**P3 SISTEMAS OPERATIVOS**

Práctica 3: Servicios POSIX para la  
gestión de hilos en Linux.

Marcos Bonilla de las Morenas

Miguel de Sande Moreno

Grupo Lunes 08:00 - 10:00

Profesora: Julia María Clemente Párraga

1. **Como aplicación de los servicios POSIX básicos de gestión de hilos, complete en el programa las líneas que a continuación se indican:**

**1.1 Línea 53 indicada mediante el comentario CODIGO 1. Inicialice la estructura de cada coche e invoque el servicio POSIX que considere adecuado.**

*/\* CODIGO 1 \*/*

Coches[i].id = i;  
 Coches[i].cadena = "Coche";

if (pthread\_create(&hilosCoches[i], NULL,  
 (void \*(\*)(void \*))funcion\_coche,  
 (void \*)&Coches[i]) != 0) {

perror("Error creando hilo");  
 exit(EXIT\_FAILURE);

}

Este código se ejecuta tantas veces como N\_COCHES valga, ya que está dentro de un bucle for. Por cada iteración del mismo se definirá un nuevo coche, asignándole como id lo que valga ien ese momento, y en cadena se le asigna “Coche”. Después se crea un hilo para ese coche, que ejecutará la función funcion\_coche(), con el servicio. pthread\_create().

**1.2 Línea 38 indicada mediante el comentario CODIGO 2. Invoque el servicio POSIX que considere adecuado.**

*/\* CODIGO 2 \*/*

pthread\_exit(NULL);

Simplemente se necesita que el hilo termine cuando acaba de ejecutar la función, por lo que se ejecuta este servicio POSIX al final de la misma.

**1.3 Línea 63 indicada mediante el comentario CODIGO 3. Invoque el servicio POSIX que considere adecuado.**

*/\* CODIGO 3 \*/*

if (pthread\_join(hilosCoches[i], NULL) != 0) {

perror("Error esperando hilo");  
 exit(EXIT\_FAILURE);

}

Este código hace que el programa principal espere a que todos los coches hayan terminado (se haya ejecutado pthread\_exit(NULL); en todos los hilos) Está dentro de un bucle for, por cada iteración se ejecuta el servicio pthread\_join() para cada hilo, estando sus identificadores guardados en el array hilosCoches[].

**2. ¿Qué efecto produce la función rand\_r() en la función funcion\_coche() que ejecuta cada hilo?**

rand\_r(), en función de una semilla que se le pase como parámetro, genera un número aleatorio. En el código, al resultado de rand\_r() se le aplica %10 para obtener un número aleatorio entre 0 y 9. Este número se guarda en una variable aleatorio, para después utilizarla como parámetro en la función sleep(aleatorio). De esta manera cada coche tardará un tiempo diferente y aleatorio en llegar a la meta.

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

Si no espera al resto de hilos, además de imprimir incorrectamente que todos los coches han llegado a la meta, el proceso termina. Cuando el hilo principal finaliza, el proceso completo termina y el sistema operativo limpia todos los recursos dedicados a él. Como no son independientes entre ellos como los procesos pesados, sino que comparten todos los recursos, los hilos de los coches también terminan en medio de su ejecución.

**4. El enunciado especifica que, cuando todos los coches hayan llegado a la meta, el hilo padre se encargará de mostrar en la 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 se espera (es decir, tras el código incluido en las líneas 60 a 65), vaya imprimiendo el identificador del coche que ha finalizado? Razone su respuesta.**

No es correcta esa solución, ya que el padre no podrá seguir ejecutándose hasta que todos los hilos hayan terminado, al estar esperando a que termine cada uno de ellos (está programado en el código 3). Para que se vaya imprimiendo el identificador de los coches que van finalizando, tienen que ser los propios hilos los que ejecuten el código antes de llamar a pthread\_exit().

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

1. **Declare las siguientes variables globales**

volatile int clasificacionFinal[N\_COCHES]  
volatile int finalCarrera = 0

1. **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[]**.**

*/\* CODIGO 4 \*/*

clasificacionFinal[finalCarrera] = pcoche->id;  
 finalCarrera++;

Este código guarda en el array clasificacionFinal[] el id del coche que ejecuta el código en ese momento. Para saber en qué posición guardar el id se utiliza la variable finalCarrera, que empieza siendo 0 y indica en qué posición hay que guardar el id. Una vez guardado, la variable se incrementa en 1, así el siguiente coche guardará su id en la siguiente posición.

