Universidad Carlos III
Grado en ingeniería informática
Curso Sistemas Operativos 2023-24
Práctica 3
Curso 2023-24

Fecha: **10/05/2024** - ENTREGA: **3**

GRUPO: **81**

Alumnos: **Mario Ramos Salsón (100495849), Miguel Fidalgo García (100495770) y Víctor Martínez de las Heras (100495829)**

ÍNDICE

1. DESCRIPCIÓN DEL CÓDIGO

1.1 Queue.h

Este archivo incluye dos de las estructuras más importantes para la ejecución del programa. La primera de ellas es la estructura ***"element"***, compuesta por 3 variables. Esta estructura se encarga de almacenar la información que hay dentro del fichero a leer y distribuir dicha información en cada una de las variables. La primera de ellas guarda el ID del producto, la segunda de ellas es un array de caracteres que almacena la operación a realizar, ya sea "PURCHASE" o "SALE" y la última de todas, la cantidad de unidades con las que se va a operar.

La segunda estructura es la estructura ***"queue"***, compuesta en este caso por 4 variables. La primera de ellas es una estructura tipo ***"element"*** que se encarga de almacenar todas las estructuras de este tipo que se inserten en ella. La segunda y tercera variable son la cabecera y la cola, estas nos ayudan a realizar toda la lógica que implementa el buffer circular. Entre estas encontramos operaciones como la inserción o la lectura. La última de las variables es simplemente el tamaño total de la cola. Esta nos ayuda a realizar todas las operaciones modulares para avanzar la cabecera o la cola en caso de una inserción o una lectura, entre otras operaciones.

Al final de dicho archivo, declaramos todas las funciones que nos sirven para darle funcionalidad al buffer circular. Estas son: ***"queue_init",*** que se encarga de inicializar la cola, ***"queue_destroy"***, que libera toda la memoria que ocupa nuestra cola, ***"queue_put",*** que se encarga de insertar información a la cola, ***"queue_get"***, que se encarga de eliminar un elemento de la cola y por último las funciones ***"queue_empty"*** y ***"queue_full"***. Estas se encargan de determinar si la cola está llena o vacía, para así evitar inserciones o lecturas cuando no son válidas.

Este archivo será posteriormente importado tanto en ***"store_manager.c"*** como en ***"queue.c"*** para hacer uso de dichas funciones y estructuras.

1.2 Queue.c

En este archivo, se implementan todas las funciones descritas anteriormente, para su posterior utilización en la realización del código principal.

- **queue_init**: Esta función se encarga de crear la cola, estableciendo por defecto el valor 0 a la cabecera y la cola. Además, inicializa su buffer interno a una estructura tipo ***"element"***, cuya longitud se ha recibido como parámetro a la hora de ejecutar el programa.
- **queue_full**: Esta función se encarga de comprobar si la cola está llena sumándole 1 a la cabecera y haciendo el módulo de esta suma con el tamaño total del buffer. Si esto coincide con el valor que tiene la cola, está llena. Esta función auxiliar se usa en ***"queue_put"*** para evitar insertar un elemento si la cola está llena.
- **queue_put**: Esta función se encarga de insertar un elemento al buffer circular y de incrementar la posición de la cabecera en 1. Previamente a toda esta operación, se hace uso de ***"queue full"*** para evitar insertar algo si la cola está llena.
- **queue_empty**: Esta función se encarga de comprobar si la cola está vacía igualando los valores de la cabecera y la cola. En caso de que sean iguales, la cola está vacía y no se podrán leer operaciones de ella. Está función auxiliar se utiliza en ***"queue_get"*** para que a la hora de leer compruebe si está vacía o no y evitar lecturas no válidas.
- ***queue_get***: Esta función se encarga de leer elementos de la cola y devolverlos para su posterior análisis y así realizar una compra o una venta. Previo a toda esta operación se comprueba si está o no vacía con la función ***"queue_empty"*** para evitar intentar leer de una cola que no tiene elementos.
- **queue_destroy**: Está función simplemente elimina todos los recursos que la cola ha ocupado en memoria.

Todas las operaciones que se realizan en este fichero tanto para incrementar la cola o la cabecera como para comprobar si la cola está vacía o llena, se realizan mediante operaciones modulares. Estas consisten en operar con el módulo del tamaño total del buffer circular, para que así los únicos valores que puedan tomar la cabecera y la cola estén en un rango entre 0 y el tamaño máximo de la cola - 1.

