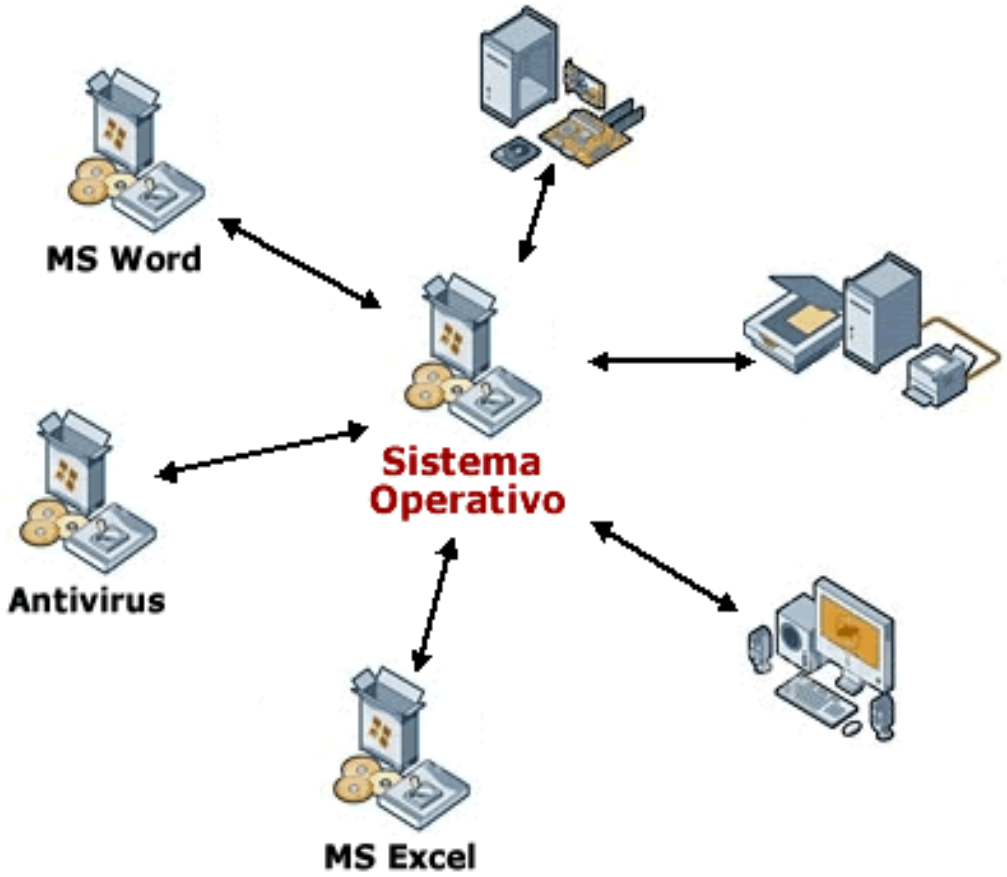
uc3m | Universidad Carlos III de Madrid

Alumno/a:	**Juan Marín Olloqui**	**NIA:**	**100475063**
Alumno/a:	**David Sierra Fernández**	**NIA:**	**100471913**
Alumno/a:	**Paulo Álvarez Da Costa**	**NIA:**	**100475757**



Titulación: GRADO INGENIERÍA INFORMÁTICA Y ADE
Año Académico: 2023/24 – 2º cuatrimestre
Asignatura: Sistemas Operativos (2º curso)
Memoria de Prácticas 3: Programación multi - hilo

Índice de Contenidos

1. Introducción

Está práctica nos ha permitido adentrarnos en el mundo de la gestión de procesos mediante los servicios POSIX. Se explorarán las funcionalidades proporcionadas por la API de hilos POSIX, como pthread_create, pthread_join, pthread_exit, así como los mecanismos de sincronización como mutex y variables condicionales.

El objetivo principal de esta práctica es diseñar y codificar un programa en lenguaje C, ejecutado sobre el sistema operativo UNIX/Linux, que actúe como gestor de beneficios y stock para una tienda. Para lograr este propósito, se deberá implementar un sistema multi-hilo concurrente que calcule el beneficio y el stock de la tienda, basándose en un archivo de entrada con un formato específico que contiene información sobre las operaciones realizadas.

