DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP de SO

4 de mayo de 2024 Sistemas Operativos

Grupo: 16

Estudiante	LU	Correo electrónico
Lucas Mas Roca	122/20	lmasroca@gmail.com
Luciana Skakovsky	131/21	lucianaskako@gmail.com
Mateo Cantagallo	143/21	mateocantagallo@gmail.com
Alan Yacar	174/21	alanroyyacar@gmail.com

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
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Introducción

Una parte crucial de los sistemas operativos modernos es la capacidad de correr procesos de manera concurrente que utilicen y modifiquen los mismos datos y recursos. Ya sea porque distintos equipos comparten estos recursos o porque un mismo equipo paraleliza una misma tarea en varios procesos distintos. Con esta herramienta se puede mejorar drásticamente el rendimiento general de nuestro sistema. De todas estas maneras, una que se centra principalmente en mejorar el rendimiento de un proceso y de la que hablaremos en este informe es el concepto de threading.
Threading puede pensarse como un proceso que se separa en varios subprocesos que corren a la par y comparten los recursos del proceso original. Este esquema funciona para paralelizar tareas que pueden claramente dividirse, por ejemplo sumar dos vectores de enteros, está claro que podemos sumar dos posiciones en paralelo y esto dividirá el tiempo que hubiera tardado sumar una posición y luego la otra en un proceso clásico. En ese ejemplo de la suma de vectores puede verse que la implementación no tendrá mucha complejidad, como cada suma ocurrirá en total paralelizar los threads aunque compartan los vectores en memoria nunca intentaran de acceder a las mismas posiciones. Cuando esto si sucede, es decir, cuando distintos threads acceden al exacto mismo recurso, en nuestro caso la misma posición de memoria, es cuando pueden surgir problemas de sincronización vistos en la materia (Como race conditions) y debemos utilizar las herramientas vistas para tener una correcta sincronización entre ellos y así evitar estos problemas.
Como ya mencionamos, en este informe vamos a explorar una implementación de threading, y todo el proceso de sincronización necesario para su funcionamiento correcto, utilizando una simulación de un juego de capturar la bandera.

2. Como funciona el juego

El juego consta de 2 equipos (Azul y Rojo) de los cuales cada equipo contaran con una cantidad de jugadores posicionados en un tablero de $N \times M$ donde su objetivo es alcanzar la bandera del equipo contrario primero. Cada equipo tiene un turno en el que se podrán mover sus jugadores, una vez hechos los movimientos le toca el turno al otro equipo. Estos se alternan los turnos moviendo a sus jugadores hasta que se alcance una bandera contraria o ninguno de los equipos pueda mover a un jugador, caso en el que se declara un empate. La manera en la que pueden moverse los jugadores es de a una casilla a la vez en una de las cuatro direcciones cardinales. La dirección de estos movimientos y que jugadores del equipo los realizan estará determinada por una estrategia que comparten ambos equipos.

3. Implementación del juego

Como mencionamos en la introducción, el objetivo es explorar el uso y sincronización de threads mediante la implementación de este juego. El código está separado en dos archivos principales: gameMaster.cpp y equipos.cpp Con el objetivo de sincronizar el juego, el Game Master, llamado belcebú, se encarga de realizar todos los cambios al tablero, la habilitación a los equipos de ejecutar en sus turnos, y los chequeos de si termino el juego, ya sea por un ganador o por empate. Al comenzar el juego se inicializan dos instancias de la clase equipo en equipo.cpp. Una representa al equipo rojo y la otra al azul. Una vez inicializados, cada equipo largará un thread por cada jugador en el juego. Estos threads que representan a los jugadores primero van a realizar una búsqueda paralela de la bandera rival, una vez encontrada, entran en un ciclo que continúa hasta que termine el juego donde cada iteración representa un turno (con el detalle de que en Round Robin pueden hacer más de

una iteración en el ciclo en un mismo turno, en este caso cada iteración representa un intento de movimiento de ese jugador). Dentro de este loop cada jugador decidirá hacia donde moverse y llamara al método `mover_jugador` de belcebú. Esta función se encarga de primero chequear que el movimiento pedido puede realizarse (si no se está moviendo a una posición ocupada o si es una posición inválida), y luego lo realiza si se puede o, si es el movimiento que captura la bandera, se setea el ganador que previamente estaba indefinido y libera a todos los threads (la acción de liberar los threads y el resto de estructuras de sincronización mencionadas se explican en la sección de sincronización). Dentro del loop de turno cada jugador entrará en el switch que corresponde según la estrategia que se eligió al comenzar el juego, estas son:

- Secuencial: Cada equipo mueve a todos sus jugadores una vez hacia la bandera contraria. Para realizar esto cada jugador llama a la función `mover` con su número, esta función consigue la posición del jugador utilizando el vector de posiciones de belcebú y la bandera contraria que ya se encontró, y con estos datos llama a la función `mover_jugador` de belcebú antes mencionada (esto sucederá en todas las estrategias, ya que siempre que nos movamos buscamos acercarnos a la bandera contraria). Cuando todos los jugadores se hayan movido, el último llamara a la función `termino_ronda` de belcebú, que decide si hubo un empate y si no le permite al otro equipo ejecutar su turno

