PRÁCTICA 3: PROGRAMACIÓN MULTI-HILO

SISTEMAS OPERATIVOS

PABLO AMOR MOLINA	100495855
SERGIO VERDE LLORENTE	100495899
HUGO CUEVAS ROMERA	100495962

10 DE MAYO DE 2024

ÍNDICE

DESCRIPCIÓN DEL CÓDIGO

A continuación, se presenta una descripción de código fuente mediante el cual se ha llevado a cabo la implementación de la instrucción store_manager, así como se ha pedido en el guion de la práctica.

Carga del fichero

Según lo especificado en el enunciado del trabajo, uno de los argumentos de la instrucción de store_manager es la dirección de un fichero que habrá que leer y procesar, pues en el mismo se hayan las instrucciones que deberán ser ejecutadas por los hilos.

Para la carga de dicho archivo, en primer lugar, se ejecuta la función fopen, que guarda como variable FILE el archivo especificado como parámetro. Tras esto, se realiza un fscanf sobre la primera línea del fichero para comprobar que esta sea un número entero y almacenarlo en la variable ops. Tras esto, se reserva memoria mediante un malloc (almacén) de tamaño ops*tamaño_de_element donde element es cada una de las instrucciones a procesar por los hilos.

Una vez reservada la memoria, se ejecuta un bucle en el que, mediante otro fscanf, se guarda en el almacén un entero (id), un string (op), y otro entero (units). En caso de que alguna línea del fichero no respete el formato establecido, salta un error indicando la ubicación aproximada del mismo (que se corresponde con enésima iteración del bucle). Además, también se comprueba que la operación almacenada sea “PURCHASE” o “SALE” y guarda un 0 o un 1, correspondientemente, en el atributo op de element, pues este debe ser un entero.

Operaciones intermedias necesarias

Una vez se han cargado en memoria todas las instrucciones del fichero, es momento de repartir y asignar las tareas que deberá llevar a cabo cada hilo productor y consumidor. Para ello, para cada tipo de hilo se calculará una división entera del número de operaciones a realizar (ops) entre el número de hilos de ese tipo especificado como parámetro (reparto_prod o reparto_con). Además, habrá que tener en cuenta que puede que la división entera deje un resto, por lo que este también se almacenará para ser tratado y repartido posteriormente (resto_prod o resto_con).

Tras esto, se reservará memoria tanto para los dos arrays de hilos como para sendas estructuras de argumentos que requerirán. Además, se llevan a cabo los init tanto del mutex como de las condiciones, que controlarán la concurrencia de los hilos y de la cola circular sobre la que se dará dicha concurrencia.

Con todo esto, se procede a crear los hilos.

Creación de los hilos

En esta sección se distinguirán los procedimientos de creación de cada uno de los tipos de hilos, aunque se enfocará principalmente en la implementación sobre los hilos productores, la más compleja, puesto que toda la lógica de los consumidores ya se puede ver reflejada en estos.

En primer lugar, es necesario contextualizar la estructura que designa el parámetro de hilo de los productores y los consumidores: En el caso de los productores, esta contendrá un índice inicial (int), un número de instrucciones (int), el puntero al almacén (struct element *) y el puntero a la cola (struct queue *, ver página 7). En el caso de los consumidores, la estructura únicamente requerirá el número de instrucciones y el puntero a la cola.

La creación de los hilos productores se lleva a cabo en un bucle con tantas iteraciones como número de productores se hayan especificado en la instrucción. Para el parámetro que designa el índice del buffer, se sumará al índice de la iteración anterior el número de operaciones de la iteración anterior, a excepción de en la primera, que el índice será 0.

Por otro lado, para el número de operaciones, en caso de que no haya que tener en cuenta el resto, simplemente se tomará reparto_prod, calculado en la anterior sección. De lo contrario, se le sumará uno a este valor y se incrementará en 1 el contador (cont_resto_prod) que controla cuántos de los hilos deben tener una operación adicional (siendo este número resto_prod).

