

Laberinto con threads y forks
Proyecto 1

Principios de Sistemas Operativos
Escuela de Computación
Tecnológico de Costa Rica

Estudiantes:

ANDREW J. GUTIERREZ CASTRO - 2019068322

KEVIN M. FALLAS ALVARADO - 2019057752

ESTEBAN A. MADRIGAL MARÍN - 2018154123

JOSUE A. ROJAS VEGA- 2019042245

20 de setiembre de 2021

Índice

| | |
|--|-----------|
| Introducción | 2 |
| Estrategia de solución: | 2 |
| Inicialización general | 2 |
| Preparación general | 3 |
| Proceso con Threads | 4 |
| Proceso con Forks | 5 |
| Estrategia general | 5 |
| Análisis de Resultados | 9 |
| Lecciones aprendidas | 9 |
| Casos de pruebas | 10 |
| Caso de prueba 1 | 10 |
| Caso de prueba 2 | 11 |
| Caso de prueba 3 | 12 |
| Caso de prueba 4 | 13 |
| Comparación | 14 |
| Comparación de resultados | 14 |
| Comparación entre Threads y Forks | 15 |
| Manual de usuario | 16 |
| Bitácora | 18 |
| Apéndice | 20 |
| Ejemplo de archivos con los laberintos | 20 |
| Ejemplo de ejecución de un programa | 21 |

Introducción

Laberinto con threads y forks es la primera tarea para el curso de Principios de Sistemas Operativos. Dicha tarea, tiene como finalidad simular un laberinto con dos versiones de solución. La primera versión crea la solución con threads, los cuales cada vez que haya una posibilidad de cambio de dirección en el laberinto un nuevo thread toma ese camino. La segunda versión crea la solución de manera similar a la primera, cambiando los threads por forks. Ambas soluciones son medidas con el fin de determinar cual solución es mejor, si la implementada con threads o la implementada con forks.

En esta documentación se presenta la solución de la estrategia utilizada para solución del laberinto, tanto para los threads como para los forks. Se analizan los resultados, los casos de prueba realizados y la comparación con detalles técnicos, rendimiento y usabilidad. Además, se presenta el manual de usuario para ejecutar el programa.

Estrategia de solución:

Inicialización general

Primeramente desde el `main` llamamos a la función llamada `run_threads_and_fork_solvers`, desde la cual empieza la preparación para poder empezar con la estrategia para resolver el laberinto mediante Threads y Forks como se visualiza en el siguiente código.

```

1  int main(int argc, char const *argv[])
2  {
3      char *filename =
4          "../files/lab2.txt";
5
6      run_threads_and_fork_solvers( filename );
7
8      return EXIT_SUCCESS;
9  }
10
11 void run_threads_and_fork_solvers(char *filename) {
12
13     assemble();
14
15     eval_solver(filename, FORKS_MODE);
16
17     printf("\n\nRun threads mode ...PRESS ENTER...\n\n");
18     while ( getchar() != '\n' );
19
20     eval_solver(filename, THREADS_MODE);
21
22     show_results();
23
24     disassemble();
25 }
```

Código 1: `main` y `run_threads_and_fork_solvers`

Preparación general

Después de esto entonces es necesario cargar el laberinto a memoria para poder empezar con la estrategia de solución, como se evidencia en el siguiente código.

```

1 void eval_solver
2 (
3     char *filename,
4     int strategy_mode
5 )
6 {
7     char *strategy_name = strategy_mode == THREADS_STRATEGY ? "THREADS" : "
    FORKS";
8
9     // load maze from file
10    original_maze = load_maze(filename, strategy_mode);
11
12    // set start time
13    clock_t begin = clock();
14
15    // call the maze solve function
16    switch(strategy_mode)
17    {
18        case THREADS_STRATEGY:
19            solve_with_threads(
20                DEFAULT_START_DIRECTION,
21                0,
22                0,
23                0,
24                0,
25                0,
26                WAIT
27            );
28            break;
29
30        case FORKS_STRATEGY:
31            solve_with_forks(
32                DEFAULT_START_DIRECTION,
33                0,
34                0,
35                0,
36                0,
37                0,
38                WAIT
39            );
40            break;
41
42        default:
43            printf("Unknown strategy mode");
44    }
45    // finish taking the time
46    clock_t end = clock();
47
48    // convert cloc ticks to seconds
49    double time_spent = (double)(end - begin) / (double) (CLOCKS_PER_SEC);
50