1. **¿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).**

Sí podría producirse algún error. En caso de que dos coches terminen a la vez, ambos hilos querrán acceder y modificar las variables al mismo tiempo, lo que provocaría una condición de carrera entre ellos, lo que provocaría que los datos de la clasificación no sean fiables dependiendo de cuál de los dos haya escrito primero, o quedar corrupta la información.

Este problema puede solucionarse con los mutex, que garantizan que un hilo acceda de forma exclusiva a las variables, impidiendo así que otros hilos puedan acceder al mismo tiempo. En caso de que se quiera modificar una variable y el mutex esté desbloqueado, el hilo lo bloqueará, escribirá en ella y lo desbloqueará. Si está bloqueado, el hilo se suspende hasta que el mutex vuelva a quedar libre. De esta forma se interrumpe el flujo de los hilos que se encuentran con un mutex bloqueado por otro hilo, reanudando su flujo cuando el mutex es desbloqueado, garantizando así que no se puedan modificar variables a la vez y producir condiciones de carrera.

Para ello, primero se debe crear el mutex con el servicio POSIX pthread\_mutex\_init(). Después, para bloquearlo se utiliza el servicio pthread\_mutex\_lock(), y para desbloquearlo se utiliza pthread\_mutex\_unlock(). Cuando no se vaya a utilizar más, se destruye con pthread\_mutex\_destroy().

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

Primero declaro como variable global el mutex con

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

Después inicializo el mutex al principio de la función main() con

*// Inicializar el mutex*

if (pthread\_mutex\_init(&mutex, NULL) != 0) {

perror("Error inicializando mutex");  
 exit(EXIT\_FAILURE);

}

Ahora ya puedo utilizar el mutex para bloquear el flujo de los hilos para que no se produzcan condiciones de carrera entre ellos. Para ello utilizo este código:

*/\* CODIGO 4 \*/*

*// Bloqueo el mutex*

if (pthread\_mutex\_lock(&mutex) != 0) {

perror("Error bloqueando mutex");  
 exit(EXIT\_FAILURE);

}

*// Guardamos la clasificacion del coche*

clasificacionFinal[finalCarrera] = pcoche->id;  
 finalCarrera++;

*// Desbloqueo el mutex*

if (pthread\_mutex\_unlock(&mutex) != 0) {

perror("Error desbloqueando mutex");  
 exit(EXIT\_FAILURE);

}

Este código, antes de modificar las variables, llama al servicio pthread\_mutex\_lock() para no permitir a otros hilos acceder mientras se modifican las variables, y desbloqueando el mutex con pthread\_mutex\_unlock() cuando ha terminado de modificarlas.

1. **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[].

*/\* CODIGO 5 \*/*

printf("CLASIFICACION FINAL:\n");

for (i=0; i<N\_COCHES; i++)  
 {  
 printf( "%dº: Coche %d\n", i+1, clasificacionFinal[i]+1);  
 }

El código simplemente recorre el array clasificacionFinal[], que es donde se guarda la posición final de los coches, y lo va imprimiendo por pantalla.

**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. Se valorará además la estructuración del código de la aplicación en sendos archivos fuentes .c e incluir un archivo de cabecera simula\_car.h con los contenidos adecuados.**

El archivo makefile es el siguiente:

CC = gcc  
CFLAGS = -Wall -Wextra -Werror -g

*#Ruta de los archivos .c*RUTA\_C = src

all: bin/simula\_car

.PHONY: clean   
clean: *#Borra el ejecutable y los archivos objeto* rm -rf bin/\* build/\*  
 rm -rf bin build

build: *#Crea la carpeta build* mkdir -p build

bin: *#Crea la carpeta bin* mkdir -p bin

bin/simula\_car: bin build build/funcion\_coche.o build/simula\_car.o  
 $(CC) $(CFLAGS) -o bin/simula\_car build/funcion\_coche.o build/simula\_car.o —lpthread

build/funcion\_coche.o: $(RUTA\_C)/funcion\_coche.c  
 $(CC) $(CFLAGS) -c $(RUTA\_C)/funcion\_coche.c -o build/funcion\_coche.o

build/simula\_car.o: $(RUTA\_C)/simula\_car.c  
 $(CC) $(CFLAGS) -c $(RUTA\_C)/simula\_car.c -o build/simula\_car.o

Se definen las variables CC para guardar el compilador a usar, CFLAGS para guardar las flags con las que se quiere compilar el código, y RUTA\_C para guardar la ruta a los archivos fuente.

