAILURE);
        }
    }

    // Mostramos la clasificación final
    printf("Clasificación final:\n");
    printf("+---------------------------------+\n");
    printf("| Posición |       Coche          |\n");
    printf("+---------------------------------+\n");
    for (i = 0; i < N_COCHES; i++)
    {
        printf("|    %2d    |     Coche %2d       |\n", i + 1,
clasificacionFinal[i]);
    }
    printf("+---------------------------------+\n");

    return 0;
}
```

Explicación de la Práctica:

La práctica no es una carrera real realmente. Se crean tantos hilos como coches hayamos definido. Cada hilo representa un coche y simula su recorrido con un tiempo aleatorio basado en una semilla específica que asegura que los tiempos sean independientes. Cada hilo tiene una semilla, de esta forma garantizamos que no toquen valores aleatorios iguales.

Cada hilo, ejecuta la función_coche; calcula su tiempo aleatorio, y luego hace un sleep por ese tiempo. Este sleep solo afecta al hilo del coche, como si estuviera avanzando. Al terminar, imprime que ha llegado y actualiza su posición en un array compartido donde almacenamos la clasificación.

El problema es que este arreglo es que puede ser modificado al mismo tiempo por varios hilos si no lo controlamos. Por eso usamos mutex[1], que actúa como un "semáforo" que bloquea el acceso al array hasta que un hilo termina su actualización. Esto evita errores que ocurrirían si dos coches intentaran escribir al mismo tiempo.

Al final, el hilo principal espera a que todos los coches terminen, usando pthread_join. Cuando han llegado, el programa muestra la clasificación final. Dado que los hilos se crean casi simultáneamente, es como si todos empezaran la carrera al mismo tiempo[2], y cada uno termina según el tiempo aleatorio que le tocó. Sin el mutex, si dos coches terminan al mismo tiempo, se podría desordenar la clasificación.

1. Un ejemplo útil para entender el funcionamiento del mutex es compararlo con una caja fuerte. Imagina que para abrir la caja, necesitas una llave. Cuando un hilo quiere acceder al recurso (como un array dinámico), primero verifica si la llave está disponible. Si está libre, toma la llave, accede al recurso y lo modifica. Cuando termina, devuelve la llave. Si otro hilo intenta acceder mientras la llave está ocupada, deberá esperar hasta que el hilo anterior termine y devuelva la llave.
2. Los hilos se ejecutan de manera concurrente, es decir, de manera paralela desde un primer momento. Cuando un hilo se crea, pasa a “competir” por recursos de la CPU junto con el hilo principal y otros hilos. Por eso, es posible que el hilo principal termine de ejecutar las líneas donde se imprimen mensajes como "Proceso de creación de hilos terminado" antes de que todos los hilos hayan tenido la oportunidad de ejecutar su parte y mostrar su mensaje de salida. Este suceso genera una pequeña desincronización en ocasiones en los mensajes mostrados por el programa.

Ejercicios:

1.1 Completa la línea 53 (Código 1): inicializar la estructura e invocar a los hilos.

```
    for (i = 0; i < N_COCHES; i++)
    {
        Coches[i].id = i;
        Coches[i].cadena = "Coche";

      if (pthread_create(&hilosCoches[i], NULL, funcion_coche, (void *)&Coches[i]) != 0)
        {
            perror("Error al crear hilo");
            exit(EXIT_FAILURE);
        }
    }
```

Explicación:

A partir de un bucle iterativo for, inicializamos las estructuras de los coches; como se observa en su definición, sus atributos son el id y un nombre. El id, identificador único, será el número de iteraciones del bucle (índice i del bucle), siendo estas únicas y el nombre la cadena “Coche”; posteriormente en la clasificación mostraremos cada estructura.

A continuación, se invocan los hilos a partir de pthread_create y en caso de error, se mostrará error al crear el hilo terminando el programa. Los argumentos de este son &hilosCoches, siendo este un puntero al identificador del hilo, almacenará el id del nuevo hilo; NULL que establece predeterminado los atributos del hilo; función_coche siendo esta, evidentemente, la que ejecutará el hilo; (void *)… es el argumento que se pasa a la función del hilo, en este caso, la dirección del array de coches.

1.2 Completa la línea 38 (Código 2)