El proceso principal del programa será responsable de varias tareas clave: leer los argumentos de entrada, cargar los datos del archivo en memoria, distribuir equitativamente la carga del archivo entre los hilos productores, lanzar los productores y consumidores, esperar la finalización de todos los hilos y mostrar el beneficio y el stock de cada producto al finalizar las operaciones.

Los hilos productores, por su parte, deberán obtener los datos del archivo correspondientes a sus operaciones asignadas y agregarlos a una cola circular compartida de forma concurrente, sin bloquear ni forzar un orden entre ellos. Mientras que los hilos consumidores deberán extraer elementos de la cola, calcular el beneficio y el stock, y devolver los resultados parciales al proceso principal.

La implementación de estos hilos requerirá el uso adecuado de mutex y variables condicionales para garantizar la sincronización y la exclusión mutua entre los hilos. Además, se deberá desarrollar una cola circular compartida con funciones específicas para insertar y extraer elementos de manera segura y eficiente.

En resumen, esta práctica proporciona una oportunidad para explorar conceptos fundamentales de la programación multi-hilo, la sincronización de procesos y la gestión de recursos compartidos, todo ello mientras se aborda un problema práctico de gestión de tiendas.

2. Descripción del código

2.1 Store_manager

El código simula un sistema de gestión de almacén utilizando el patrón de diseño de productor-consumidor con hilos y sincronización.

El programa comienza con la inclusión de las bibliotecas necesarias y la definición de constantes y estructuras. La constante `NUM_PRODUCTS` define el número de productos diferentes en el almacén. Las estructuras `operation`, `producer_arg`, y `consumer_result` se utilizan para representar las operaciones de compra y venta, los argumentos pasados a los hilos productores y los resultados de los hilos consumidores, respectivamente. Además, se inicializan variables globales para la cola de operaciones y las variables de sincronización como mutex y variables de condición.

La función `load_operations` se encarga de leer un archivo que contiene una lista de operaciones. Cada operación tiene un identificador de producto, un tipo de operación (compra o venta) y la cantidad de unidades involucradas. Esta función almacena las operaciones en un arreglo dinámico y devuelve un puntero a este arreglo.

Los hilos productores (`producer`) toman segmentos del arreglo de operaciones y los colocan en la cola (`buffer`). Utilizan un mutex para asegurarse de que solo un hilo acceda a la cola a la vez y esperan usando una variable de condición (`can_produce`) si la cola está llena.

Los hilos consumidores (`consumer`) leen y procesan las operaciones de la cola. Para cada operación, actualizan el stock y el beneficio acumulado. La memoria para el stock se asigna usando `calloc`, lo que garantiza que todos los valores se inicialicen en cero. Si `calloc` falla, se libera la memoria asignada y se termina el hilo.

La función `process_task` ajusta el stock y el beneficio basándose en el tipo de operación. Si es una compra, aumenta el stock y disminuye el beneficio según el precio de compra. Si es una venta, disminuye el stock y aumenta el beneficio según el precio de venta.

La función `main` es el punto de entrada del programa. Verifica los argumentos de entrada, carga las operaciones del archivo y crea los hilos productores y consumidores. Luego espera a que todos los hilos terminen su ejecución. Finalmente, calcula y muestra el beneficio total y el stock de productos, verificando que no haya inconsistencias como stock negativo.

Al final del programa, se liberan todos los recursos utilizados, incluyendo la destrucción de las variables de condición y el mutex, y se termina la ejecución del programa.

Este programa es un ejemplo de cómo se pueden utilizar hilos y mecanismos de sincronización en C para manejar operaciones concurrentes en un entorno de productor-consumidor. Es importante destacar que el uso correcto de la sincronización es crucial para evitar condiciones de carrera y asegurar la consistencia de los datos.

2.2 Queue.c

El archivo `queue.c` proporciona la implementación de una cola circular. Este código es un componente esencial para el manejo de la sincronización entre productores y consumidores en el programa principal. La cola actúa como un buffer entre los hilos productores, que colocan elementos en ella, y los hilos consumidores, que los retiran y procesan.

