UNIVERSIDAD SIMÓN BOLÍVAR
DEPARTAMENTO DE COMPUTACIÓN Y TECNOLOGÍA DE LA INFORMACIÓN
SISTEMAS DE OPERACIÓN I (CI3825)
PROF. GUILLERMO PALMA

PROYECTO 2 : TEATRO DE OPERACIONES

Estudiantes:

Franco Murillo 16-10782

Andrea Carrillo 17-10107

Sartenejas,Marzo 2025

DESCRIPCIÓN DEL SISTEMA

Para la resolución del problema del "Teatro de Operaciones para Drones", se han desarrollado dos implementaciones: una basada en procesos mediante la creación de procesos hijos en un entorno Linux. Y otra con hilos utilizando la biblioteca POSIX pthread.h .Para ambas implementaciones el programa inicia en main**,** donde se reciben el número de procesos/hilos n y el archivo de entrada con la configuración del teatro. Y se definen las estructuras Object y Drone, donde Object guarda la posición, la resistencia y el estado de destrucción de los objetos (así como un mutex asociado al objeto, para la implementación con hilos), mientras que Drone guarda su posición, radio de explosión y el poder destructivo. Las definiciones de las estructuras para cada implementación se encuentran en sus respectivos archivos **teatro.h.**

Solución Basada en Hilos

Para mejorar la eficiencia, el trabajo se divide entre varios hilos, cada uno encargado de procesar un subconjunto de drones. En **teatro.h**, se cuenta con una estructura adicional **ThreadData**, que contiene el rango de drones a procesar, además de punteros a una lista de objetos y a una de drones. También almacena el número total de objetos y las dimensiones del teatro de operaciones. Esto permite que cada hilo tenga acceso eficiente a la información necesaria para calcular los efectos de las explosiones en la cuadrícula.

En **main.c**, se lee el número de hilos y el archivo que contiene la instancia del teatro, pasados por el usuario, y se pasan a la función **executeT,** que se encuentra en **teatro.c.** Allí, se leen los datos del archivo de entrada, se inicializan los objetos y drones en memoria, y se crean los hilos para distribuir la carga de procesamiento.

Cada hilo ejecuta **process_drone()**, donde se recorren los drones asignados y se calcula el área afectada por su explosión. Para cada objeto dentro del área de impacto, se usa un **pthread_mutex_t** asociado al objeto para evitar condiciones de carrera cuando múltiples hilos intentan modificarlo simultáneamente. Si un objeto está en el radio de impacto de un dron, se ajusta su resistencia sumando o restando el poder explosivo, dependiendo de si es un objetivo militar (OM) o infraestructura civil (IC). Si la resistencia de un objeto llega a cero o cambia de signo (para OM cambia de negativa a positiva, y para IC viceversa), se marca como destruido.

Cada vez que un hilo va a verificar si un objeto entra en el rango explosivo del dron para modificar su resistencia, bloquea su mutex con **pthread_mutex_lock()**, realiza la verificación y modificación, y luego libera el mutex con **pthread_mutex_unlock()**, garantizando acceso seguro en entornos concurrentes.

Cuando todos los hilos han finalizado su ejecución, **executeT** usa **pthread_join()** para esperar su finalización. Luego, se evalúan los estados finales de los objetos y se clasifican como intactos, parcialmente destruidos o totalmente destruidos, dependiendo de su resistencia original y final. Finalmente, los resultados se imprimen en pantalla, se liberan los recursos utilizados y se destruyen los mutex asociados a los objetos.[1]

Solución Basada en Procesos

En la implementación con procesos (**teopp**), se crean procesos hijos en lugar de hilos para distribuir el trabajo. Considerando que un solo proceso que ejecute todas las explosiones

sería ineficiente, se dividió el trabajo entre varios procesos hijos que se encargan de procesar un subconjunto de drones. En **ejecutar_teatro()**, ubicada en **teatro.c**, se utiliza **fork()** [2][3]para crear n procesos, cada uno encargado de aplicar los efectos de un subconjunto de drones en la cuadrícula del teatro. Dado que los procesos tienen memoria independiente, se emplea **mmap()**[4] para crear un segmento de memoria compartida donde se almacenan los objetos, permitiendo que todos los procesos accedan y modifiquen la resistencia de los objetos sin necesidad de copiar estructuras.

