PRÁCTICA 3 –
Programación multi-hilo

Autores:
Guillermo Rubio Bolger (NIA: 100487574).
Jaime Rabazo Fernández (NIA: 100499019).
Daniele Laporta (NIA: 100502703).

Grupo de clase: Grupo 801.

Titulación cursada: Doble Grado en Ingeniería Informática y Administración de Empresas.

Tabla de contenido

1. Introducción

En este documento se va a exponer y explicar la resolución implementada por los integrantes del grupo para los enunciados propuestos en la práctica. El documento está dividido en tres secciones. La primera describe la implementación del código y las decisiones de diseño tomadas, la segunda especifica las pruebas realizadas sobre el programa, y la última abarca las dificultades enfrentadas y lo que hemos aprendido durante el desarrollo de la práctica. Antes de comenzar, hemos de mencionar que el código ha sido desarrollado y comentado mayoritariamente en inglés, ya que hay integrantes del grupo procedentes del programa Erasmus.

2. Descripción del código

En este apartado de la memoria, se van a detallar las principales funciones desarrolladas, en orden de ejecución lineal del código. Se explicarán las estructuras de datos implementadas y utilizadas, la ejecución de operaciones bancarias, y se pondrá mayor énfasis en el funcionamiento concurrente de los hilos *"Trabajador"* y *"Cajero"*.

2.1. Cola compartida *queue*

Antes de empezar con la ejecución lineal del código, es imprescindible conocer el funcionamiento de la cola compartida utilizada para implementar el sistema bancario. Nuestra estructura *"queue"* consta de cuatro atributos que permiten su funcionamiento. En primer lugar, una lista de punteros a estructuras *"struct element"* llamado *"operation_queue"*, que servirá para saber que elementos están dentro de la cola y en que posición. En segundo lugar, dos número enteros *"int to_put"* e *"int to_get"*que representarán la posición de la lista donde se debe introducir el siguiente elemento con el método*"queue_put"*, y la posición de donde se debe extraer el siguiente elemento con el método *"queue_get"*. En tercer lugar, una variable entera que representa el número de elementos dentro de la cola, cuya finalidad será la implementación de los métodos, *"queue_empty"* y *"queue_full"* que retornarán si la cola está vacía o llena. Por último, el tamaño *"int size"* de la cola, para la inicialización de *"operation_queue"*, y su uso en los métodos de la estructura.

Los métodos utilizados, tras la inicialización de la cola compartida, son *"queue_put"* y *"queue_get"*. Estas funciones están diseñadas para, a través de la consulta y la modificación de los atributos previamente enumerados, implementar una estructura que cumple estrictamente la restricción FIFO a modo circular, y que comprueba y retorna error al intentar introducir elementos cuando la cola está llena o extraerlos cuando está vacía.

2.2. Main y "extract_operations"

En segundo lugar, tenemos el main thread. En este segmento, se validan y guardan los argumentos introducidos en argv, asignando valores a las variables compartidas *"n_cajeros"*, *"n_trabajadores"*, *"max_cuentas"* y *"buffer_size"*. Tras esto, si el programa no ha sido

interrumpido por la introducción de un argumento inválido, se procede a extraer las operaciones del fichero indicado.

Para la extracción de las operaciones, hemos decidido crear la función *"struct element **extract_operations(char *input_file_path)"*, que recibe el directorio de un fichero a procesar, guarda el número de operaciones a procesar en una variable global, indicada en la cabeza del fichero y retorna una lista de elementos con las operaciones indicadas en cada línea. Esta función, en resumidas cuentas, abre el fichero con directorio *"input_file_path"* y guarda la primera línea del fichero en la variable *"n_operation"*, siempre y cuando esta sea un número entero aceptado. Tras ello, crea un *"struct element*"* por cada línea del fichero y lo guarda en una lista de punteros a elementos *"to_return"* de elementos a retornar; cambiando el valor del atributo del elemento*"char *operation"* al indicado en la línea del fichero, reservando memoria para cada elemento de forma dinámica en la lista y chequeando que efectivamente el número de operaciones es el indicado en *"n_operation"*. Una vez hayamos extraído todas las operaciones en una lista de *"element"s*, las guardamos en la variable global *"list_client_ops"*.