Con todo esto, se crea el hilo llamando a la función productor y utilizando como parámetro de hilo la estructura que acabamos de modificar. En el caso de los consumidores, al no trabajar con lo cargado en memoria, únicamente les será necesarios calcular el número de operaciones a realizar por cada uno, de forma análoga a como se ha implementado en los productores.

Función Productor

La función productor se ha implementado siguiendo el modelo del problema Productor-Consumidor visto en clase, solo que adaptando la ejecución a nuestra cola circular. Además, donde adquiere complejidad, como se ha comentado antes, es en lo relacionado con la gestión de índices, que se extrae de la estructura introducida como parámetro del hilo.

El número de operaciones se procesa en un bucle con tantas iteraciones como tareas que tiene que hacer ese hilo (reparto_prod). Sin embargo, la variable de dicho bucle en la mayoría de casos no actuará sobre el rango [0, # de operaciones], sino que lo hará sobre [índice inicial, índice inicial + número de instrucciones]. Con esto, la variable que maneja el bucle contendrá directamente el valor del índice del almacén en memoria al que quedemos acceder para cada iteración.

Consumidor

Al igual que la función productor, la función Consumidor se ha implementado tomando como referencia la obtenida de las diapositivas de clase, pero aplicándola a nuestro buffer circular. Las operaciones se ejecutan en un bucle con tantas operaciones como reparto_con haya pasado como parámetro.

Además, para devolver el beneficio y los stocks parciales calculados por el hilo, se crea una nueva estructura de nombre acum con ambos datos como atributos. Antes de desbloquear el mutex del consumidor, se realizan los cálculos del beneficio y los stocks parciales:

- Si la operación es SALE se resta al stock acumulado de ese producto en concreto las unidades vendidas, y se suma al beneficio acumulado el precio de ese producto multiplicado por las unidades.

- Si la operación es PURCHASE, el proceso es el mismo pero invirtiendo los operadores.

Cabe mencionar que, para acceder a un producto en concreto, póngase de ejemplo el id=5, hay que acceder al índice de valor inmediatamente anterior en el array donde están almacenados (fuera del main), en este caso, 4. Esto se realiza mediante la operación:

Productos[dato->id-1]

Una vez calculado todo, finaliza el hilo y se devuelve la estructura creada.

Recoger los hilos y cálculos finales

En la última parte del main, se procede a recoger los hilos tras su ejecución de manera concurrente. Para el caso de los hilos Productores, esto es una tarea bastante simple, pues basta con hacer un bucle con tantas iteraciones como hilos se hayan creado para recogerlos.

En el caso de los consumidores, es necesario además calcular el beneficio total y los stocks finales de cada producto a partir de lo que ha devuelto cada hilo. Esto se consigue creando otra estructura acum en la que se recogerá la dirección de memoria de la estructura que devuelve cada hilo consumidor. Se sumará a la variable profit y a los stocks (por medio de un bucle de 5 iteraciones) los valores almacenados en esa estructura, y por último se hará el free, todavía dentro del bucle principal, para cada uno de los malloc mediante los cuales se creó en consumidor la estructura acum.

Una vez recogidos ya todos los hilos, se hacen los prints que componen la salida de store_manager y se ejecutan todos los destroy y free necesarios.

Gestión de errores

Además de lo pedido expresamente en el enunciado de la práctica, se ha incluido pruebas de control para los siguientes casos de errores:

- Número de argumentos incorrecto.
- Argumentos no numéricos donde no corresponde.
- Tamaño mínimo de buffer = 1.
- Error al abrir el fichero introducido como argumento (por ejemplo, por nombre incorrecto).
- La primera línea del archivo no es un entero.

Para los siguientes errores, relativos al formato del fichero, se muestra además la línea en torno a la que se sitúa el error detectado.

- Una línea del fichero no sigue el formato: “id op units”.
    - Este error también controla que el número de operaciones no sea mayor al número de líneas de instrucciones, puesto que cuando se llegue a la línea n+1 detectará que, efectivamente, no cumple el formato anterior.
- El id de producto de una de las operaciones es incorrecto (id > 5 || id <1).
- La operación de cierta línea no es ni PURCHASE ni SALE.