```

```

51 // show results
52 printf("\n[%s] -> %3.5f seconds", strategy_name, time_spent);
53 }

```

Código 2: eval_solver

En el código anterior, en la línea 10 se ve como se carga el laberinto, en la 13 inicia el reloj para medir la duración, y después mediante el **switch** decidimos si correr la estrategia de forks o de threads, la cual veremos de manera detallada más adelante, también es importante recalcar como en las líneas 26 y 38 enviamos una bandera como **WAIT** esto es debido a que en ambas estrategias el primer fork o thread debe hacer algo diferente a los demás, esto también se vera detallado mas adelante, finalmente se muestran los resultados.

Proceso con Threads

Ahora bien para la estrategia de los Threads, primeramente se crea un **Walker**, que es la estructura que maneja todos los atributos necesarios para el movimiento de los threads, como se ve a continuación.

```

1 typedef struct walker_unit {
2     char    direction;
3     int     start_col;
4     int     start_row;
5     int     current_col;
6     int     current_row;
7     int     steps;
8     char    color;
9 } walker_unit;
10
11 typedef struct walker_unit* Walker;

```

Código 3: Walker

Seguidamente se crea un thread con su función correspondiente y como parámetro nuestro **Walker** antes mencionado, todo esto se ve evidenciado en el siguiente código.

```

1 pthread_t solve_with_threads(char direction, int start_row, int start_col,
2                               int current_row, int current_column, int steps, int waiting_flag)
3 {
4     pthread_t child_thread;
5
6     // create the parent walker
7     Walker current_walker = build_walker(direction, start_row, start_col,
8     current_row, current_column, steps);
9
10    // create the parent thread, invoking walk function
11    int thread_id = pthread_create(&child_thread, NULL, walk_with_threads,
12    (void*) current_walker);
13
14    // wait until his child thread end
15    if (waiting_flag == WAIT)
16        pthread_join(child_thread, NULL);
17
18    return child_thread;

```

16 }

Código 4: solve_with_threads

Ahora podemos notar en la línea 12 la importancia de la bandera que mencionamos anteriormente, si lo que se está generando es el primer thread, entonces debemos esperar que el concluya, porque sino nuestro programa termina, ahora si el primer thread muere, debemos esperar a los threads hijos que se generaron, y así hasta que todos los threads mueran entonces termina el programa.

Proceso con Forks

Ahora bien para la estrategia de los forks, también se debe crear un **Walker**, después creamos un fork y verificamos si es un hijo, entonces corremos la lógica de caminar por el laberinto.

```

1 pid_t solve_with_forks(char direction, int start_row, int start_col, int
    current_row, int current_column, int steps, int waiting_flag)
2 {
3     // create the parent walker
4     Walker current_walker =
5         build_walker(direction, start_row, start_col, current_row,
        current_column, steps);
6
7     pid_t pid = fork();
8     if (pid == 0) // child
9     {
10         walk_with_forks(current_walker);
11     }
12     else if (pid > 0 && waiting_flag == WAIT) // parent
13     {
14         while (wait(NULL) > 0);
15     }
16 }
```

Código 5: solve_with_forks

Aquí también vemos la bandera, en el caso de que sea el fork o proceso original por decirlo así, este debe esperar a que todos sus procesos hijos terminen, esto se consigue con la línea 14.