- Round Robin: Como en secuencial, cada jugador tendrá que llamar a la función `mover`. La diferencia es que ahora usamos una variable quantum que determina cuantos jugadores pueden moverse en cada equipo. Estos jugadores deben moverse en orden, es decir, si el quantum es 5 se moverá primero el jugador 1, luego el 2, hasta el 5. Si el quantum excede la cantidad de jugadores entonces luego del último jugador se vuelve al primero. Por esta razón cada jugador primero debe mirar la variable `quantum_restante` que se resta cada vez que un jugador realiza su movimiento, cuando llega a 0 el jugador lo reinicia y llama a `termino_ronda` (este jugador no realiza un movimiento en esa iteracion ya que de lo contrario nos moveriamos quantum+1 veces).

- Shortest: Cada equipo no solo busca la bandera contraria al comenzar, sino que también cada uno busca al jugador más cercano a la bandera contraria y lo guarda en la variable `masCercano`. En cada turno los equipos solo moverán a este jugador.

- Ustedes: Pensamos varias estrategias posibles al principio. Una posible era agregar un elemento de defensa al juego. Los equipos no solo se moverían en busca de la bandera contraria, sino que también tratarían de defender su bandera bloqueando los caminos posibles de los contrarios. Otra posibilidad era usar una de las estrategias como shortest o Round Robin pero agregar una lógica que permita esquivar al encontrarse con un obstáculo. Por como estaba implementado el código template original, específicamente tener toda la lógica del tablero en una clase separada de los jugadores probo ser difícil implementar estas ideas. Vimos que implementar estas ideas requería agregar bastante complejidad (los jugadores en cada uno de sus turnos deberían tener que hacer varios llamados a belcebú y en base a eso tomar decisiones de hacia donde moverse o si pasar su "turno") que no era complejidad asociada a la sincronización que es nuestro principal tema de estudio, por lo tanto, decidimos simplemente implementar una estrategia similar a shortest, pero permitiendo a ese mismo jugador moverse la cantidad de veces que el quantum lo permita.

4. Sincronización

En la sección anterior se explicó como funciona el juego y las estrategias a grandes rasgos para poder comprender con más facilidad el principal tema a tratar. La dificultad que planteaba este problema es resolver la sincronización que requieren todos los jugadores corriendo en paralelo. Debemos resolver tres principales situaciones de sincronización: El acceso a los recursos compartidos como el tablero, el orden entre los jugadores para cada estrategia, y por último los turnos entre equipos. Para resolver los turnos primero pensamos en utilizar un semáforo para cada equipo que sea inicializado en cantidad de jugadores para un equipo y en 0 para el otro, luego agregar un wait al comienzo de cada loop. Así, para el equipo con el semáforo habilitado pueden entrar la cantidad de jugadores al loop, realizar su turno, y al volver a la siguiente iteración el semáforo estará en 0 y quedará trabado hasta que se le realice un signal, los signal se mandarían con la función `termino_ronda` al otro equipo. Los problemas que encontramos con este método fueron dos. Primero, no proveía suficiente versatilidad para realizar la sincronización entre jugadores de un mismo equipo para estrategias como Round Robin; segundo, al ser un código tan corto el que se encuentra dentro de la iteración del loop, existía la posibilidad de que un mismo jugador entre al loop y realice toda la iteración antes que se desaloje por el scheduler con lo que volvería a consumir otra instancia del semáforo, "robándole" un turno a un compañero. Por estas razones descartamos la idea y, en cambio, usamos dos vectores de semáforos en belcebú, uno por cada equipo donde cada semáforo corresponde con un jugador, este esquema nos provee con mucha más versatilidad a la hora de sincronizar.
El problema del tablero compartido se presenta una vez comienzan los turnos y los jugadores lo empiezan a modificar mediante la función `mover`. Al comienzo de la partida, mientras se están buscando las banderas, todos los jugadores están solamente leyendo el tablero, por lo que no se presenta un problema por la paralelización. Para resolver el problema, una vez empiezan las escrituras, agregamos un mutex dentro de la función `mover_jugador` de belcebú para que solo un jugador pueda acceder y modificar el tablero a la vez.
La función `mover_jugador` consigue la coordenada anterior del jugador que llama a la función, verifica que el movimiento que intenta realizar es válido (no se sale del tablero), luego dentro de un área crítica verifica si ya termino el juego (en ese caso simplemente libera el área crítica y retorna), se puede mover el jugador y lo mueve, o no se puede mover el jugador. La función `termino_ronda` realiza una detección de empate guardando las posiciones del último turno de cada equipo, verifica si ambos equipos no consiguieron mover jugadores en dos rondas consecutivas y en ese caso declara un empate (además envía signals a todos los jugadores de ambos equipos para liberar el juego), además se encarga de cambiar el equipo que juega, aumentar el número de ronda y enviar los signals necesarios (según la estrategia) al equipo que le toca jugar, restableciendo sus semáforos.