Al finalizar el segmento de extracción de operaciones, pasamos a la inicialización y asignación de otras variables necesarias. Se reserva un byte de memoria para la futura lista de cuentas, cuyo espacio en memoria se irá reasignando al crear cuentas de forma dinámica. En este tramo de código también se inicializan las dos variables condicionales *"no_lleno"* y *"no_vacio"*, el mutex *"mutex"* y la cola circular compartida *"q"*.

Una vez inicializadas y asignadas todas las variables y estructuras de datos necesarias para el funcionamiento del banco, procedemos a crear todos los hilos *"Trabajador"* y *"Cajero"* de forma iterativa, guardando cada uno en su lista correspondiente. Tras esto se procede a recorrer estas listas para esperar a que acaben los hilos, y finalmente se libera la memoria reservada de todas las variables.

2.3. Operaciones bancarias y *"execute_operation"*

Antes de pasar al funcionamiento concurrente de los hilos del programa, es necesario que comprendamos las operaciones bancarias *"crear", "ingresar", "retirar" y "traspasar"*, y la función *"execute_operation"*, la cual será necesaria para la ejecución de las operaciones por parte de los hilos *"Trabajador"*.

Comencemos con las operaciones bancarias. Tomamos la decisión de diseño de crear la estructura de datos *"account"*, con los atributos *"id"* y *"balance"*: números enteros que representan el identificador y el balance de una cuenta para gestionar las operaciones bancarias. Al ejecutar la operación crear, se comprueba que no se haya llegado al número máximo de cuentas indicado cómo parámetro en *argv* y guardado en la variable global *"max_cuentas"*, se inicializa una estructura *"account"* con el id pasado cómo argumento, se reserva memoria dinámicamente para la lista de cuentas *"account_list"* previamente inicializada en el *main*, se aumenta la variable global *"n_accounts"*, y se guarda la cuenta creada en esta lista.

El resto de operaciones, encuentran las cuentas indicadas en sus argumentos a través de la función auxiliar *"find_account"*, que recibe el identificador de una cuenta y retorna la posición en la lista

de cuentas *"account_list"* de la cuenta con ese identificador, o -1 en caso de no encontrarla. Una vez encontradas las cuentas necesarias para las operaciones, las funciones *"ingresar", "retirar" y "traspasar"* modifican los balances de estas aumentando, reduciendo o traspasando el valor indicado en sus argumentos *"balance"* respectivos.

Por otro lado, la función *"execute_operation"*, en primer lugar, recibe un puntero a una estructura de datos *"element"*. A continuación, identifica la operación indicada en la cadena de caracteres *"operation"* de *"element"*. A través de la función auxiliar *"extract_arguments*", extrae, convierte a *int*, y guarda en una lista de enteros los argumentos numéricos de la *"operation"*. Finalmente ejecuta la operación indicada *"crear", "ingresar", "retirar" o "traspasar"* con los argumentos numéricos de la lista de enteros.

Es importante mencionar que, en la ejecución de todas estas funciones, se lleva a cabo una rigurosa validación de los argumentos con los que se las llama, interrumpiendo el programa en caso de un error de formato o valor inválido.

2.4. Cajero y Trabajador

Trabajador y Cajero son los procedimientos que ejecutarán los hilos lanzados en el *main*, y son la esencia de la práctica, puesto que es en estos segmentos de código se maneja la concurrencia del programa.

En primer lugar, el procedimiento *"Cajero"* es el ejecutado por los hilos con el fin de extraer los elementos guardados previamente por *"extract_operations"* en el vector *"list_client_ops"*. En la ejecución de estos hilos, los requisitos de concurrencia a solucionar son las siguientes:

1. Los cajeros deben tener exclusión mutua sobre el acceso a los elementos del vector *"list_client_ops"*, el acceso al entero que marca la posición del vector que se ha de extraer, y su introducción en la cola compartida, si esto no ocurre, podría haber problemas en el orden de introducción de los elementos, repeticiones o se podrían saltar algún elemento del vector. La exclusión mutua debe ser también con los hilos de tipo *Trabajador*, puesto que la ejecución paralela de estas instrucciones supone errores a la hora de ejecutar las operaciones, al existir la posibilidad de modificar datos mientras se están procesando.
2. Los cajeros deben introducir elementos en la cola siempre que la cola no esté llena, en el momento en el que esté llena, se ha de esperar a los hilos consumidores *"Trabajador"* para que liberen huecos en la cola.
3. En el momento en el que un *"cajero"* introduzca un elemento en la cola compartida, se debe señalar a los hilos que la cola ya no está vacía.
4. En el momento en el que un *"cajero"* introduzca el elemento con la última operación, todos los hilos *"cajero"* deben finalizar su ejecución.