Por último, cabe mencionar el trabajo de depuración que se tuvo que hacer al probar la instrucción en el entorno Guernika, puesto que, en un inicio, todas las variables mostradas por pantalla tenían por valor 0.

Tras la depuración, se concluyó con que el problema se ubicaba en la línea siguiente:

```
if (fscanf(infile, "%d %s %d", &almacen[i].product_id, tipo_op, &almacen[i].units) < 3)
```

En ella, por alguna razón que desconocemos, la variable ops, donde se guardaban el número de instrucciones a procesar, pasaba a tomar el valor 0. Esto alteraba completamente la ejecución de la instrucción.

Para solventarlo, simplemente guardamos el valor real de ops antes de acceder al bucle donde se encuentra la anterior línea en la variable error_guernika. Posteriormente, lo restauramos de nuevo en la variable ops.

Cola circular

Para el procesamiento de forma concurrente de las operaciones del fichero, es necesario la implementación de una estructura auxiliar para almacenar las instrucciones. En el caso de nuestra implementación, esta estructura es una cola circular FIFO, como así se nos ha indicado. Los atributos de esta cola son su cabecera, su cola y el tamaño total de la cola.

Esta incluye los métodos básicos necesarios para su correcta ejecución:

- Init: Reserva memoria e inicializa la cola.
- Isfull: Devuelve si la cabecera está en la posición por defecto (vacía).
- IsEmpty: Devuelve si la siguiente posición a la cola es la cabeza (llena).
- Get: Extrae un elemento de la cabecera y la avanza.
- Put: Avanza la cola e inserta un elemento en ella.
- Destroy: Libera la memoria reservada por la cola.

Batería de Pruebas

Prueba	Descripción	Salida	Salida esperada
./store_manager file.txt 2 2 8	Caso genérico con dos productores, dos consumidores y 8 de tamaño del buffer. Se ejecutan 24 operaciones.	Total: 60 euros Stock: Product 1: 17 Product 2: 30 Product 3: 10 Product 4: 9 Product 5: 1	Total: 60 euros Stock: Product 1: 17 Product 2: 30 Product 3: 10 Product 4: 9 Product 5: 1
./store_manager file.txt 2 2 8 1	Error en el número de argumentos de ejecución (arg != 5).	ERROR: se necesitan 4 argumentos para ejecutar la instrucción	ERROR: se necesitan 4 argumentos para ejecutar la instrucción
./store_manager file.txt a 2 8 ./store_manager file.txt 2 b 8 ./store_manager file.txt 2 2 e	Error por no insertar un número como argumento del número de productores, consumidores o tamaño de buffer.	ERROR: La estructura para ejecutar la instrucción es: ./store manager <file name><num producers><num consumers><buff size>	ERROR: La estructura para ejecutar la instrucción es: ./store manager <file name><num producers><num consumers><buff size>
./store_manager file.txt 2 2 0	Tamaño de buffer incorrecto (x =< 0).	ERROR: El tamaño del buffer debe ser mayor que 0	ERROR: El tamaño del buffer debe ser mayor que 0
./store_manager store.txt 2 2 8	El archivo de texto introducido no existe.	Error al abrir el archivo: No such file or directory	Error al abrir el archivo: No such file or directory
./store_manager file.txt 2 2 8	La primera línea del archivo de operaciones no es un número.	ERROR: La primera línea del archivo no es un entero.	ERROR: La primera línea del archivo no es un entero.
./store_manager file.txt 2 2 8	Error al reservar memoria para las operaciones, debido a que el número de ops es menor o igual a -1.	Error al asignar memoria: Cannot allocate memory	Error al asignar memoria: Cannot allocate memory
./store_manager file.txt 2 2 8	Error en el formato de las operaciones del archivo de operaciones. “Línea 8 : 1 SALEs 10”	ERROR: Operación no identificada en la línea 7	ERROR: Formato de fichero incorrecto en torno la línea 8