Cada estrategia va a resolver de su manera la sincronización entre compañeros de equipo:

- Secuencial: Un equipo comienza con sus semáforos en 1 y el otro en 0. Cada jugador que entre en el primer equipo va a primero moverse y luego modificar una variable atómica (y de forma atómica verificar si es el último jugador del equipo) que funciona como contador de cuantos jugadores ya realizaron su turno, así el último puede llamar a la función `termino_ronda` que mandará signals a todos los semáforos del equipo contrario.

- Round Robin: Ahora queremos tener un orden entre los jugadores. Los semáforos serán todos 0 excepto el del primer jugador del primer equipo. Todos quedarán esperando en el wait anterior al loop (en la primera iteración, luego, en las iteraciones subsiguientes esperaran en el wait al final del cuerpo del loop) y el único que entrara es el primer jugador. Este jugador restará el `quantum_restante` y realizara su movimiento, luego mandara un signal al siguiente jugador en el anillo, cuando se acabe el `quantum_restante` el jugador llamara a `termino_ronda` que

mandará un signal al primer jugador del equipo contrario. Cabe destacar que en esta estrategia no hay paralelización, ya que el loop está siendo ejecutado por un solo thread a la vez, por lo que no es necesario que `quantum_restante` sea atómica.

- Shortest: En esta estrategia solamente juega un jugador de cada equipo, por lo que se inicializa igual que secuencial (haciendo un checkeo de si el jugador es el más cercano, si es así se mueve y si no vuelve al wait), pero no hay chequeo de quantum, cada jugador llama siempre a `termino_ronda` después de moverse, el cual manda los signal a todos los jugadores del equipo rival.
- Ustedes: Igual que shortest, pero se realizan quantum movimientos antes de llamar a `termino_ronda`.

5. Experimentación

Puede verse que, aunque se corran en paralelo los threads de los jugadores, la tarea de jugar el juego no se beneficia de estar paralelizada. Primero, al tener que jugar por turnos, la mayoría de threads están esperando para un instante de tiempo y segundo, la lógica que corre cada jugador es tan corta que correrlos en secuencia no presentaría mucha diferencia. Como mencionamos al principio, la paralelización es usada para mejorar el rendimiento de tareas que pueden correrse en partes a la par. En nuestro código la única parte que puede verdaderamente aprovechar la paralelización es cuando ambos equipos buscan a la bandera contraria. En nuestro código separamos el tablero bloques y se le asigna uno a cada jugador donde va a buscar la bandera. Usamos un booleano atómico (este valor podría ser no atómico, ya que ese valor solo se modifica una vez de false a true y no debería pisarse nunca porque no hay intersección entre las casillas que busca cada jugador, por lo cual solo uno encontrara la bandera y modificara el booleano) para detectar cuando encontramos la bandera contraria, de esta forma frenamos la búsqueda si ya encontramos la bandera, además usamos esto para que los jugadores que ya terminaron de revisar su bloque del tablero se queden esperando a que se encuentre la bandera antes de seguir jugando. En cuanto a esto último, usamos busy waiting sobre este booleano (aunque una mejor solución posible sería usar un semáforo y realizar un wait, como el código es tan corto, consideramos que no valía la pena). Esta es la manera más simple de separar una búsqueda, ya que en la materia vimos una mejor forma donde cada thread busca en la siguiente posición libre, así se podría evitar que si un thread termina su sección antes que otro no se quede esperando. En este caso no tenemos razón para pensar que un thread tarde más en buscar en su sector que cualquier otro, por lo que lo dejamos con el primer método. En esta sección vamos a experimentar con el fin de ver que tanto rendimiento puede mejorarse paralelizando la tarea. Los experimentos fueron realizados con las siguientes especificaciones:

- OS: Ubuntu 22.04 LTS x86_64
- Kernel: 5.15.0-52-generic
- CPU: Intel i7-10700F **(16 threads)** @ 4.8GHz
- Memoria: 31979MiB

Separamos los experimentos más largos poniendo sleep(60) cada cierta cantidad de búsquedas de bandera para dejar que la CPU baje su temperatura, con el objetivo de disminuir (o idealmente eliminar) la CPU throttling.