La forma en la que hemos organizado los cajeros es mediante una función iterativa cuya condición es que la variable compartida *"int client_numop"* que marca la operación que se quiere introducir sea estrictamente menor que el número de operaciones que se indican en la

cabecera del fichero de operaciones. Llamaremos a esta condición: *condición de finalización de los cajeros.*

El primer requisito lo satisfacemos a través de operaciones de *lock* y *unlock* de un *mutex* que será común a todos los *hilos*, tanto los de *trabajador* cómo los de *cajero*, que garantizará la exclusión mutua de la sección crítica mencionada.

Dentro del tramo mutuamente excluido, resolvemos el segundo requisito a través de un *wait* a una variable condicional *"no_lleno"*, que detiene la ejecución del hilo cuando la cola esté llena y la *condición de finalización de los cajeros* se cumple. El *wait* está dentro de un bucle *while* para comprobar que las condiciones se cumplen también cuando los hilos son liberados de la espera.

El tercer requisito lo completamos mediante un signal de la variable condicional *"no_vacio"* antes de liberar el *mutex*.

Por último, hemos hecho que el cuarto requisito se cumpla a través de una sentencia condicional estratégicamente colocada tras el *lock* del *mutex* y después del *wait*. La condición de esta sentencia es que se cumpla la *condición de finalización de los cajeros.* El primer *thread* que detecte que se ha acabado de introducir los elementos del vector en el bucle, entrará en la condición, señalizará que todos los hilos salgan del *wait* mediante un *broadcast*, y finalizará la ejecución del proceso ligero. El resto de hilos, acabarán su ejecución por el final de la función iterativa, o entrarán también en la sentencia condicional si están bloqueados en el *wait* o en el *mutex.*

En segundo lugar, el procedimiento *"Trabajador"* es el ejecutado por los hilos con el fin de extraer los elementos de la cola compartida, leer sus atributos *"operación"*, y ejecutar la operación junto a los argumentos indicados. En la ejecución de estos hilos, los requisitos de concurrencia a solucionar son las siguientes:

1. Los trabajadores, al igual que los cajeros deben tener exclusión mutua sobre el acceso a los elementos de la cola compartida *"q",* la ejecución del elemento extraído y el acceso al entero que marca el número de la operación que se ha de ejecutar. Si esto no ocurre, podría haber problemas en el orden de ejecución de los elementos o repeticiones. La exclusión mutua debe ser también con los hilos de tipo *Cajero,* puesto que la ejecución paralela de estas instrucciones supone errores a la hora de ejecutar las operaciones, al existir la posibilidad de modificar datos mientras se están procesando.
2. Los trabajadores deben retirar elementos de la cola siempre que no esté vacía, en el momento en el que esté vacía, se ha de esperar a los hilos productores *"Cajero"* para que vuelvan a ocupar huecos en la cola.
3. En el momento en el que un *"trabajador"* consuma un elemento en la cola compartida, se debe señalar a los hilos que la cola ya no está llena.
4. En el momento en el que un *"trabajador"* consuma el elemento con la última operación, todos los hilos *"trabajador"* deben finalizar su ejecución.

Para satisfacer todos estos requisitos, se ha creado algoritmo similar y compatible con el de los hilos *"Cajero"*. Se organiza la función de forma iterativa con una *condición de finalización de*

los trabajadores: que la variable que representa el número de la operación del elemento a ejecutar *"int bank_numop"* sea estrictamente menor que el número de operaciones especificado.

Se garantiza la exclusión mutua a través de un *mutex* que protege cada iteración de la función. Se espera a los productores cuando la cola está vacía a través de un wait dentro de un while, que espera siempre y cuando la cola esté vacía y no se cumpla la *condición de finalización de los trabajadores*. Se señala que la cola ya no está llena, y por ente ya pueden intentar producir los hilos *"Cajero"*, antes de liberar el *mutex*, y por último se señala la finalización de todos los procesos ligeros *"Trabajador"* a travñes de una sentencia condicional estratégicamente colocada tras el *wait*.