./store_manager file.txt 2 2 8	Error en el formato de las cantidades del archivo de operaciones. “Línea 8 : 1 SALE a”	ERROR: Formato de fichero incorrecto en torno la línea 8	ERROR: Formato de fichero incorrecto en torno la línea 8
./store_manager file.txt 2 2 8	Error en el formato de los productos del archivo de operaciones. “Línea 8 : a SALE 10”	ERROR: Formato de fichero incorrecto en torno la línea 8	ERROR: Formato de fichero incorrecto en torno la línea 8
./store_manager file.txt 2 2 8	Error en el formato de los productos del archivo de operaciones. “Línea 8 : 7 SALE 10”	ERROR: Producto no identificado en la línea 7	ERROR: Producto no identificado en la línea 7
./store_manager file.txt 5 2 8	Caso en el que probamos el reparto de operaciones entre los productores con resto. En este caso son 5 productores y 24 operaciones, dando como resto 4. Un productor hará 4 operaciones y los otros harán 5.	TAREAS POR PRODUCTOR: 4 RESTO: 4 Total: 39 euros Stock: Product 1: 17 Product 2: 30 Product 3: 10 Product 4: 9 Product 5: 1	TAREAS POR PRODUCTOR: 4 RESTO: 4 Total: 39 euros Stock: Product 1: 17 Product 2: 30 Product 3: 10 Product 4: 9 Product 5: 1
./store_manager file.txt 2 5 8	Caso en el que probamos el reparto de operaciones entre los consumidores con resto. Mismo reparto que en el caso anterior.	TAREAS POR CONSUMIDOR: 4 RESTO: 4 Total: 39 euros Stock: Product 1: 17 Product 2: 30 Product 3: 10 Product 4: 9 Product 5: 1	TAREAS POR CONSUMIDOR: 4 RESTO: 4 Total: 39 euros Stock: Product 1: 17 Product 2: 30 Product 3: 10 Product 4: 9 Product 5: 1
./store_manager file.txt 9 9 8	Caso en el que hay mayor número de hilos productores y consumidores que de operaciones.	Total: -2410 euros Stock: Product 1: 95 Product 2: 55 Product 3: 30 Product 4: 20 Product 5: 10	Total: -2410 euros Stock: Product 1: 95 Product 2: 55 Product 3: 30 Product 4: 20 Product 5: 10

Conclusión

Esta práctica sobre la programación multi-hilo ha sido muy beneficiosa a la hora de comprender la gestión de procesos individuales y de ejecución simultánea y concurrente, además de aplicar todos los conocimientos aprendidos en anteriores prácticas. En este caso, se nos solicitaba crear el diseño de un programa de gestor de beneficios y stock de una tienda, en la cuál hemos tenido que aplicar, además de otros servicios de POSIX, procesos concurrentes (ejecutados en hilos o *threads*) de tipo productor y consumidor que nos ayudasen a obtener correctamente los resultados requeridos.

En cuanto a los problemas que han ido surgiendo a lo largo de la práctica, podemos destacar algunos relacionados con el diseño y codificación del programa solicitado. Principalmente, la mayor complicación que hemos encontrado realizando la práctica ha sido debido a un error nuestro con la gestión del mutex. Al desbloquearlo antes de tiempo, teníamos cambios en las distintas ejecuciones de la instrucción, y, a pesar de ser conscientes de la causa del problema, no conseguíamos dar con la solución. Finalmente, solo bastó con restrasar en el código el desbloqueo del mutex, como ya se ha mencionado.

Otro problema destacable fue por el mencionado en la descripción del código relativo a la ejecución en Guernika (ver Gestión de errores). Este nos puso en una situación complicada debido a que lo reconocimos a un día de la entrega del trabajo, por lo que tuvimos que proceder a depurar el código de manera excepcional hasta encontrar el error.

Finalmente, la opinión general del grupo coincide en que esta práctica sobre la programación multihilo ha facilitado mucho el estudio a fondo de todos los servicios que ofrece el sistema POSIX para la gestión de procesos concurrentes. La práctica es de gran interés para todo aquel que necesite comprender correctamente los procesos concurrentes debido a que simulamos una situación de la vida (administración de una tienda) y no se requieren demasiados conocimientos previos acerca de la programación de estos mismos debido a la detallada explicación del enunciado. Todo esto hace que su realización sea mucho más amena y que su aprehensión sea más eficiente para el alumno.