Se crea la regla all, que servirá para compilar normalmente el código.

Se crea la regla clean, para borrar todos los archivos generados al compilar

Se crean dos reglas, build y bin, para crear las carpetas build y bin.

Se crea la regla que bin/simulacar, que es la que generará el ejecutable, con la flag -lpthread

Se crean las reglas para generar los archivos objeto de los diferentes archivos .c

REESTRUCTURACIÓN DEL CÓDIGO:

Se han creado dos carpetas: src y include. En src estarán todos los archivos .c, y en include los .h. El archivo .h creado es simula\_car.h:

#ifndef SIMULA\_CAR\_H  
#define SIMULA\_CAR\_H  
#include <pthread.h>

#define N\_COCHES 8

*// Tipo de datos que representa un coche*typedef struct {

int id;  
 char \*cadena;

} coche\_t;

extern coche\_t Coches[N\_COCHES];  
extern volatile int clasificacionFinal[N\_COCHES];  
extern volatile int finalCarrera;  
extern pthread\_mutex\_t mutex;

void \*funcion\_coche(coche\_t \*pcoche);

#endif *// SIMULA\_CAR\_H*

En él se define la estructura de coche\_t y se indican las variables y funciones.

Además, se ha dividido el archivo simula\_car.c en dos: uno con el mismo nombre y otro funcion\_coche.c. De esta manera funcion\_coche.c tiene la función funcion\_coche(), y simula\_car.c solo tiene la función main() y las variables globales, ya que la definición de coches\_t está en simula\_car.h. Ambos archivos importarán simula\_car.h.

**EXTRAS:**

Hay un pequeño problema en el código a la hora de imprimir por pantalla lo que va ocurriendo, a veces el mensaje que indica que un coche ya ha salido se imprime después del mensaje "Proceso de creación de hilos terminado". Para arreglarlo hemos creado otro mutex más que se cerrará antes de crear los hilos y se volverá a abrir cuando hayan terminado de crearse.

El código es el siguiente:

Defino en simula\_car.h un mutex nuevo:

extern pthread\_mutex\_t mutexInicio;

En simula\_car.c, declaro el mutex y dentro de main() lo inicializo:

pthread\_mutex\_t mutexInicio = PTHREAD\_MUTEX\_INITIALIZER;

*// Inicializar el mutex para que los hilos no empiecen hasta que se hayan creado todos*

if (pthread\_mutex\_init(&mutexInicio, NULL) != 0) {

perror("Error inicializando mutex");  
 exit(EXIT\_FAILURE);

}

Después, antes y después del bucle for que crea los hilos, hay que bloquar y desbloquear el mutex respectivamente:

*// Bloqueamos el mutex para que los hilos no empiecen hasta que se hayan creado todos*

if(pthread\_mutex\_lock(&mutexInicio) != 0) {

perror("Error bloqueando mutex");  
 exit(EXIT\_FAILURE);

}

for (i=0; i<N\_COCHES; i++)

{

*/\* CODIGO 1 \*/*

*// Inicializamos los datos del coche* Coches[i].id = i;  
 Coches[i].cadena = "Coche";

*// Creamos el hilo del coche*

if (pthread\_create(&hilosCoches[i], NULL,  
 (void \*(\*)(void \*))funcion\_coche,  
 (void \*)&Coches[i]) != 0) {

perror("Error creando hilo");  
 exit(EXIT\_FAILURE);

}

}

*// Desbloqueamos el mutex para que los hilos puedan empezar*

if (pthread\_mutex\_unlock(&mutexInicio) != 0) {

perror("Error desbloqueando mutex");  
 exit(EXIT\_FAILURE);

}

Y por último, en funcion\_coche.c, bloqueo el mutex al principio de funcion\_coche() para que espere, y seguidamente lo desbloqueo para que los hilos no se bloqueen entre ellos:

*// Bloqueamos el mutex para que los hilos no empiecen hasta que se hayan creado todos*

if (pthread\_mutex\_lock(&mutexInicio) != 0) {

perror("Error bloqueando mutex");  
 exit(EXIT\_FAILURE);

}

*// Desbloqueamos el mutex para que los demás hilos también puedan empezar*

if (pthread\_mutex\_unlock(&mutexInicio) != 0) {

perror("Error desbloqueando mutex");  
 exit(EXIT\_FAILURE);

}