Al cumplir todos estos requisitos, aseguramos que el programa *bank.c* siempre sea capaz de ejecutar las operaciones en orden, sin condiciones de carrera, inanición o interbloqueo, creando un sistema bancario multi-hilo simplificado.

2.4 Nota

En esta sección, nos enfocaremos en la validación exhaustiva y el tratamiento de errores en nuestro programa. En la práctica anterior, no abordamos adecuadamente los posibles errores de las funciones que llamamos en el código. Por lo tanto, hemos decidido hacer un esfuerzo serio en el tratamiento exhaustivo de errores en bank.c. Se ha realizado una amplia prueba del código y se han identificado muchos casos de error. Aunque esperamos haber detectado todos, hemos notado que el tratamiento de errores dificulta la legibilidad y comprensión del código para el desarrollador. Por lo tanto, entendemos que es mejor realizar esta tarea al final del proceso de desarrollo.

3. Batería de pruebas

A continuación se presenta una batería de pruebas que recogen los tests ejecutados sobre nuestro programa para comprobar su fiabilidad, valores límite y casos de uso.

bank.c				
Id:	**Datos para introducir:**	**Objetivos de la prueba:**	**Resultados esperados:**	**Resultados obtenidos:**

01	`./bank file.txt 12 12 13 1 178`
	
	`or`
	`./bank file.txt 12`
		Se quiere comprobar que se lanza el mensaje de error correspondiente al introducir un numero distinto a 6 parámetros numéricos.	`[ERROR] Usage: ./bank <nombre_fichero> <num_cajero> <num_trabajadores> <max_cuentas> <tam_buff>`	`[ERROR] Usage: ./bank <nombre_fichero> <num_cajero> <num_trabajadores> <max_cuentas> <tam_buff>`

02	`./bank file.txt 10000 1 150 10` `./bank file.txt 0 1 150 10`	Se quiere comprobar que el número de cajeros debe estar comprendido entre 1 y 5000	`[ERROR] Number of cashiers (cajeros) must be between 1 and 5000`	`[ERROR] Number of cashiers (cajeros) must be between 1 and 5000`
03	`./bank file.txt 1 10000 150 10` `./bank file.txt 1 0 150 10`	Se quiere comprobar que el número de trabajadores debe estar comprendido entre 1 y 5000	`[ERROR] Number of workers (trabajadores) must be between 1 and 5000`	`[ERROR] Number of workers (trabajadores) must be between 1 and 5000`
04	`./bank file.txt 10 10 10001 10` `./bank file.txt 10 10 0 10`	Se quiere comprobar que el número de cuentas debe estar comprendido entre 1 y 10000	`[ERROR] Number of maximum accounts (cuentas) must be between 1 and 10000`	`[ERROR] Number of maximum accounts (cuentas) must be between 1 and 10000`
05	`./bank file.txt 10 10 100 0` `./bank file.txt 10 10 100 20000`	Se quiere comprobar que el tamaño del buffer compartido está comprendido entre 1 y 199999	`[ERROR] Size of queue (buffer) must be between 1 and 20000`	`[ERROR] Size of queue (buffer) must be between 1 and 20000`

06	`./bank documento-inexistente.txt 40 40 40 40`	Se quiere comprobar que se para el programa si se introduce un archivo '.txt' inexistente.	`[FATAL ERROR] File cannot be read`
			`Exiting the program[FATAL ERROR] File cannot be read`
			`Exiting the program`
				`[FATAL ERROR] File cannot be read`
				`Exiting the program[FATAL ERROR] File cannot be read`
				`Exiting the program`

07	`./bank file.txt 40 40 40 40`	Se quiere comprobar que al introducir un archivo sin indicar	`[ERROR]: operation file does not contain`	`[ERROR]: operation file does not contain`