5.1. Experimento 1: Tiempo vs threads

La idea de este primer experimento es comparar el rendimiento de la búsqueda de la bandera en paralelo usando distintas cantidades de threads y compararlo con buscarla secuencialmente. Para esto, corrimos el algoritmo de buscar la bandera en un tablero de 10000 $\times$ 10000 y ubicamos la bandera en la última posición, de forma que estaríamos experimentando en el peor caso de búsqueda para ambos casos.
Por un lado, lo corrimos de forma secuencial sin threads y luego corrimos el mismo algoritmo paralelizando la búsqueda en distintos threads donde se dividían las secciones del tablero donde buscan la bandera. Tanto la corrida secuencial como la paralelizada con cada cantidad de threads se corrió 100 veces y se calculó el tiempo promedio que le tomo en cada caso encontrar la bandera. El experimento se probo con 1, 5, 8, 10, 16, 20, 25, 32, 40 y 64 threads y se compararon los resultados.
Hipótesis
Lo que esperábamos ver en este experimento es una clara mejora en rendimiento al aumentar la cantidad de threads hasta conseguir el rendimiento ideal en 16 threads, ya que esta sería la máxima cantidad que podrían estar corriendo en paralelo con el multithreading de los 8 cores y siguiéndole a eso ver poca mejora o incluso un peor rendimiento con mayores threads debido a posible overhead que el intercambio de estos generaría.

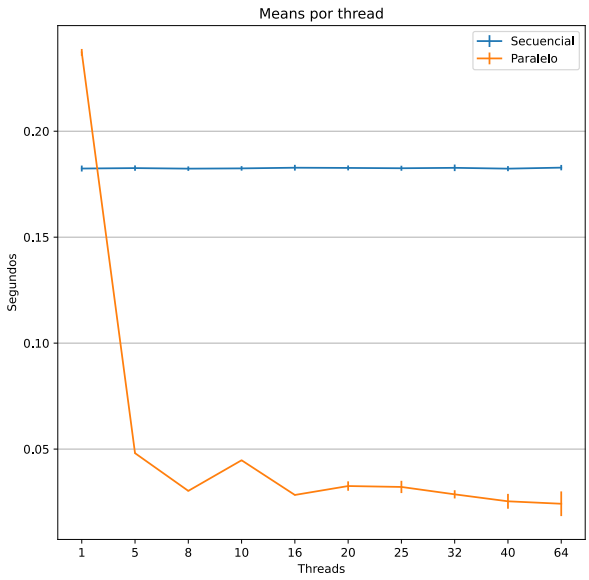

Figura 1: Tiempo promedio en segundos para cada cantidad de threads

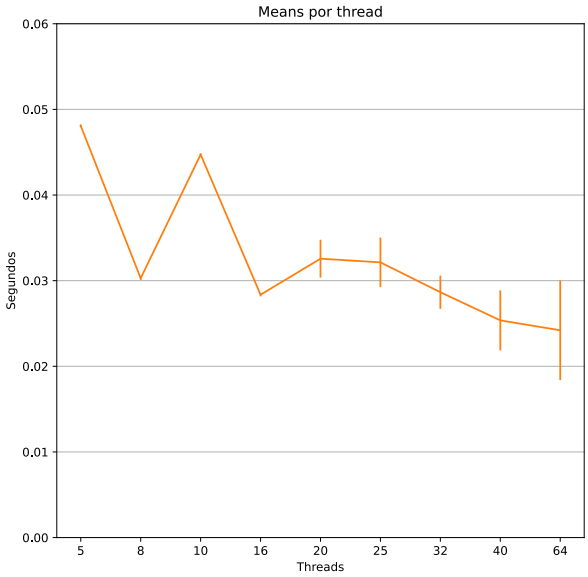

Figura 2: Desvío estándar del tiempo en segundos para cada cantidad de threads

Resultados

Los resultados observados para la media fueron los esperados, mostrando como aumentando la cantidad de threads aumenta el rendimiento en la corrida secuencial, superando ampliamente el rendimiento de la corrida secuencial para este tamaño de tablero. Esto se nota especialmente al pasar de 1 thread (el cual seria como hacer secuencial, pero con el overhead de crear el thread y usar una función más compleja con la lógica de paralelismo para resolver el problema) a 5, ya que se empiezan a paralelizar las verificaciones. Luego de 5 a 16 podemos observar una mejora más pequeña, mientras que al subir de 16 la cantidad de threads el rendimiento disminuye. Sin embargo, podemos ver que los tiempos de 32, 40 y 64 threads son ligeramente menores a los de 16.

Por último, observamos que la varianza de los tiempos aumenta rápidamente al subir la cantidad de threads por encima de los 16, siendo los valores entre 5 y 16 los que menor varianza mostraron. Esto puede ser que suceda debido a que 16 threads pueden correr en paralelo en la CPU en la cual se realizaron los experimentos, pero no más que eso, entonces al seguir aumentando la cantidad de threads empezamos a tener que cambiar de thread y empezamos a depender más del azar (si el thread que le toco la parte del tablero que contiene a la bandera contraria le toca ejecutar más tiempo vamos a encontrar la bandera más rápido), mientras que con menos threads todos corren a la vez, notar también que de vuelta con 1 thread tiene una mayor varianza parecida a la de secuencial ya que se comportan de la misma manera.