Estrategia general

La estrategia de solución por parte de los threads y de los forks es exactamente la misma excepto por un par de líneas de código las cuales aclararemos adelante, dicha lógica se encuentra en las funciones llamadas **walk_with_threads** para el proceso con threads y **walk_with_forks** para el proceso con forks, en ambas primeramente se obtiene la cantidad total de las filas y columnas del laberinto y además la posición actual del **Walker** que estamos haciendo caminar. Para poder visualizar esto adelante se muestran extractos de la función **walk_with_threads**.

```

1 void *walk_with_threads(void *_walker)
2 \{
3     // void pointer casting for current walker
```

```

4     Walker current_walker = (Walker) _walker;
5
6     int total_rows = original_maze->height;
7     int total_columns = original_maze->width;
8
9     int row = current_walker->current_row;
10    int column = current_walker->current_col;
11
12    int is_death = 0;

```

Código 6: Primera parte de la función walk_with_threads

En el caso de los threads es necesario guardar los identificadores de los threads que se crean mientras el thread padre este vivo, por lo tanto tenemos lo siguiente.

```

1     pthread_t pid_list[original_maze->height*original_maze->width*2];
2     int pid_counter = 0;

```

Código 7: Almacenar identificadores de threads hijos

Como se puede ver en el código 6 tenemos la bandera llamada `is_death`, la cual nos permite verificar que mientras nuestro `Walker` no haya muerto, entonces pintamos el camino y verificamos si ya encontró la salida, cabe volver a recalcar que esta misma estrategia es que se utiliza en forks.

```

1     while(!is_death) {
2
3         paint_path(current_walker->color, row, column);
4
5         if(is_at_finish(row, column))
6         {
7             handle_winner(current_walker);
8         }

```

Código 8: Segunda parte de la función walk_with_threads

Después verificamos que por cada posible dirección, si no esta en un borde, si no tiene una pared y no es la dirección original del `Walker`, entonces se debe crear otro thread o fork para que siga esa nueva ruta, por lo que volvemos a llamar a la función `solve_with_threads` en el caso de estar trabajando con threads, si fuera con forks entonces llamamos a la función `solve_with_forks` las cuales tienen en común la creación de un `Walker` nuevo y después todo el proceso que vimos anteriormente de cada una.

```

1     int walker_map[][2] = // {<dimension constraint>, <proposed
    direction>}
2     {
3         { row != 0,                UP    },
4         { column != 0,             LEFT  },
5         { row != total_rows-1,     DOWN  },
6         { column != total_columns-1, RIGHT }
7     };
8
9     // for every possible direction
10    for (int i = 0; i < MOVEMENT_AMOUNT; i++)
11    {
12        // allow us to walk by w,a,s,d (UP, LEFT, DOWN, RIGHT) into the
    maze

```

```

13         int row_shifted = row + ROW_MOVEMENT[i];
14         int col_shifted = column + COL_MOVEMENT[i];
15
16         // constraint evaluation, free space in the proposed direction,
17         has a not equal direction
18         if (
19             walker_map[i][DIM_CONSTRAINT] &&
20             original_maze->map[row_shifted][col_shifted] == FREE_SPACE
21             &&
22             current_walker->direction != walker_map[i][
23             PROPOSED_DIRECTION]
24         )
25         {
26             pthread_t tid =
27                 solve_with_threads(
28                     walker_map[i][PROPOSED_DIRECTION],
29                     row, column,
30                     row_shifted, col_shifted,
31                     current_walker->steps,
32                     NO_WAIT
33                 );
34             pid_list[ pid_counter++ ] = tid;
35         }

```

Código 9: Tercera parte de la función walk_with_threads

En el código anterior la única diferencia entre las dos estrategias es que en la línea 23 y 32, obtenemos y almacenamos el id del thread que se creo, esto en la estrategia con forks no es necesario. Después debemos verificar si nuestro Walker debe morir, si no debe morir entonces damos otro paso en la dirección que tenia.

```

1         // check if the thread should die
2         is_death = should_die(current_walker->direction, row, column);
3
4         if(!is_death)
5         {
6             // continue walking in the maze
7             take_a_step(current_walker);
8
9             row = current_walker->current_row;
10            column = current_walker->current_col;
11        }
12        else
13        {
14            for(int tid = 0 ; tid < pid_counter ; tid++)
15            {
16                pthread_join(pid_list[tid], NULL);
17            }
18        }