```
pthread_exit(NULL);
```

Explicación:

Termina la ejecución del hilo; NULL implica que el hilo no devuelve ningún valor específico al finalizar.

Ejercicios:

1.3 Completa la línea 63 (Código 3)

```
for (i = 0; i < N_COCHES; i++)
{
    if (pthread_join(hilosCoches[i], NULL) != 0)
    {
        perror("Error al esperar el hilo");
        exit(EXIT_FAILURE);
    }
}
```

Explicación:

Este código espera a que un hilo termine su ejecución antes de continuar. Se esperan tantos como fueros generados (N_COCHES), hilosCoches[i] referencia el identificador del hilo que se espera, NULL implica que no se captura el valor de retorno del hilo (lógico dado 1.1). La función pthread_join, devolverá 0 si la espera fue exitosa, en caso, de que cualquiera de los hilos no verifiquen esto, muestra un error y termina con el ciclo de ejecución del programa.

2. ¿Qué efecto produce la función rand_r() en la función funcion_coche() que ejecuta cada hilo?

La función rand_r () nos garantiza que los valores aleatorios generados sean únicos de cada hilo. La función rand_r es reentrante lo que significa que puede ser utilizada sin problema en las aplicaciones multihilo sin necesidad de emplear mutex. De esta forma, cada hilo tiene su propia semilla de generación. Esto no implica que no puedan generar el mismo valor aleatorio dos hilos, sino que la generación de dicho valor no interfiere en el proceso de generación del otro.

3. ¿Qué ocurre si el hilo inicial (que ejecuta la función main()) no espera la finalización del resto de hilos?

Si el hilo principal no espera la finalización de los hilos secundarios, puede terminar antes que ellos, lo que provoca que el programa finalice abruptamente y los hilos no puedan completar su ejecución, interrumpiendo la simulación de la carrera. En otras palabras, esto evitará que en gran parte de los casos muchos de los hilos puedan terminar su tarea; son interrumpidos de manera abrupta mientras realizan su tarea.

Para evitar esto, es que realizamos lo visto en el ejercicio 1.3 con pthread_join.

4. El enunciado especifica que, cuando todos los coches hayan llegado a la meta, el hilo padre se encargará de mostrar en pantalla la clasificación final. Respecto a esta funcionalidad, responda a las siguientes cuestiones:

4.1 ¿Es correcto pensar en obtener la clasificación mediante una solución en la que el padre, a medida que vaya finalizando cada hilo (coche) por el que espera (es decir, tras el código incluido en las líneas 60 a 65), vaya imprimiendo el identificador del coche que ha finalizado?

Es totalmente incorrecto. Según se ha planteado en todo este documento, trabajamos con hilos de forma concurrente, en otras palabras, los procesos ligeros funcionan de manera paralela. Además, como se ha reflejado en el código, la duración de estos hilos depende de un valor generado aleatoriamente, con su respectiva semilla (rand_r). Es decir, el primer hilo creado podría ser tanto el primero, segundo como el último en terminar su proceso; consecuentemente el último en clasificación. Además, habría de añadirse la componente, previamente mencionada, de la "disputa" por los recursos. Los resultados son impredecibles. Por eso sincronizamos los hilos y mostramos sus resultados una vez todos hayan terminado su respectivo proceso.

4.2 Resuelva el problema planteado en este apartado ejecutando los pasos que a continuación se describen:

a) Declara las siguientes variables

```
volatile int clasificacionFinal[N_COCHES];
volatile int posicion = 0;
```

La palabra clave volatile en C se usa para indicar que una variable puede ser modificada de manera inesperada fuera del control del programa, como en el caso de variables que son accesibles por múltiples hilos. La palabra volatile, le indica al compilador que no debe optimizar dicha variable; que sea guardada en un registro y se acuda a dicho registro en lugar de a memoria.

b) Inserte las líneas de código que considere necesarias en el espacio indicado mediante el comentario CODIGO 4 para que, al finalizar cada hilo coche la carrera, almacene su identificador (campo id) en la siguiente posición vacía de clasificacionFinal[]