1.3 Store_manager.c

Este archivo se encarga de utilizar todas las funciones implementadas anteriormente, junto con otras propias para así llevar a cabo toda la lógica de la práctica e implementar bien el problema de los consumidores y productores.

Al principio del archivo, se declaran las variables necesarias para llevar un conteo de los beneficios de la empresa junto con 3 arrays encargados de almacenar el stock de los productos que se tienen junto con sus precios de compra y venta. Posteriormente se declaran dos estructuras que son utilizadas por los consumidores y los productores. La primera de ellas ***"argProd"***, se encarga de almacenar todas las variables necesarias que utilizan los productores para realizar su trabajo. Estas variables son, el buffer circular, un buffer interno que almacena estructuras tipo ***"element"*** y el tamaño que tiene dicho buffer interno. La segunda estructura es ***"argCon"***, que se encarga de almacenar las variables que los consumidores necesitan para realizar su trabajo. Estas son el buffer circular, el número de elementos que van a tener que leer, junto con el beneficio y stock parcial que cada hilo va a procesar.

Además, se crea una estructura auxiliar ***"retCon"***, que se encarga de guardar el stock y beneficio parcial para que cuando el hilo acabe, devuelva estos valores previamente calculados.

A continuación, se declaran dos funciones auxiliares. La primera de ellas ***"maxOperaciones"***, se encarga de leer la primera línea del archivo que le pasamos por parámetro para así determinar el máximo número de operaciones que se van a realizar. Cabe destacar que si se indican menos operaciones que el número real de líneas que contiene el archivo, solo se realizarán las operaciones indicadas en la primera línea. La segunda función auxiliar ***"guardarOp"***, se encarga de guardar en memoría las operaciones que se van a realizar durante la ejecución del programa. Para ello recibe por parámetro 3 variables. La primera de ellas es el nombre del fichero a leer, la segunda de ellas, un array que almacena estructuras tipo ***"element"*** y la tercera, el total de estructuras que tiene que guardar. Para guardar dichas estructuras, se hace uso de la función ***"fscanf"*** de la biblioteca ***"stdio.h"*** dentro de un bucle while, que comprueba si leen 3 elementos por fila, y que no se supera el número máximo de filas a leer en cada iteración. Una vez que se han leído todas las filas, la función termina.

En la siguiente parte del código, se implementan las funciones de los productores y consumidores ***("funProductores")*** y ***("funConsumidores")*** respectivamente. La de los productores, se encarga de insertar elementos al buffer circular. Aquí se empieza a hacer uso de las variables condicionales declaradas al principio del archivo. Cada productor tiene un número determinado de operaciones a insertar, para ello se hace uso de un bucle for, controlado por un while que determina si la cola está llena o no para evitar insertar cuando no le corresponde. Esta comprobación se realiza mediante la función ***"queue_full"*** explicada anteriormente. Si se intenta insertar cuando la cola está llena, se activa la variable condicional ***"wait"*** sobre la condición ***"vacío"*** y se desbloquea el ***"mutex"***. Esta variable condicional, se encarga de bloquear el hilo hasta que se notifique un cambio en ***"vacío"*** para así dar paso a dicho hilo que se había quedado bloqueado esperando a insertar. Una vez que todos los hilos productores han insertado sus operaciones correspondientes mediante la función ***"queue_put"***, se recogen. Cabe destacar que cada vez que un hilo productor inserta, envía una señal a la variable condicional ***"lleno"***, para que en caso de que un consumidor estuviese esperando a que se insertase un elemento en el buffer, este pueda leer.

La segunda función, se encarga de implementar toda la lógica detrás de los consumidores. Su desarrollo es muy similar al de los productores, un bucle for seguido de un bucle while que controla mediante la función ***"queue_empty"*** si la cola tiene algún elemento o no para permitir al hilo leer. En caso de que esta función indique que está vacía, se bloqueará al hilo haciendo uso de una variable condicional ***"lleno"*** y esperará hasta que se notifique un cambio en la misma para así poder realizar su lectura. Una vez que se realice una lectura exitosa mediante la función ***"queue_get"***, se analiza si lo que se ha leído conlleva realizar una compra o una venta. En función de qué operación haya que realizar, se suma o resta el beneficio parcial y se establece un precio u otro dependiendo del producto. Para terminar la función, se notifica un cambio en la variable condicional ***"vacío"*** para que en caso de que un productor esté esperando a insertar, pueda hacerlo, y se devuelve una estructura ***"retCon"*** que guarda el stock y el beneficio parcial calculado. Esto sirve para que después el programa main pueda actualizar el beneficio y el stock total.