	Contenido de file.txt: CREAR 1 …	el número de operaciones a realizar para la ejecución del programa.	number operations as a header Exiting the program...	number operations as a header Exiting the program...
08	./bank file.txt 40 40 40 40 Contenido de file.txt: 201 CREAR 1 ...	Se quiere comprobar que se levanta un error si el número de operaciones a realizar es superior a #define MAX_NUM_OP 200	[FATAL ERROR] Number of operations cannot be greater than 200 Exiting the program...	[FATAL ERROR] Number of operations cannot be greater than 200 Exiting the program...
09	./bank file.txt 40 40 40 40 Contenido de file.txt: 150 CREAR 1 ...	Se quiere comprobar que se levanta un error cuando el número de operaciones especificado en 'file.txt' es distinto al número real de operaciones, tanto mayor como menor. En este caso, hay 200 operaciones en 'file.txt' y hemos indicado que hay 150 en la primera línea.	FATAL ERROR] Number of operations different from indicated in file (150) Exiting the program...	[FATAL ERROR] Number of operations different from indicated in file (150) Exiting the program...
10	Fichero vacío	Se quiere comprobar que se levanta un error si se introduce un fichero vacío.	[FATAL ERROR] File file.txt is empty. Exiting the program...	[FATAL ERROR] File file.txt is empty. Exiting the program...
11	./bank file.txt 40 40 40 40 Contenido de file.txt: 200	Se quiere comprobar que los argumentos numéricos de las operaciones no	[ERROR]: Maximum argument size: 9 digits	[ERROR]: Maximum argument size: 9 digits

	`CREAR 12345678910`	superan los 9 dígitos. Es decir, los 'id' de las cuentas, las cantidades a ingresar, retirar, traspasar, no pueden ser superiores a 9 dígitos.		

12	`./bank file.txt 10 10 20 5`
	
	`Contenido de file.txt:`
	`CREAR 1`
	`...`
	`CREAR 20`
	`CREAR 21`
	`...`
	`CREAR 50`
		Se quiere comprobar que, habiendo indicado que el número máximo de cuentas es 20, si se quieren crear más cuentas, en este caso, 50, se levanta un error.	`[ERROR]: Reached maximum number of accounts (20)`
			
			`[ERROR]: Cannot create account`
				`[ERROR]: Reached maximum number of accounts (20)`
				
				`[ERROR]: Cannot create account`

13	`./bank file.txt 10 10 50 5`
	
	`Contenido de file.txt:`
	`INGRESAR 1 100`
	`CREAR 2`
		Se quiere comprobar que, si una cuenta no existe, se lance un error indicando la imposibilidad de hacer operaciones con ella.	`[ERROR]: Account 1 does not exist.`
			
			`[ERROR]: In deposit (ingresar) operation`
			
			`[ERROR]: Cannot deposit into account`
				`[ERROR]: Account 1 does not exist.`
				
				`[ERROR]: In deposit (ingresar) operation`
				
				`[ERROR]: Cannot deposit into account`

14	`./bank file.txt 10 10 50 5`
	
	`Contenido de file.txt:`
	`ABC`
	`CREAR 1`
	`...`
		Se quiere comprobar que se levanta un error si se introduce un valor no numérico como indicador del número de cuentas en el fichero.	`[ERROR]: operation file does not contain number operations as a header`
			
			`[FATAL ERROR] extract_operations, cannot validate`
				`[ERROR]: operation file does not contain number operations as a header`
				
				`[FATAL ERROR] extract_operations, cannot validate`

			operation: ABC Exiting the program...	operation: ABC Exiting the program...
15	./bank file.txt 10 10 50 5 Contenido de file.txt: 50 CREAR 1 2 3 INGRESAR 1 100 23 ...	Se quiere comprobar que se levanta un error cuando se introduce un número distinto de argumentos a los designados por cada operación (por ejemplo, en crear solo entra 1, en Ingresar 2, …)	[ERROR]: CREAR 1 2 3 must have 1 arguments [ERROR]: Extracting arguments of CREAR 1 2 3 [ERROR]: INGRESAR 1 100 234 must have 2 arguments [ERROR]: Extracting arguments of INGRESAR 1 100 23	[ERROR]: CREAR 1 2 3 must have 1 arguments [ERROR]: Extracting arguments of CREAR 1 2 3 [ERROR]: INGRESAR 1 100 234 must have 2 arguments [ERROR]: Extracting arguments of INGRESAR 1 100 23
16	./bank file.txt 10 10 50 5 Contenido de file.txt: 50 CREAR 1 INGRESAR 1 100 RETIRAR 1 -40 ...	Se quiere comprobar que no se pueden retirar números negativos de las cuentas, es decir, no se pueden retirar -40€, por ejemplo.	[ERROR]: Cannot substract a negative amount of money [ERROR]: Cannot withdraw from account	[ERROR]: Cannot substract a negative amount of money [ERROR]: Cannot withdraw from account
17	./bank file.txt 10 10 50 5 Contenido de file.txt: 50 CREAR 1 CREAR 2 INGRESAR 1 100 INGRESAR 2 100	Se quiere comprobar que no se pueden traspasar números negativos entre cuentas, es decir, no se pueden traspasar -40€, por ejemplo.	[ERROR]: You cannot transfer negative amounts of money. [ERROR]: Cannot transfer	[ERROR]: You cannot transfer negative amounts of money. [ERROR]: Cannot transfer

