DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Sistemas Operativos

23 de marzo de 2023 — 2^{do} cuatrimestre 2022

Integrante	LU	Correo electrónico
Fiorino Santiago	516/20	fiorinosanti@gmail.com
Lebon Juan Pablo	228/21	juanpablolebon98@gmail.com
Ponce Juan Ignacio	420/21	juaniponce0@gmail.com
Valentini Nicolas	86/21	nicolasvalentini@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

Índice

1. Game Master **3**

2. Equipo **3**
2.1. Buscar bandera contraria 3
2.2. Jugador 4

3. Estrategias del Juego **4**
3.1. Estrategia Secuencial 4
3.2. Estrategia Round Robin 4
3.3. Estrategia Shortest 5
3.4. Estrategia Por Prioridades 5

4. Testing **5**

5. Resultados Empíricos **6**
5.1. Marco experimental 6
5.2. Resultados del experimento 6

Introducción

En este informe explicaremos el desarrollo y la implementación de los algoritmos realizados para programar el juego "Capturar la bandera", donde tenemos dos equipos que deben conseguir la bandera del equipo contrario. En este caso, queremos que el juego sea desarrollado utilizando técnicas de paralelismo vistos en la materia. A este fin, pensaremos a los jugadores de cada equipo como *threads*. Además, las estrategias que rigen cómo se mueven los jugadores estarán basadas en conceptos de *scheduling*.

La organización de este informe es la siguiente: las secciones 1 y 2 dan una explicación de alto nivel de la estructura de nuestro programa. Respectivamente, introducen las dos componentes principales del mismo: ***Game Master*** y ***Equipo***. La sección 3 da explicaciones más detalladas del funcionamiento de nuestro programa, explicando como estas dos componentes interactúan entre sí según la estrategia empleada. La sección 4 comprende la experimentación realizada como parte de este trabajo y los resultados de los mismos.

1. Game Master

La clase Game Master es la encargada de orquestar el juego. Sus responsabilidades incluyen iniciar la partida, determinar qué equipo está en turno, ordenarle a los jugadores que se muevan según la estrategia empleada, actualizar el estado del tablero luego de cada movimiento, y terminar la partida al haber un ganador o un empate.

La información pertinente al tablero (sus dimensiones, la ubicación de los jugadores y de las banderas) le es pasada a Game Master por el usuario. Lo primero que hace es chequear que los datos pasados por parámetro sean coherentes – que las dimensiones del tablero sean números positivos, que las posiciones de los jugadores y banderas se encuentren dentro del tablero, que haya misma cantidad de jugadores que de posiciones. Una vez pasado por los chequeos básicos, construye el tablero y luego agrega a los jugadores y a las banderas en sus respectivas posiciones dentro del mismo. Por último, inicializa los semáforos que utilizaremos para la sincronización de los turnos y los jugadores.

Como queremos que los equipos se turnen, empleamos semáforos que determinan quién es el equipo en turno en cada momento de la partida. En cada ronda, al semáforo correspondiente al próximo equipo a jugar le es asignada la cantidad de movimientos que debe realizar un equipo (según la estrategia). Una vez empezado el turno, cada intento de movimiento le resta una unidad al valor del semáforo. Contamos, además, con una variable que nos permite consultar la cantidad de movimientos restantes que tiene un equipo en un turno.

Como buscamos que los jugadores puedan moverse independientemente de sus compañeros, optamos por usar vectores de semáforos, y habrá uno para cada equipo. A cada jugador le corresponderá el semáforo que esté en el índice de su número de jugador. La idea es tener una estructura universal que alcance para implementar todas las estrategias; dependiendo de la estrategia elegida, el Game Master determinará cuáles semáforos de estos vectores activar, cuántas veces, y en qué orden, para poder llevarla a cabo.

En el trascurso del juego, el Game Master se encarga de mantener y actualizar todas las estructuras mencionadas.

El Game Master, además, se encarga de terminar la partida antes de que algún equipo encuentre la bandera contraria, si lo considera necesario. Esto podría pasar, por ejemplo, si todos los jugadores están trabados (es decir, si las posiciones a las que desean moverse están ocupadas por otro jugador). Para evitar que la partida se cuelgue, al final de cada ronda el Game Master determina si esto está ocurriendo y en tal caso termina la partida en empate.

