DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico: *Threading*

Sistemas Operativos

6 de noviembre de 2022 Sistemas Operativos

Integrante	LU	Correo electrónico
Schwartzmann, Alejandro Ezequiel	390/20	a.schwartzmann@hotmail.com
Kim, Lucas	456/20	lucasthkim99@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
https://exactas.uba.ar

Índice

Índice de figuras

1. Introducción

El objetivo de este trabajo práctico es desarrollar el juego Captura la Bandera. Para ello, contamos con un esquema básico del sistema, el cual estaba incompleto. Nuestro trabajo fue completar dicha instancia para que este pudiese correr con el funcionamiento correcto.

El funcionamiento del programa era el siguiente: Dada una configuración de una partida, recibida a través de estándar input, el programa la carga en memoria, construyendo la instancia necesaria de "GameMaster", el código coordinador del juego, y los dos equipos, con sus respectivos jugadores inicializados en el tablero, y utilizando una de las cuatro estrategias definidas.

Luego se inicia la partida, donde cada equipo se ejecuta en su respectivo turno y juegan hasta que alguno de los dos o encuentre la bandera, o por condiciones de la configuración no puedan ejecutar más movimientos que los acerquen a su objetivo y, por lo tanto, empatan.

1.1. Detalles de implementación

Para implementar el juego, fue necesario hacer uso de *multi-threading*, ya que a pesar de que cada equipo se ejecuta en su respectivo turno, los jugadores dentro de dicho equipo corre en simultáneo representado por un thread. Se utilizó la biblioteca thread provista por la biblioteca estándar de C++. [1] Al habilitar el uso de threading, hizo falta garantizar *thread-safety* en todas las estructuras que pudiesen ser escritas y leídas en simultáneo por varios threads. Para ello, utilizamos los métodos y estructuras estudiadas de sincronización durante la materia para asegurarnos que no se presenten condiciones de carrera.

1.2. Como compilar

Simplemente, ejecutar el script compile.sh, o compileAndTest.sh. El script compile.sh remueve el binario, el caché de cmake y vuelve a hacer make del proyecto. La versión con testeo ejecuta luego testing.sh.

1.3. Código y Funciones

A partir de las estructuras provistas por la cátedra, agregamos estructuras adicionales, atributos y funciones para poder asegurar el funcionamiento correcto del juego.

El programa se compone de 2 clases grandes designadas a manejar el juego: la clase GameMaster y la clase Equipo.

2. Ejercicios

2.1.

Hicimos las implementaciones de belcebú (gameMaster)

2.1.1. Constructor gameMaster

El constructor se encarga de dada una configuración de instancia de juego, inicializar dicha instancia con el tamaño del tablero, los jugadores y sus respectivas posiciones. Dado que los jugadores (threads) corren en paralelo, se podría dar el caso de que un thread pise una celda que ya está siendo escrita, lo cual debemos evitar. Es por eso que, fue necesario desarrollar estructuras de control de sincronización para asegurarnos que no haya *race condition*.

Dado que el GameMaster tiene el rol de sincronizar los turnos entre los dos equipos y además garantiza *thread-safety* en el tablero, en él caen los dos semáforos de cada equipo.

Se añadieron dos semáforos, turno_rojo y turno_azul los cuales controlan la ejecución de cada jugador, de modo que solo se ejecuten cuando es el turno correspondiente del equipo. También,

[1] https://en.cppreference.com/w/cpp/thread/thread

se añadió una barrera como otra herramienta de sincronización para asegurar que un equipo haya terminado correctamente para pasar al turno del oponente. De este modo, nos aseguramos que ningún jugador pueda intentar moverse cuando no le corresponde el turno y además que para pasar de turno, el equipo completo debe salir de la estrategia, y, por lo tanto, debió realizar la cantidad de movimientos que tenía disponibles para hacer.

Para el constructor de gameMaster, solo hizo falta completar las estructuras de sincronización. Inicializamos los semáforos con la función sem_init con el valor inicial 0. Estas fueron declaradas anteriormente en el header de gameMaster.

2.1.2. Mover jugador

Es el método encargado de mover un jugador en el tablero dada la dirección en la que se quiere mover y el número de jugadores. Con el número del jugador, sabemos cuál es el índice dentro del vector de posiciones en donde se encuentra guardada la posición del jugador. El método se encarga de obtener la posición actual del jugador dependiendo de a que equipo corresponde el turno y luego chequear que el movimiento sea válido (es una posición válida y que este libre). Luego usamos el método mover_jugador_tablero que ya nos fue dada y actualizamos el vector de posiciones. Por último, comprobamos si al moverse el jugador, este capturo la bandera. Devolvemos 0 si se capturó la bandera, y de lo contrario, el número de ronda. No hacemos uso de los semáforos u otras estructuras de sincronización en el método, ya que este es llamado desde la clase Equipo, la cual se encargara de que no se puedan mover 2 jugadores a la vez.