`queue_init` crea una nueva cola con una capacidad dada. Primero, asigna memoria para la estructura de la cola y luego para el arreglo de elementos (`elems`) que contendrá los datos de la cola. Si alguna de las asignaciones de memoria falla, se libera la memoria asignada y se devuelve `NULL`. La cola se inicializa con tamaño 0, la capacidad especificada, y los índices `front` y `rear` que indican dónde se añadirá o quitará el próximo elemento.

`queue_destroy` libera la memoria asignada para los elementos de la cola y para la estructura de la cola misma. Se debe llamar a esta función para evitar fugas de memoria cuando la cola ya no es necesaria.

`queue_put` añade un nuevo elemento al final de la cola si hay espacio disponible. Actualiza el índice `rear` y aumenta el tamaño de la cola. Si la cola está llena, no hace nada. La operación de añadir es segura para una cola circular, ya que el índice `rear` se envuelve alrededor del final del arreglo usando el operador módulo con la capacidad de la cola.

`queue_get` devuelve el elemento al frente de la cola y lo elimina de ella. Actualiza el índice `front` y disminuye el tamaño de la cola. Si la cola está vacía, devuelve `NULL`. Al igual que con `queue_put`, la operación de obtener es segura para una cola circular, ya que el índice `front` se envuelve alrededor del final del arreglo.

`queue_empty` comprueba si la cola está vacía comparando el tamaño de la cola con 0. Devuelve `true` si la cola está vacía y `false` en caso contrario.

`queue_full` comprueba si la cola está llena comparando el tamaño de la cola con su capacidad. Devuelve `true` si la cola está llena y `false` en caso contrario.

2.3 Queue.h

Es un archivo de cabecera en C, comúnmente utilizado para definir estructuras de datos y declaraciones de funciones que se pueden compartir entre varios archivos de código fuente. En este caso, el archivo define una estructura de datos para una cola, que es una colección ordenada de elementos donde los nuevos elementos se añaden al final y los elementos existentes se retiran desde el principio, siguiendo el principio de “primero en entrar, primero en salir” (FIFO).

La cola está diseñada para almacenar elementos que contienen información sobre productos, como un identificador único, una operación a realizar y la cantidad de unidades del producto. La estructura de la cola mantiene un registro del tamaño actual, la capacidad máxima, y los índices del primer y último elemento, lo que permite gestionar eficientemente las operaciones de añadir y retirar elementos.

Además, el archivo de cabecera proporciona declaraciones de varias funciones que permiten inicializar una nueva cola con una capacidad específica, destruir la cola liberando los recursos asignados, añadir un nuevo elemento al final de la cola, obtener y eliminar el primer elemento de la cola, y comprobar si la cola está vacía o llena.

Este tipo de estructura de datos y las operaciones asociadas son fundamentales en muchas aplicaciones de software, como la gestión de tareas, la planificación de procesos y la simulación de escenarios que requieren un orden específico de ejecución. La implementación de una cola mediante una estructura de datos permite a los desarrolladores organizar y manejar datos de manera eficiente y predecible.

3. Batería de pruebas

3.1 Store_manager

Descripción	Uso	Resultado esperado	Resultado obtenido
Verificamos que el tamaño del buffer sea positivo	**./store_manager <file> <num_prod> <num_consum> -1**	[ERROR] El tamaño del buffer debe ser mayor que 0	[ERROR] El tamaño del buffer debe ser mayor que 0
Verificamos que el número de productores y consumidores sean positivos	**./store_manager <file> -1 <num_consum> <buffer>** **ó** **./store_manager <file> <num_prod> -1<buffer>**	[ERROR] El número de productores debe ser mayor que 0 ó [ERROR] El número de consumidores debe ser mayor que 0	[ERROR] El número de productores debe ser mayor que 0 ó [ERROR] El número de consumidores debe ser mayor que 0
Si el fichero está vacío, el total y el stock de los productos es 0	**./store_manager empty_file <num_prod> <num_consum> <buffer>**	Total: 0 euros Stock: Product 1: 0 Product 2: 0 Product 3: 0 Product 4: 0 Product 5: 0	Total: 0 euros Stock: Product 1: 0 Product 2: 0 Product 3: 0 Product 4: 0 Product 5: 0