2. Equipo

En esta clase se encuentran las funciones que definen la lógica de cada jugador; es aquí donde definimos qué hace cada jugador antes y después de recibir ordenes del Game Master. Para esto, contamos con los siguientes métodos:

2.1. Buscar bandera contraria

Aquí definimos el proceso mediante el cual cada equipo encuentra la posición de la bandera contraria antes de empezar la partida. A cada jugador de un equipo le asignamos una sección del tablero para recorrer en su búsqueda; la idea es que estas búsquedas se realicen en paralelo por cuestiones de eficiencia temporal.

Si un jugador termina de recorrer su sección del tablero y no encontró la bandera, se queda esperando a que otro jugador la encuentre. Cuando un jugador logra encontrar la bandera, le avisa a los otros jugadores que pueden dejar de esperar, y el equipo empieza a jugar.

2.2. Jugador

Esta es la función que cada thread ejecuta – define la lógica que todo jugador seguirá en bucle durante toda la partida. En general, cada jugador espera a que el Game Master le permita realizar un intento de movimiento y luego lo efectúa. Hablamos de "intentos de movimiento" porque si un jugador busca moverse a una posición que está ocupada por otro jugador, o que se pasa de rango del tablero, no permitiremos que lo realice. Qué hace después de intentar moverse depende de la estrategia y está explicado en la siguiente sección.

3. Estrategias del Juego

En nuestra implementación, contamos con 4 tipos de estrategias que implementamos para el funcionamiento del juego. Estas son: **Secuencial**, **Round Robin** (RR), **Shortest Distance First**, y **Por Prioridades**.

3.1. Estrategia Secuencial

Esta estrategia se basa en darle, a cada jugador del equipo en turno, la oportunidad de moverse una vez. Dicho turno termina solo cuando todos los jugadores del equipo hayan realizado un intento de movimiento.

- **Game Master**: Como empieza jugando el equipo rojo, los semáforos de su vector se iniciarán en 1 para permitirle a cada jugador que realice un intento de movimiento. En cambio, los del equipo azul se inicializarán todos en 0, para que esperen su turno. Tanto el semáforo del próximo turno como el contador de movimientos restantes se inicializan en la cantidad de jugadores por equipo. Al terminar una ronda, Game Master restaura estos valores para el próximo equipo a jugar.

  El manejo de empate se lleva a cabo al final de cada ronda, y en esta estrategia es el siguiente: Si al final de una ronda todos los jugadores de ambos equipos están trabados, el juego es terminado, el resultado declarado como empate.

- **Equipo**: Al recibir el alta de Game Master, cada jugador del equipo en turno realiza un intento de movimiento. En esta estrategia los jugadores no hacen nada más luego de moverse; solo esperan a que Game Master les vuelva a permitir moverse en el turno siguiente de su equipo. Si un jugador no pudo efectuar dicho movimiento, será etiquetado como "trabado". Esta etiqueta puede cambiar si logra moverse en una ronda futura.

3.2. Estrategia Round Robin

Esta estrategia cuenta con un *quantum*. Similarmente a su función en *scheduling*, aquí este número determina cuántos movimientos puede realizar un equipo en cada turno. Análogamente a la lógica Round Robin en scheduling, los jugadores se moverán en un orden predeterminado hasta que se agote el quantum. Si al moverse el último jugador no se agotó el quantum del equipo, se volverán a mover los jugadores en el mismo orden que antes.

- **Game Master**: Aquí, a diferencia de la estrategia anterior, queremos que los jugadores se muevan en un orden específico. A este fin, Game Master inicialmente solo le permite jugar al primer jugador del equipo rojo. Tanto el semáforo del próximo turno como el contador de movimientos restantes se inicializan en el valor del quantum.

  Al terminar una ronda, Game Master restaura estos valores para el próximo equipo a jugar. El testeo de empate es idéntico al de la estrategia anterior; preguntamos si todos los jugadores de ambos equipos están trabados.

- **Equipo**: A diferencia de la estrategia secuencial, los jugadores deben aquí tomar acciones adicionales al terminar su intento de movimiento: cada jugador le notifica al jugador siguiente que debe moverse,

si es que el equipo sigue teniendo intentos de movimientos restantes. Es decir que a diferencia de la estrategia anterior, donde *todos* los jugadores reciben el alta para moverse al principio de cada ronda, aquí lo recibe solo uno, y este se encarga de empezar la cadena de notificaciones a los otros jugadores. Esto asegura el orden entre los jugadores.