5.2. Experimento 2: Tiempo vs tamaño

Lo que queremos ver en este caso es como varía el tiempo de la búsqueda secuencial y paralela a medida que se aumentan las dimensiones del tablero.

Para este experimento correremos 100 veces la búsqueda de la bandera para distintos tamaños de tablero, lo haremos tanto para secuencial como paralelo (este último utilizando 16 threads, ya que por el experimento anterior, este valor era el que tenía menor promedio de tiempo entre los que tenían la menor varianza). Experimentaremos usando los tamaños 20, 40, 100, 200, 500, 1000 y 2000 siendo estos los tamaños tanto de x como de y (en otras palabras, todos los tableros serán cuadra-

dos). Ubicaremos la bandera siempre en la última posición (última fila y última columna) como en el experimento anterior, para probar el peor caso para ambos algoritmos.

Hipótesis

Lo que esperábamos ver en este experimento es que si bien aumentar el tamaño del tablero va a hacer más lenta la búsqueda (ya que agregamos más casillas para revisar), esperábamos que este efecto sea aún más notable en la búsqueda secuencial, ya que en paralelo aumenta $aumentoTamaño/cantJugadores$ lo que tiene que revisar cada jugador (como los 16 threads pueden ejecutar en paralelo el aumento del tiempo total será menor al de secuencial, ya que este aumento se distribuye entre los distintos threads) y adicionalmente, gracias a nuestra experimentación preliminar notamos que crear los threads tiene un overhead que no está presente en secuencial y que para tamaños de tableros muy chicos este overhead puede dominar el tiempo de la búsqueda haciendo que no valga la pena paralelizar en algunos casos.

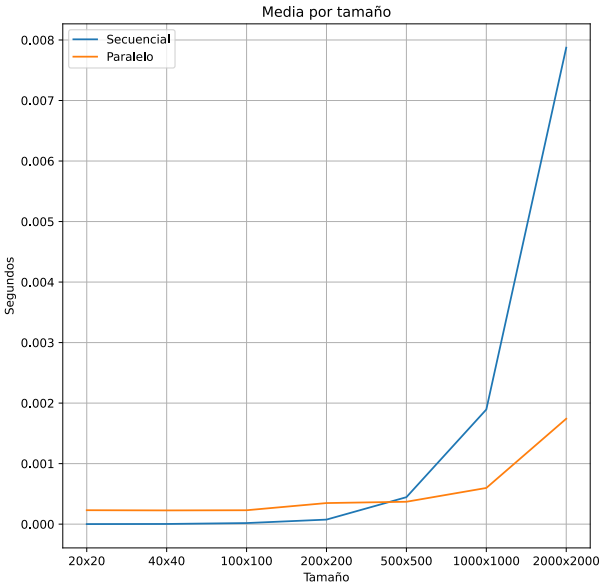

Figura 3: Tiempo promedio en segundos para cada tamaño de tablero

Resultados

Podemos ver como a medida que aumenta en tamaño del tablero los promedios de los tiempos de ambos aumentan, pero podemos notar como a partir de cierto punto el tiempo de secuencial empieza a crecer mucho más rápido que el de paralelo. Al rededor del tamaño 500×500 podemos ver como el rendimiento de secuencial se vuelve peor que el de paralelo (cuando para tamaños menores a este el rendimiento de secuencial era mejor que el de paralelo).

Estos resultados coinciden con la hipótesis propuesta, ya que paralelo siempre tuvo un overhead para todos los tamaños (incluso para tableros muy chicos, el cual en ese caso dominaba el tiempo de la búsqueda).

6. Conclusión

Luego de toda esta experimentación pudimos llegar a varias conclusiones. En primer lugar, con el experimento 1 pudimos ver como aumentar la cantidad de threads hasta el máximo que puede correr la CPU en paralelo (en este caso 16) aumenta substancialmente la eficiencia de los programas que pueden sacarle provecho a esta paralelización de tareas.
También con el experimento 2 pudimos concluir que a pesar de la eficiencia que genera la paralelización por sobre la secuencialidad hubieron dimensiones de tablero donde correrlo secuencial era más rápido que paralelo. Esto nos muestra que antes de intentar paralelizar una tarea debemos considerar si la tarea es paralelizable, la dificultad de paralelizarla (y mantener la sincronización en caso de que sea necesario) y si la tarea que debemos realizar realmente justifica paralelizar o si simplemente el overhead de la lógica extra y la inicialización de las estructuras de paralelización dominaran el tiempo de la resolución de la tarea, siendo una resolución secuencial una solución más eficiente a nuestro problema.