```
    pthread_mutex_lock(&mutex);
    clasificacionFinal[posicion] = pcoche->id;
    posicion++;
    pthread_mutex_unlock(&mutex);
```

Tal y como ha sido comentado anteriormente, el mutex sincroniza el acceso de los hilos a las variables clasificacionFinal y posición. El hilo realiza un bloqueo exclusivo al llegar a esta sentencia, haciendo que solo él pueda modificar dichas variables, posteriormente libera dicho bloqueo para el resto de hilos. En el array se introduce el id del coche clasificado, su índice será su posición. Posteriormente, se aumenta una unidad el valor del posición, moviendo el puntero a la siguiente posición del array donde ingresará de igual forma el siguiente coche, del hilo que venga a continuación.

c) ¿Se podrían producir errores al ejecutar las operaciones anteriores sin ningún tipo de control? ¿Y con la función rand_r()? Razone claramente la respuesta. En caso afirmativo, indique cómo solucionaría el problema mencionando alguna de las funciones incluidas en la Tabla 1 (se pide sólo indicar las funciones sin codificar en este apartado)

En caso de no haber control, las operaciones clasificacionFinal[] se podrían producir errores como ha sido mencionado, dos hilos podrían intentar acceder de manera simultánea para modificar el array. Para solucionar este problema, añadimos el mutex sincronizando su acceso entre hilos. En relación a la función rand_r, también ha sido comentado, al generarse con semillas diferentes mitigamos cualquier problema relativo a la generación de valores aleatorios.

d) Si la respuesta a la pregunta anterior ha sido afirmativa, codifique correctamente el programa incluyendo la solución al problema indicado.

Ya ha sido codificada en los apartados anteriores, el resultado global se observa al inicio de este proyecto.

e) Inserte las líneas de código que considere necesarias en el espacio indicado mediante el comentario CODIGO 5 para que el hilo padre, una vez que todos los hilos hayan llegado a la meta, muestre en pantalla la clasificación final de la carrera accediendo a cada una de las posiciones del array clasificacionFinal[].

```
    // Mostramos la clasificación final
    printf("Clasificación final:\n");
    printf("+---------------------------------+\n");
    printf("| Posición |       Coche          |\n");
    printf("+---------------------------------+\n");
    for (i = 0; i < N_COCHES; i++)
    {
        printf("|    %2d    |     Coche %2d       |\n", i + 1,
clasificacionFinal[i]);
    }
    printf("+---------------------------------+\n");

    return 0;
}
```

Se trata de un bucle for que recorre las diferentes posiciones del array clasificacionFinal. El índice + 1 del bucle es la posición seguido del coche y su id, accedido desde clasificacionFinal[i]. Se muestra en un formato de tabla gracias al resto de printf.

5. Realice un makefile que permita generar correctamente la aplicación y que incluya un objetivo ficticio clean, tal y como ya se ha explicado en el laboratorio.

```
CC = gcc # Compilador a usar

CFLAGS = -Wall -pthread # Flags de compilación

OBJ = CarreraHilos.o # Archivos objeto

# Crear ejecutable 'CarreraHilos' a partir de 'CarreraHilos.o'

CarreraHilos: $(OBJ)
    # Compila y enlaza el ejecutable
    $(CC) $(CFLAGS) -o CarreraHilos $(OBJ)
# Crear archivo objeto 'CarreraHilos.o' a partir de 'CarreraHilos.c' y 'CarreraHilos.h'

CarreraHilos.o: CarreraHilos.c CarreraHilos.h
    $(CC) $(CFLAGS) -c CarreraHilos.c

clean: # Limpiar archivos generados, elimina archivo objeto y ejecutable

    rm -f *.o CarreraHilos
```