2.1.3. Termino ronda

Actualizamos los valores del turno y el número de ronda. Si luego de la ronda, el juego no termino, se hacen N waits en la barrera esperando los N signals de los jugadores que son llamados en el método jugador dentro de la clase equipo. Cada jugador, luego de terminar su comportamiento dentro de su turno, que depende de cada estrategia, hace un signal a la barrera de gameMaster avisándole que termino de jugar. Esto asegura la correcta terminación de un equipo. Una vez hecho los N waits en la barrera de gameMaster, se hacen los N signals/post al semáforo de turno del equipo correspondiente al finalizar la ronda. De este modo, el equipo contrario podrá jugar N veces, o mejor dicho pasan los N respectivos jugadores usando el semáforo turno_azul o turno_rojo en su turno correspondiente. Y al terminar de jugar vuelven a esperar este semáforo. En caso de que el número de rondas supere 500, se declara empate. Si alguien gano, al terminar la ronda, se hacen N signals a los semáforos de turnos de ambos equipos.

En resumen, termino_ronda se asegura con la barrera que hayan jugado los N jugadores del equipo y hasta no recibir los signals correspondientes no pasara a hacer los signals del semáforo del equipo al que le tocara jugar después de finalizar la ronda. De lo contrario, podrían existir trazas donde juegue el equipo contrario sin que sea su turno.

2.2. Equipo

Desarrollamos los métodos de equipo.

2.2.1. Constructor Equipo

Para completar el constructor de equipo, necesitamos realizar inicializaciones de sincronismos para poder mantener thread-safety en la clase: garantizar que los jugadores se ejecuten una cantidad limitada de veces (dependiendo de la estrategia sería la cantidad de jugadores o el quantum) y si la estrategia lo requiere, un orden establecido entre los threads.

Para ello, declaramos en el header de equipo 2 estructuras de sincronización, una barrera y un vector de N semáforos, siendo N la cantidad de jugadores por equipo. Agregamos el código necesario para inicializar los N semáforos del vector (vec_sem) y la barrera. Ambas fueron inicializados en 0.

2.2.2. Comenzar

Definimos que equipo comienza a jugar, en nuestro caso, fue el equipo rojo e hicimos N signals al semáforo turno_rojo de gameMaster para que los jugadores puedan jugar N veces. Asimismo, debimos modificar las estructuras de sincronización y/o variables dependiendo de la estrategia a usar. Para Round Robin, hicimos un signal al primer semáforo del vector para que el primer jugador del equipo pueda moverse. Para la estrategia shortest, usando el método jugador_minima_distancia, calculamos el primer jugador a moverse. Y para la estrategia USTEDES usando el método jugador_maxima_distancia, calculamos el primer jugador a moverse. Lanzamos los threads con el método jugador y el índice respectivo de cada thread/jugador.

2.2.3. Jugador

Es el método que llama el thread, debemos definir todo comportamiento de jugador acá. Nos encargamos del correcto funcionamiento de nuestras estructuras para la sincronización por cada estrategia.

Nos aseguramos la sincronización de turnos, no puede haber un jugador intentando jugar cuando no es su turno. Definimos en GameMaster los semáforos por equipo y los usamos en este método. Para ello, usamos los semáforos de gameMaster turno_rojo y turno_azul, hacemos un wait de modo que jugadores se quedaran esperando en el semáforo si no les corresponde el turno y juegan si les corresponde. Nos aseguramos que funcione correctamente al haber implementado termino_ronda en gameMaster, allí hacemos N signals si y solo si el método termino_ronda es llamado y jugaron los N jugadores (usamos la barrera para asegurarlo).

Aseguramos que solo un jugador/thread esté ejecutando con un mutex delimitando la zona de escritura como una sección crítica. A su vez para simplificar la terminación del juego, cuando un equipo gana, el jugador libera todas las estructuras de sincronización, idea que se ve reflejada en Belcebu, ya que al terminar la ronda checkea si el equipo gano, no espera los posts de cada jugador para cambiar la ronda sino que libera las estructuras para permitir los jugadores terminar. Cada jugador al inicio del turno checkea si termino el juego y termina su ejecucion.