7. Anexo: Testeo del correcto funcionamiento del juego

Al momento de la implementación de las estrategias se colocaron distintos prints y asserts que brindaran información al respecto de la ejecución del código en distintas partes del mismo. Principalmente para verificar el correcto funcionamiento de las estrategias y la correcta sincronización entre los threads. La idea era, luego de que un jugador se moviera imprimir por pantalla; el jugador, la posición anterior y la nueva, el equipo al que pertenece, y en caso de que no sea posible el movimiento se imprime por pantalla un mensaje de movimiento cancelado. El assert verifica que el llamado a termino ronda sea hecho por el equipo correcto. La idea es que si la sincronización del sistema es correcta entonces se vera como los jugadores juegan en el orden correspondiente a cada estrategia. Para poder ver que no hay un orden de ejecución que haga que no se cumpla con la estrategia decidimos repetir estos testeos varias veces, hasta que nos parecieron suficientes.
A continuación mostramos algunas partes (no lo ponemos entero porque es muy largo y es siempre lo mismo) de las trazas de ejecución de algunas estrategias. Estos testeos se hicieron para todas pero acá se muestran solo algunas.

Estrategia Round Robin

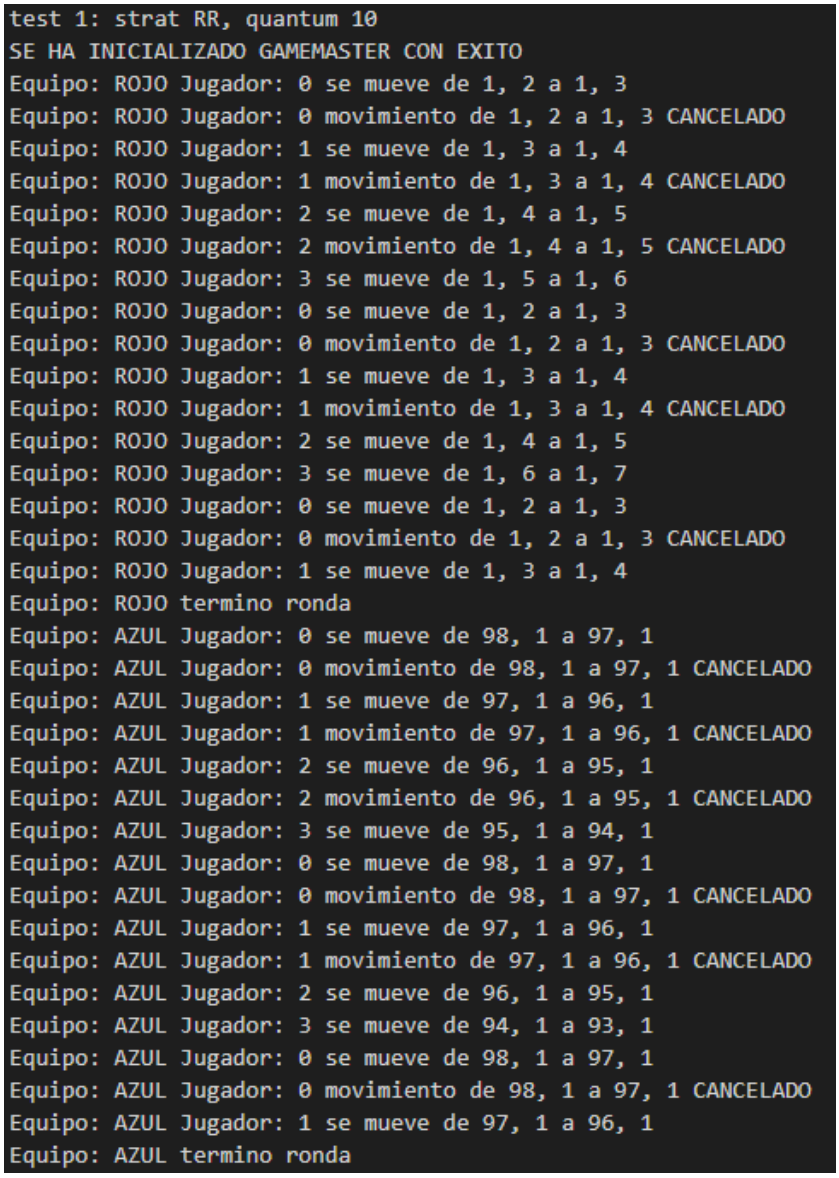

Figura 4: Principió de la traza de ejecución de la estrategia RR

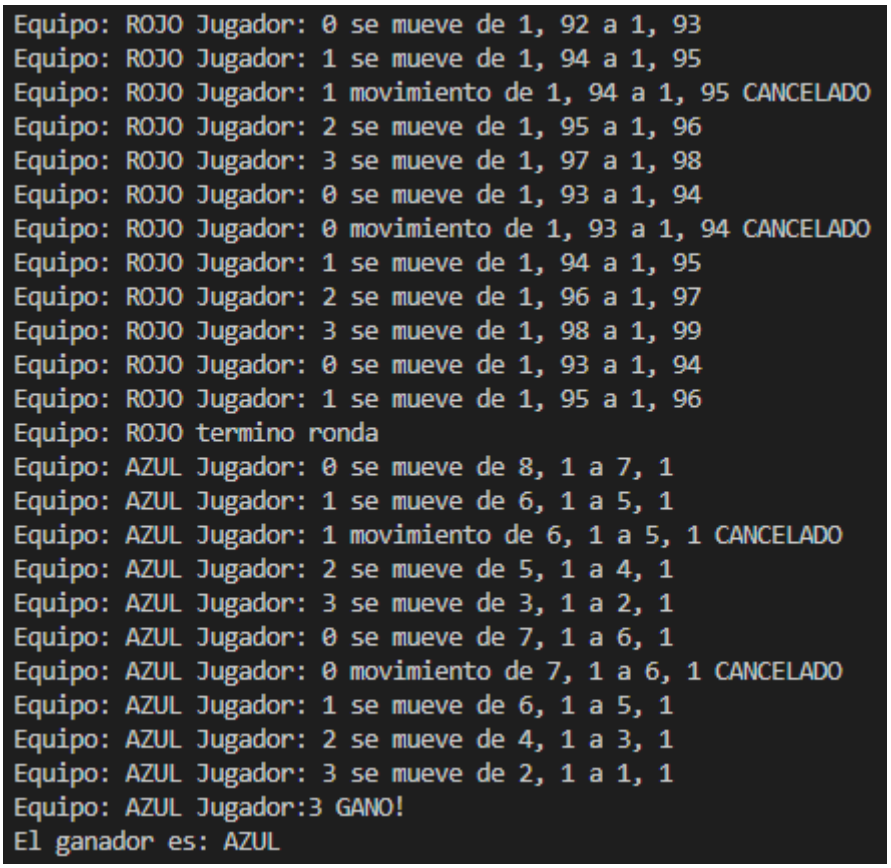

Figura 5: Final de la traza de ejecución de la estrategia RR

Se puede ver como se respeta los movimientos correspondientes con la estrategia Round Robin. Los jugadores se mueven en el orden correspondiente, se remarca que ningún jugador se mueve a una posición ocupada por otro (los movimientos cancelados), los turnos de los equipos son correctos (empieza el rojo y van uno y uno). También gana el equipo esperado, que se puede determinar por la disposición del tablero y la estrategia usada.

Estrategia Secuencial

En estas trazas de la estrategia secuencial podemos nuevamente ver como los movimientos se corresponden con esta estrategia. En este el orden en que los jugadores de cada equipo se mueve cambia en cada ronda por que depende de que thread llega primero, y eso era exactamente lo que se quería en esta estrategia. siguen habiendo algunos movimientos cancelados (cuando un jugador se quiere mover a una posición ocupada) y se siguen respetando las rondas.
Para las demás estrategias se hizo el mismo análisis y así se pudo concluir que se contaba con el correcto funcionamiento de cada estrategia.