	TRASPASAR 1 2 -40 ...		between accounts	between accounts
18	./bank file.txt 10 10 50 5 Contenido de file.txt: 50 CREAR 1 CREAR 2 INGRESAR 1 100 INGRESAR 2 100 TRASPASAR 1 2 200 ...	Se quiere comprobar que no se puede traspasar dinero del que no dispone una cuenta. Es decir, si la cuenta 1 tiene 100€, no puede traspasar más de esa cantidad a cualquier otra cuenta.	[ERROR]: Insufficient balance for transfer [ERROR]: Cannot transfer between accounts	[ERROR]: Insufficient balance for transfer [ERROR]: Cannot transfer between accounts
19	./bank file.txt 10 10 50 5 Contenido de file.txt: 50 HOLA 1 QUE 1 100 TAL 2 ? 2 50	Se quiere comprobar que se levanta un error si se introduce una operación distinta a las permitidas.	ERROR]: Operation HOLA is not valid [ERROR]: Operation QUE is not valid [ERROR]: Operation TAL is not valid [ERROR]: Operation ? is not valid	[ERROR]: Operation HOLA is not valid [ERROR]: Operation QUE is not valid [ERROR]: Operation TAL is not valid [ERROR]: Operation ? is not valid

NOTA: Tanto en *'bank.c'* como en *'queue.c'* se han tratado los errores relativos a los *'mallocs'*, la creación de *'threads'*, tanto de Trabajador como de Cajero, los *'joins*' de los mismos, y los *'reallocs'* también. Todos ellos no se ven reflejados en la tabla de pruebas, dado que, ejecutando la instrucción de *'./bank.c <resto de argumentos>'* o cambiando el contenido de *'file.txt'* no podemos realmente desencadenar los errores relativos a lo anteriormente mencionado, y por tanto, no disponemos de casos de prueba para ellos.

No obstante, durante el desarrollo del código, nos han sido muy útiles para comprobar dónde estaba fallando, además de definir el principal camino de ejecución de este.

Para finalizar esta nota, hay que mencionar que en *'bank.c'* y en *'queue.c'* tratamos los errores para las funciones *'queue_put'* y *'queue_get'*. En el primer caso, se lanza un error si se intentan meter más elementos cuando la cola está llena, en el segundo caso, se lanza un error cuando se intentan sacar elementos de una cola vacía. Hemos intentado forzar estos errores programando 5000 cajeros y 1 trabajador, o al revés, pero para nuestra tranquilidad, la sincronización entre trabajadores y cajeros es efectiva y no levanta un error.

4. Conclusiones

Esta práctica resultó ser muy desafiante ya que tuvimos que realizar varias pruebas para obtener correctamente la concurrencia de los procesos.

Debemos comentar que hemos tenido problemas pasando las pruebas cuando hemos hecho uso de *'close(fd)'*. En un principio, había aleatoriedad en los resultados obtenidos, pasando en ocasiones todas las pruebas, y en otras no. Tras comentarlo en clase, varios compañeros coincidieron en tener el mismo problema. Aún así, actualmente nuestro código cuenta con los *'close(fd)'* correspondientes, pasando todas las pruebas.

En *bank.c*, la mayor dificultad fue la implementación de la manipulación sincronizada de las operaciones. Sumado a ello, durante el desarrollo del código, los *'threads*' trabajadores se quedaban esperando o bloqueados infinitamente. Tuvimos que hacer un análisis exhaustivo, evaluando todos los posibles caminos de ejecución para averiguar los sitios de bloqueo y espera infinita. Finalmente, conseguimos asegurar el trabajo sincronizado entre *threads* (mediante el uso de *mutex* y variables condición) y la terminación de estos cuando las tareas están ya acabadas, evitando así la inanición o la espera infinita.

En definitiva, esta práctica nos ha dotado de un firme entendimiento práctico de la programación multi-hilo, el concepto de concurrencia y el funcionamiento del lenguaje C, y la hemos considerado de gran utilidad para entender el contenido teórico de la asignatura.