Cada proceso hijo ejecuta **process_drones()**, que procesa un grupo de drones y aplica sus explosiones en la cuadrícula del teatro. Se calcula el área afectada por cada dron considerando su posición, radio de destrucción y los límites del teatro. Luego, se verifica qué objetos están dentro del área de impacto y se modifica su resistencia según su tipo. Como múltiples procesos pueden modificar la resistencia de un mismo objeto simultáneamente, es necesario evitar condiciones de carrera. Para garantizar que los procesos no accedan simultáneamente a la misma celda, se utilizan **semáforos POSIX (sem_t)** [5][6], que garantizan acceso exclusivo a los objetos.Cada vez que un proceso va a modificar la resistencia de un objeto en la memoria compartida, ejecuta **sem_wait()** para bloquear el acceso a otros procesos. Una vez que la modificación ha finalizado, libera el recurso con **sem_post()**, permitiendo que otros procesos puedan operar sobre el objeto.

Una vez calculados el ataque de los drones, cada proceso hijo finaliza su ejecución y el proceso padre utiliza **wait()** para asegurarse de que todos han concluido antes de imprimir los resultados. Luego, evalúa el estado de los objetos en la memoria compartida y los clasifica como **intactos, parcialmente destruidos o totalmente destruidos**, para cada tipo de objeto, comparando su resistencia actual con la original. Por último, se imprimen los resultados en pantalla y se liberan los recursos utilizados, cerrando los semáforos con **sem_close()** y **sem_unlink()**, y liberando la memoria compartida con **munmap().**

Finalmente, se puede observar las diferencias entre las dos implementaciones.En la solución con procesos, cada proceso hijo opera de manera independiente con su propia memoria, lo que requiere el uso de memoria compartida y semáforos para la sincronización. Aunque este enfoque permite un mayor aislamiento entre tareas, introduce una sobrecarga significativa en la creación y gestión de procesos, ya que cada uno necesita su propio espacio de memoria y mecanismos de comunicación.

Por otro lado, la solución con hilos es más eficiente en términos de uso de memoria y velocidad de ejecución, ya que todos los hilos comparten el mismo espacio de direcciones, eliminando la necesidad de memoria compartida adicional. La sincronización se maneja mediante mutexes , que garantizan el acceso exclusivo a los objetos afectados por los drones sin la complejidad de los semáforos. Esto reduce la sobrecarga de administración y hace que el cambio de contexto sea más rápido. En general, el uso de hilos reduce el consumo de recursos del sistema y permite una comunicación más rápida, opción más eficiente para el problema planteado.

REFERENCIAS BIBLIOGRÁFICAS

[1]J.-B. Persson, *Linux System Programming Techniques*. Packt, 2023.

[2]Manpage fork(2), Linux man-pages. Disponible en: https://man7.org/linux/man-pages/man2/fork.2.html.

[3]GeeksforGeeks, "Fork System Call", Disponible en: https://www.geeksforgeeks.org/fork-system-call/?ref=header_outind, [fecha de acceso].

[4]Manpage mmap(2), Linux man-pages. Disponible en: https://man7.org/linux/man-pages/man2/mmap.2.html.

[5]Manpage sem_init(3), Linux man-pages. Disponible en: https://man7.org/linux/man-pages/man3/sem_init.3.html.

[6]GeeksforGeeks, "Use of POSIX Semaphores in C", Disponible en: https://www.geeksforgeeks.org/use-posix-semaphores-c/?ref=header_outind, [fecha de acceso].