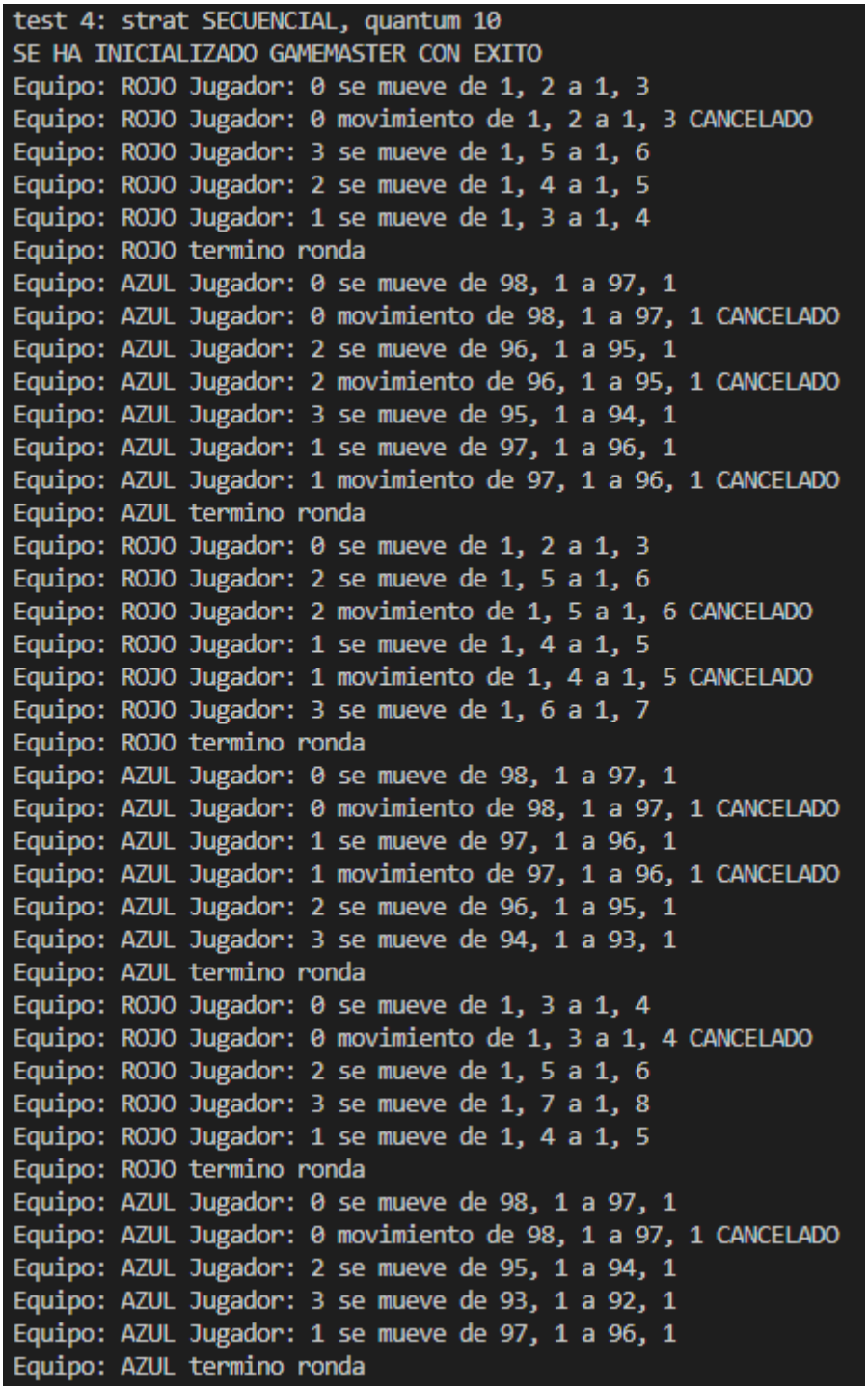

Figura 6: Principió de la traza de ejecución de la estrategia secuencial

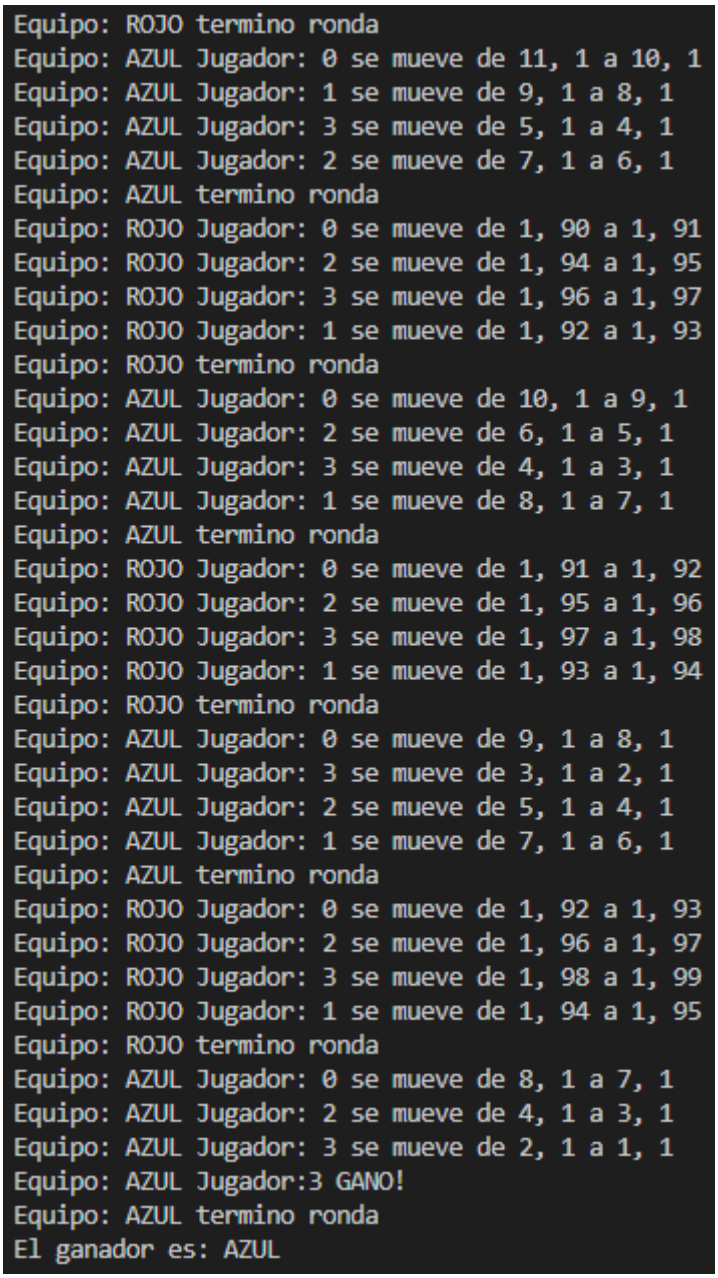

Figura 7: Final de la traza de ejecución de la estrategia secuencial