Comprobamos la funcionalidad del programa con el file.txt de aula global	**./store_manager file.txt <num_prod> <num_consum> <buffer>**	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
Verificamos que el programa maneje correctamente la concurrencia y que no haya condiciones de carrera ni problemas de sincronización entre los hilos cuando el tamaño del buffer, num_prod y num_consumidores es muy alto	**./store_manager file.txt 10000 10000 10000**	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
Verificamos que el stock de un producto no puede ser negativo, es decir que las unidades compradas (purchase) no pueden ser menores que las vendidas (sale).	**./store_manager negative_stock.txt <num_prod> <num_consum> <buffer>**	[ERROR] El stock del producto 3 no puede ser negativo	[ERROR] El stock del producto 3 no puede ser negativo
Verificamos que el número de operaciones en la cabecera del fichero sea menor o igual que el número de operaciones dentro del fichero. Si es mayor, se produce un error	**./store_manager file.txt <num_prod> <num_consum> <buffer>**	[ERROR] El número de operaciones en el archivo 'file.txt' es menor que el número esperado de operaciones	[ERROR] El número de operaciones en el archivo 'file.txt' es menor que el número esperado de operaciones

Verificamos que el número de argumentos sea el adecuado	**./store_manager <file> <num_prod>** **./store_manager <file> <num_prod> <num_consum>** **./store_manager <file> <num_prod> <num_consum> <buffer> holaa**	USO: ./store_manager <archivo_operaciones> <num_productores> <num_consumidores> <tamaño_buffer> USO: ./store_manager <archivo_operaciones> <num_productores> <num_consumidores> <tamaño_buffer> USO: ./store_manager <archivo_operaciones> <num_productores> <num_consumidores> <tamaño_buffer>	USO: ./store_manager <archivo_operaciones> <num_productores> <num_consumidores> <tamaño_buffer> USO: ./store_manager <archivo_operaciones> <num_productores> <num_consumidores> <tamaño_buffer> USO: ./store_manager <archivo_operaciones> <num_productores> <num_consumidores> <tamaño_buffer>

4. Conclusiones

4.1 Soluciones a problemas encontrados

Nos hemos enfrentado a varios desafíos durante esta práctica, pero hemos logrado superarlos con éxito gracias a una planificación cuidadosa y una implementación meticulosa.

Uno de los mayores desafíos fue garantizar una sincronización efectiva entre los hilos productores y consumidores para evitar condiciones de carrera y asegurar un acceso seguro a la cola circular compartida. Para resolver este problema, hemos hecho un esquema de sincronización utilizando mutex y variables condicionales. Utilizamos mutex para garantizar que solo un hilo pudiera acceder a la cola a la vez, evitando así conflictos y condiciones de carrera. Además, empleamos variables condicionales para coordinar la espera de los hilos cuando la cola estaba vacía o llena, lo que ayudó a evitar el consumo innecesario de recursos del sistema.

Otro desafío importante fue la distribución equitativa de las operaciones del archivo entre los hilos productores. Para abordar este problema, diseñamos un mecanismo de asignación de operaciones que dividía las tareas de manera justa entre los hilos y evitaba la sobrecarga de trabajo en algunos de ellos. Esto implicó asignar rangos de operaciones a cada hilo de manera inteligente, asegurándonos de que todas las operaciones se procesaran de manera eficiente y en paralelo.

En resumen, logramos superar los desafíos de esta práctica mediante una comprensión sólida de los conceptos de concurrencia y una implementación cuidadosa de la sincronización entre hilos.

4.2 Conclusiones personales

En esta última práctica, hemos fortalecido nuestras habilidades en programación en lenguaje C y hemos ganado comprensión sobre los desafíos y consideraciones necesarias al desarrollar aplicaciones concurrentes. Además, hemos tenido la oportunidad de aplicar conocimientos teóricos adquiridos en clases sobre sistemas operativos y programación concurrente en un contexto práctico y relevante.

En última instancia, esta experiencia ha proporcionado una base sólida para abordar problemas más complejos relacionados con la concurrencia y la gestión de recursos en sistemas informáticos que esperemos que nos sirvan para enfrentar desafíos futuros en el desarrollo de software escalable y eficiente.