3.3. Estrategia Shortest

En esta estrategia solo podrá moverse el jugador de cada equipo que se encuentre más cerca a la bandera contraria, y este podrá moverse solo una vez por turno.

- **Game Master**: Tanto el semáforo del próximo turno como el contador de movimientos restantes lo iniciamos en 1. Todos los jugadores empiezan con sus respectivos semáforos en 0. En cada ronda, le indicamos al jugador del equipo en turno más cercano a la bandera contraria que se mueva una vez.

  Como en esta estrategia solo se mueve un jugador de cada equipo en toda la partida, no seria correcto pedir que *todos* los jugadores estén trabados para declarar un empate. En cambio, aquí el Game Master solo pregunta si los dos jugadores más cercanos a la bandera contraria pudieron moverse en su último turno, terminando la partida en caso negativo.

- **Equipo**: Similarmente a la estrategia secuencial, los jugadores no deben hacer nada adicional luego de moverse; el equipo solo debe esperar a que el jugador más cercano a la bandera (que será siempre el mismo) reciba el alta del Game Master para moverse.

3.4. Estrategia Por Prioridades

Esta es la estrategia diseñada por nosotros, y puede pensarse como una modificación de "Shortest" con *aging*. Se mueve un jugador por turno, y en el primer turno se prioriza al jugador más cercano a la bandera. De ahí en adelante, cada jugador que no se haya movido en algún turno verá su prioridad incrementada al final del turno. Por otro lado, el jugador que se movió en dicho turno verá su prioridad decrementada. Así, cuanto más tiempo haya pasado desde el último movimiento de algún jugador, más prioridad se le dará a este al determinar quien se moverá en el turno siguiente. Esto lleva a que cada jugador tenga la oportunidad de participar del juego, a diferencia de "Shortest".

- **Game Master**: Para esta estrategia utilizaremos la misma inicialización usada para la estrategia Shortest; todos los jugadores empiezan con su respectivo semáforo en 0, y además los semáforos de turno y el contador de movimientos restantes empiezan en 1. En cada ronda, le indicamos al jugador de mayor prioridad del equipo en turno que se mueva una vez.

  El testeo de empate es idéntico al de Secuencial y Round Robin, ya que se espera que todos los jugadores puedan moverse durante la partida.

- **Equipo**: Como siempre, los jugadores esperan a que el Game Master les permita realizar un intento de movimiento. En esta estrategia, como explicamos antes, un jugador debe además actualizar las prioridades de todos los jugadores de su equipo al moverse. En particular, debe decrementar la suya y aumentar las del resto, para implementar el *aging*. Una vez realizada esta actualización, identifica el nuevo jugador prioritario – este será el que Game Master ordene moverse en el turno siguiente del equipo.

4. Testing

Para poner a prueba tanto el código implementado como el estado del juego agregamos varias lineas de texto que se irán imprimiendo en pantalla a lo largo de la partida para poder seguir el funcionamiento del juego en todo momento.

Para la estrategia secuencial, nuestro criterio para saber que funcionaba fue ver que todos los jugadores de un equipo se hayan movido antes de que se moviera un jugador del equipo contrario. Para Round Robin, nos fijamos que se movieran la cantidad de veces que se definió en el quantum y que además siguieran el orden establecido. Luego para Shortest pedimos que solo se moviera un jugador por turno y que este fuera el más cercano a la bandera contraria. Y por último, en el de prioridades, había que ver que además de que

solo se moviera uno solo por turno y fuera el más prioritario, el aging funcionara correctamente, es decir, que luego de algunos turnos, existiera un jugador diferente que se moviera.

En general, también mandamos una notificación a pantalla cuando algún equipo encuentra la bandera contraria, para verificar que la partida termine en consecuencia.

5. Resultados Empíricos

Como mencionamos anteriormente, al momento de buscar la bandera contraria hacemos uso de los *threads*. En esta sección vamos a generar un experimento para observar si, tal como esperamos, separar la búsqueda en *threads* resulta más eficiente que no separarla, y que un solo proceso haga una búsqueda secuencial.