A continuación, empezamos con la función ***"main"*** del programa, la cual controla todas las variables, toda la creación de los hilos y el reparto de trabajo a cada uno de ellos. Al principio del todo encontramos las comprobaciones básicas para garantizar que el programa tiene una buena base, como que si se han

pasado los suficientes parámetros o dichos parámetros se encuentran entre sus valores mínimos.

El siguiente paso es obtener el total de operaciones que se van a realizar haciendo uso de la función ***"maxOperaciones"***, que previamente hemos creado. Este valor lo guardamos en una variable ya que nos va a ayudar a gestionar todo el reparto a lo largo del programa. A continuación, creamos la variable ***"operaciones"*** que almacena datos de tipo ***"element"***. Esta se encarga de guardar todas las operaciones que se van a realizar para posteriormente distribuirlas entre los productores equitativamente. Para llenar esta matriz hacemos uso de la función explicada previamente ***"guardarOp"***, la cual recibe por parámetro el nombre del fichero que tiene que leer, dicha estructura ***"operaciones"*** y el total de elementos a guardar.

Seguido de esto viene la inicialización de los hilos consumidores y productores. Para ello, se reserva espacio de memoria suficiente para cada uno de ellos en función de lo que nos hayan especificado por parámetro al iniciar el programa. También se inicializan las variables condicionales, el ***"mutex"*** que controla todo el paralelismo del programa y la cola que va a almacenar toda la información.

Lo siguiente es la creación y el paso de parámetros a los hilos productores. La primera comprobación que se hace es si el número total de operaciones a realizar entre el número de productores es exacto. Esto juega un papel muy importante ya que, en función de esto, un hilo tendrá más carga de trabajo, o todos tendrán la misma. Posteriormente se reserva memoria para almacenar todos los argumentos de los hilos productores y se procede a su reparto de manera equitativa y ordenada. Para ello hacemos uso de un bucle for donde el primer paso es asignarle a cada hilo el buffer circular en el que van a tener que trabajar y el tamaño de su buffer interno en función del reparto que les toque, haciendo una comprobación extra por si toca crear el último hilo y este además tiene que llevar más carga de trabajo. Después, se itera sobre unos contadores que acceden a la estructura ***"operaciones"*** e introducen las operaciones correspondientes a cada hilo dentro de su buffer propio. Una vez que todo este reparto se ha hecho correctamente, se crea el hilo pasándole por parámetro el propio hilo, la función que tiene que ejecutar ***("funProductores")*** y sus argumentos correspondientes.

A continuación, se crean los hilos consumidores. Estos siguen una creación muy similar a la de los productores. Primero se comprueba si el número de operaciones a realizar entre el número de hilos consumidores es exacto, para así distribuir su carga de trabajo de una manera justa. El siguiente paso es su creación, donde al igual que a los productores, lo primero que se hace es asignarles el buffer circular en el cual van a tener que trabajar y después, se les otorga un contador de operaciones que van a tener que realizar. Cabe destacar que no se le asigna las operaciones que van a tener que ejecutar, si no que se les establece un número de operaciones a realizar, haciendo después estos las operaciones que justo les corresponda y nunca superando su máximo. Gracias a esto, se hace un buen uso del paralelismo ya que cada hilo puede ejecutar una operación diferente cada vez que se realice el programa. Una vez hecho bien el conteo de operaciones a realizar, se pasa a la creación. Esta recibe por parámetros el propio hilo, la función que tienen que realizar ***("funConsumidores")*** y sus argumentos propios.

Una vez que ya están lanzados todos los hilos productores y consumidores, se espera a que terminen para recogerlos, actualizar los precios y el stock, y liberar su espacio en memoria con ***"free()"***. Además, se destruyen todas las variables condiciones junto con el ***"mutex"*** y para finalizar, se imprime el resultado total del programa para verificar que se ha calculado correctamente.