El método queda en un ciclo hasta que termine el juego, este caso hace un return. De lo contrario, definimos el comportamiento de los jugadores dependiendo de la estrategia usada:

1. `Secuencial`

   La idea es que los jugadores de un equipo jueguen uno por uno sin dar un orden. Para evitar escrituras concurrentes, hacemos uso del mutex, que además de evitar race-condition, nos da un orden no determinístico. Luego de que un jugador libere el mutex, el próximo en jugar será el que lo adquiera. De este modo, usando un mutex podemos hacer que los jugadores jueguen uno por uno y sin un orden determinado. Dentro de la sección crítica, si el jugador se puede mover hacia la dirección que quiere, se lo mueve y se actualiza en el vector de posiciones. Al liberar el mutex, los jugadores se quedan haciendo wait en la barrera hasta que el último jugador del equipo o el jugador que captura la bandera, libera los N jugadores haciendo la misma cantidad de signals y llama al método termino_ronda del gameMaster. Por último, cada jugador hace un signal de la barrera dentro de la clase gameMaster el cual asegura que todos los jugadores hayan terminado su ronda antes de dejar al equipo contrario jugar (Este hace N signals al semáforo de turno del equipo contrario).

2. `Round Robin`

   En esta estrategia, utilizando el semáforo de su respectivo turno, cada equipo coordina la entrada a jugar. Siempre inicia el turno el jugador 0 de cada equipo. Como las demás estrategias inicialmente los jugadores esperan por su turno y cuando reciben el post avanzan. Al avanzar entran en la estrategia RR donde la idea es la siguiente:
   Cada jugador espera su respectivo momento para jugar en el turno de su equipo. Un jugador pasa su respectivo wait solamente si es el primero numéricamente o si su predecesor numérico lo habilito, esto permite ordenar la ejecución por número. Para lograr esto, utilizamos

un vector de semáforos, cada jugador tiene un semáforo al que espera. En cada ronda de movimientos que se juega durante el turno de un equipo, un jugador está constantemente loopeando un while, intentando jugar o checkeando condiciones para salir del while. El jugador inicia el while esperando el post de su semáforo, pide el mutex y a partir de esto pueden ocurrir las siguientes situaciones:

- Si termino el juego en el movimiento del jugador predecesor. El jugador entonces libera todas las barreras, levanta la barrera interna de equipo y la de vec_sem para cada jugador. Y luego retorna el mutex y al hacer return joinea su thread terminando su ejecución y dejando la sincronización tal que el próximo también termine.
- Si termino la ronda el jugador predecesor, entonces suelta el mutex y sale del while y espera en barrera.
- Si tiene quantum para jugar, realiza su movimiento, checkea que no haya terminado el juego. Si así lo hizo, repite los pasos previamente explicados. Si no termino, entonces habilita para que el próximo jugador juegue, loopea volviendo al inicio y se queda esperando en su semáforo vec_sem.
- Si el quantum es 0, libera los semáforos de cada jugador, setea la variable local vuelta_rr en false, para que los jugadores restantes sepan que termino la ronda, hace un post extra para el jugador 0, ya que ahora que termino la ronda él estaba esperando en vec_sem, pisa el signal, pero para la próxima ronda necesita que el semáforo esté en 1, por lo tanto, recibe 2 posts ese semáforo. Luego libera el mutex y llama a belcebu para terminar la ronda.

La barrera se encuentra fuera del loop para que esperen una vez terminada la ronda y poder coordinar el término de turno, una vez que la barrera se llena, cada uno hace el post a belcebu y vuelven a esperar en el semáforo de su respectivo equipo.

3. `Shortest distance first`

   La estrategia consiste en dejar mover al jugador que más cercano este a la bandera contraria, un jugador a la vez por equipo. Para esta estrategia, tenemos precalculado el índice del jugador que se encuentra más cerca de la bandera, y cada vez que el jugador se mueva, se calcula de vuelta con el método jugador_minima_distancia. Reutilizamos la sincronización de secuencial, donde usamos un mutex para que únicamente se intente mover un jugador a la vez. Una vez obtenido el mutex, solo se moverá el jugador que realmente esté más cercano a la bandera, y de lo contrario, libera el mutex y se queda esperando en la barrera de Equipo. Nuevamente, el ultimo jugador en obtener el mutex (o el que capturo la bandera), sea el que debía jugar o no, resetea el contador de jugadores que ya jugaron, levanta la barrera haciendo N signals y termina la ronda usando el método del belcebu. Finalmente, cada thread hace el signal para la barrera de gameMaster para poder finalizar la ronda. En resumen, todos los jugadores piden el lock, pero el único que puede modificar su posición es el jugador con la mínima distancia a la bandera. Luego de que todos los threads hayan intentado jugar, se pasa de ronda para que el jugador del equipo contrario se mueva y así sucesivamente.