5.1. Marco experimental

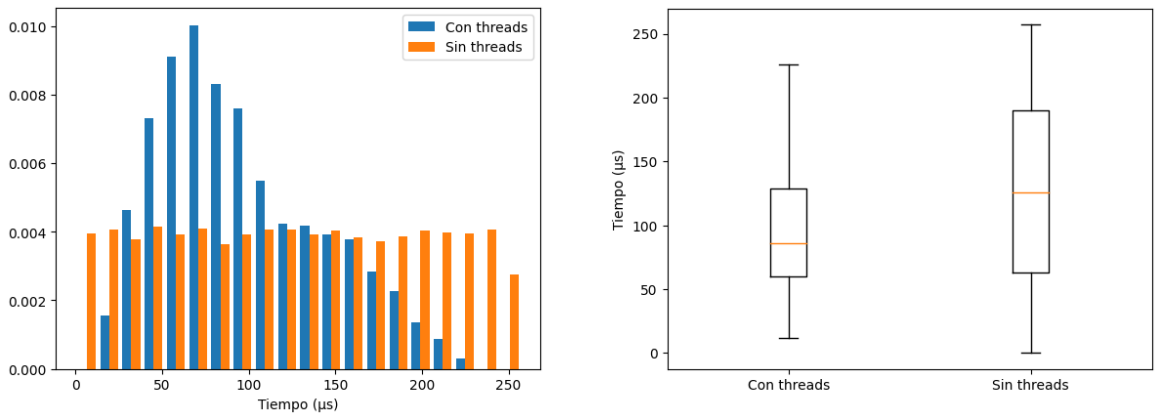
El experimento consistirá en efectuar 20,000 búsquedas de banderas sobre un tablero de 100×100, y medir el tiempo tomado. La mitad de las búsquedas serán efectuadas de forma secuencial por un solo proceso, mientras que la otra mitad serán dividiendo el trabajo en *threads*.

Para medir los tiempos de ejecución, usamos la biblioteca *Chrono* de *C++*. En el caso de la búsqueda de un solo proceso, empezamos a medir el tiempo justo antes del ciclo que empieza la búsqueda secuencial. En el caso de la búsqueda con *threads*, empezamos a medir el tiempo justo antes del ciclo que crea los *threads*. En ambos casos, apenas se encuentra la bandera, se para el cronómetro.

Si el experimento lo corremos dejando la bandera siempre en el mismo lugar, la búsqueda de un solo proceso tardará siempre lo mismo (aproximadamente). Si la bandera está siempre en la primera celda del tablero, estaríamos beneficiando la búsqueda de un solo proceso, ya que este la encontrará inmediatamente. Siguiendo la misma lógica, si la bandera está siempre al final estaríamos perjudicando a la búsqueda de un solo proceso, ya que éste es el peor caso de la búsqueda secuencial. Entonces, para que el experimento sea justo, cada vez que se efectúa una búsqueda de bandera, se le asigna a la bandera una posición aleatoria dentro del tablero.

5.2. Resultados del experimento

Una vez corrido el experimento, eliminamos el 2 % más grande y más chico de los datos para descartar posibles *outliers*, y obtuvimos los siguientes resultados:



(a) Histograma

(b) Boxplot

Figura 1: Resultados obtenidos

En la Figura 1 (a) podemos observar que el tiempo de búsqueda de la bandera sin usar *threads* sigue una distribución uniforme con valores entre 10 y 250. Esto es lo que esperábamos, ya que el tiempo que tarda es directamente proporcional a la cantidad de celdas que tiene que visitar. De los datos obtenidos podemos deducir que cuando la bandera está en la primera celda, la búsqueda sin *threads* tarda aproximadamente 10 microsegundos, y cuando está en la última, la búsqueda tarda 250. En la misma Figura, podemos observar que la distribución de los tiempos de búsqueda de la bandera usando *threads* es muy diferente a una uniforme.

Este también era el comportamiento esperado, ya que estas búsquedas no dependen tanto de la posición de la bandera, sino que dependen fuertemente del orden de ejecución de los *threads*. Podemos observar que la distribución tiene un pico elevado en los 60 microsegundos, y que toma en la mayoría de los casos un tiempo entre 50 y 80 microsegundos; muy pocas veces llega a valores mayores a los 150 microsegundos.

En la Figura 1 (b) podemos ver que la media de los tiempos de ejecución usando *threads* es notablemente menor a la media de los tiempos de ejecución sin usarlos. Incluso podemos observar que la media del tiempo tomado en la ejecución sin *threads* está en el precentil 75 de los tiempos tomados con *threads*.

En conclusión, el resultado del experimento fue el esperado: **Separar la búsqueda en *threads* reduce notablemente el tiempo de búsqueda.**