2. BATERÍA DE PRUEBAS

Para la descripción de las pruebas usaremos el siguiente formato:

COMANDO EJECUTADO	DESCRIPCIÓN DE PRUEBA
RESULTADO ESPERADO	RESULTADO OBTENIDO

Además, hacemos uso de un fichero ***"file.txt"*** que tiene la siguiente estructura para realizar todas las pruebas de ejecución:

50
1 PURCHASE 100
2 PURCHASE 55
3 PURCHASE 30
4 PURCHASE 20
5 PURCHASE 10
1 SALE 5
1 SALE 10
3 SALE 2
4 SALE 1
4 SALE 5
5 SALE 3

2 SALE 15	1 SALE 5
1 SALE 22	1 SALE 10
3 SALE 10	3 SALE 2
5 SALE 5	4 SALE 1
1 SALE 13	4 SALE 5
2 SALE 7	5 SALE 3
3 SALE 2	2 SALE 15
1 SALE 33	1 SALE 22
2 SALE 1	3 SALE 10
5 SALE 1	5 SALE 5
4 SALE 5	1 SALE 13
3 SALE 6	2 SALE 7
2 SALE 2	3 SALE 2
1 SALE 7	1 SALE 33
1 PURCHASE 100	2 SALE 1
2 PURCHASE 55	5 SALE 1
3 PURCHASE 30	4 SALE 5
4 PURCHASE 20	3 SALE 6
5 PURCHASE 10	2 SALE 2
	1 SALE 7

./store_manager	Se llama al programa sin ningún parámetro
El número de argumentos no es correcto	El número de argumentos no es correcto

./store_manager file.txt	Se llama al programa solo con el archivo
El número de argumentos no es correcto	El número de argumentos no es correcto

./store_manager file.txt 1	Se llama al programa con el archivo y los productores

El número de argumentos no es correcto	El número de argumentos no es correcto

./store_manager file.txt 1 1	Se llama al programa con el archivo, los productores y los consumidores
El número de argumentos no es correcto	El número de argumentos no es correcto

./store_manager file.txt 1 1 1 1	Se llama al programa con argumentos de más
El número de argumentos no es correcto	El número de argumentos no es correcto

./store_manager file.txt 0 0 10	Número de argumentos correctos pero el valor de los productores o consumidores no es correcto ya que tiene que ser mayor que 0
Tiene que haber más de un productor y consumidor	Tiene que haber más de un productor y consumidor

./store_manager file.txt 5 2 10	Se realiza una ejecución normal del programa
Total: 120 euros Stock: Product 1: 20 Product 2: 60 Product 3: 20 Product 4: 18 Product 5: 2	Total: 120 euros Stock: Product 1: 20 Product 2: 60 Product 3: 20 Product 4: 18 Product 5: 2

./store_manager file.txt 3 1 5	Se realiza una operación sobre el mismo fichero pero con diferente tamaño de buffer y número de productores y consumidores para ver que no afecta
Total: 120 euros Stock: Product 1: 20 Product 2: 60 Product 3: 20 Product 4: 18 Product 5: 2	Total: 120 euros Stock: Product 1: 20 Product 2: 60 Product 3: 20 Product 4: 18 Product 5: 2

./store_manager file.txt 5 2 10	Se modifica la primera línea del fichero ***"file.txt"*** y se cambia por un 200
Total: 120 euros Stock: Product 1: 20 Product 2: 60 Product 3: 20 Product 4: 18 Product 5: 2	Total: 120 euros Stock: Product 1: 20 Product 2: 60 Product 3: 20 Product 4: 18 Product 5: 2

./store_manager file.txt 5 2 10	Se modifica la primera línea del fichero ***"file.txt"*** y se cambia por un 49. Esto debería darnos 21 euros menos de beneficio, ya que no se ejecuta la línea "1 SALE 7". Además el stock restante tiene 7 unidades más ya que no se han vendido
Total: 99 euros Stock: Product 1: 27 Product 2: 60 Product 3: 20 Product 4: 18 Product 5: 2	Total: 99 euros Stock: Product 1: 27 Product 2: 60 Product 3: 20 Product 4: 18 Product 5: 2

3. CONCLUSIONES

Cabe destacar que no hemos encontrado ningún problema en el desarrollo de la práctica y que esta ha ido bien encaminada desde un principio. Consideramos además, que ha sido bastante asequible e incluso nos ha resultado la más sencilla de las 3 realizadas hasta la fecha. Creemos que esto también se debe a que nuestros conocimientos acerca de C en general han aumentado a lo largo del curso y a la ayuda proporcionada por los materiales académicos. Además opinamos que ha sido muy dinámica y productiva para ayudarnos a entender en gran medida el manejo del paralelismo y las variables condicionales.