4. `Nuestra estrategia` Consiste en mover al jugador que se encuentre más lejos de la bandera, una cantidad de veces igual al quantum inicializado, en un mismo turno. Luego termina la ronda y juega el oponente. Si este no se puede mover, pierde el turno. Por lo tanto, al igual que shortest distance first, vamos a tener precalculado el índice del jugador más lejano a la bandera. Al igual que la estrategia anterior, usamos un mutex para garantizar que un jugador actualice las variables a la vez para evitar race-condition. Una vez que cada jugador obtiene el mutex, solo se mueve si es el jugador que está más lejos de la bandera, de ser así, se mueve quantum cantidad de veces o hasta que no se pueda mover más. De igual manera que secuencial y shortest, los jugadores se quedan esperando en la barrera y solo el último o el que haya capturado la bandera levanta la barrera. Además, el último actualiza la variable

del jugador al que le corresponde moverse en el próximo turno. Luego, cada jugador hace un signal a la barrera de gameMaster.

2.3. Buscar bandera contraria

Para poder medir la diferencia entre tiempos de búsqueda paralelos vs. secuencial, usamos la librería chronos porque permite castear los tiempos en microsegundos de manera más sencilla que time.h.

2.3.1. Secuencial

Función que busca la bandera en el tablero de manera secuencial. No implementamos paralelismo ni sincronización. La utilizamos como benchmark para comparar contra buscar_bandera_contraria.

2.3.2. Paralela

Los jugadores se reparten las celdas del tablero como: cantidad de celdas / cantidad de jugadores. Es decir, buscan en simultáneo sobre un grupo de celdas asignadas para cada jugador dependiendo de su índice. No es necesario usar las estructuras de sincronización de los puntos anteriores, ya que son lecturas de un tablero que, usando el método en_posición de GameMaster, no se modifican. Lo que hace esta función es dividir el tablero en fracciones iguales, y se le asigna una fracción del tablero a cada jugador para que lea e intente encontrar la bandera. Se implementa en la función Equipo::jugador(int nro_jugador), donde al inicio de la llamada cada jugador busca la bandera contraria. Agregamos un mutex para controlar la escritura de la variable que contabiliza el tiempo total que lleva encontrar la bandera.

Comparamos los tiempos medidos entre la búsqueda paralela y la secuencial. A priori, esperamos que la búsqueda paralela sea más rápida que la secuencial, ya que al igual que un algoritmo "divide and conquer", cada thread/jugador obtiene una parte del tablero para buscar y la paraleliza. En cambio, la búsqueda secuencial no aprovecha la posibilidad de paralelizar el trabajo y usa un solo jugador que tiene que buscar la bandera sobre toda la tabla. Al comparar los resultados, pudimos ver que la búsqueda secuencial era en casi todos los casos igual o mejor que la búsqueda paralela. Por un lado, es posible que el rendimiento de la búsqueda estuviese muy afectado por la posición de la bandera en comparación con el punto de partida de la búsqueda. En el caso de la bandera roja, la búsqueda secuencial era mucho más efectiva, ya que la bandera roja siempre se encuentra en la primer columna, haciendo que el punto de partida (1,1) este cerca de la bandera. (La config de gameMaster hace un assert que pide que la primer coordenada de la bandera roja sea igual a 1). Asimismo, es posible que en la búsqueda paralela, el thread encargado de buscar la zona donde se encuentra la bandera, por temas de scheduling, etc., sea el último en buscar empeorando la performance. Para poder ver los tiempos, el método buscar_bandera_contraria_secuencial tiene comentado el print con el tiempo de la búsqueda secuencial. Asimismo, el método jugador en la clase Equipo tiene comentado el print del tiempo de la búsqueda paralela.

3. Testing

Para testear el código utilizamos un script hecho en bash que automatiza la ejecución del ejecutable compilado, donde ejecuta 100 veces cada estrategia utilizando la configuración de config.csv. Inicialmente, utilizamos 5 archivos CSV escritos a mano donde sabíamos que equipo debía ganar según con qué estrategia.
A partir de estos archivos y utilizando la función de belcebu dibujame(), que escribe por pantalla el estado del tablero, pudimos asegurar la correcta ejecución de cada turno. Una vez hecho esto pasamos a testear con tableros de mayor tamaño generados automáticamente y con un número mayor de jugadores

4. Casos de test

Los casos de test encontrados en la carpeta /config fueron creados con un script en Python. Este crea archivos CSV con los parámetros necesarios para crear una instancia del juego CTF. (Coordenadas de jugadores, cantidad de jugadores, posiciones de las banderas, etc.). Adjuntamos 10 casos generados de test con el script de Python de forma aleatoria usando la función random. El archivo documentacion.txt contiene el significado de cada valor guardado en el archivo CSV.