```

Código 10: Cuarta parte de la función walk_with_threads

Pero si debe morir, en el caso de los threads, es necesario entonces indicar que debemos esperar a que todos los threads hijos que se generaron terminen, de esta manera, hasta que todos

los threads terminen, termina el programa. Por esta razón guardamos los identificadores en una lista, en el siguiente diagrama se ilustra lo antes mencionado.

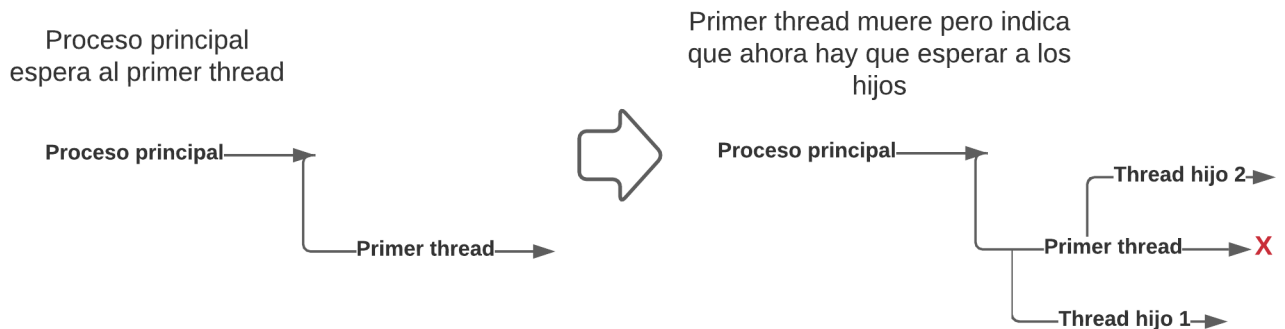


Figura 1

Lógica de creación y muerte de threads.

Con forks es un poco diferente, en el caso de los forks cuando un fork debe morir, en realidad se queda esperando a que sus hijos mueran, y los hijos hacen lo mismo, y en el momento en el que no haya que esperar a nadie entonces todos los que se quedaron esperando dejan de esperar y el programa termina, esto se ilustra en el siguiente diagrama.

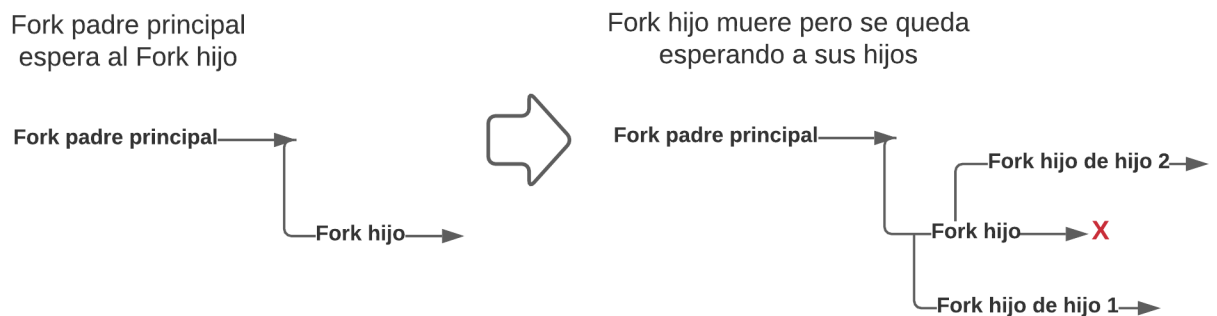


Figura 2

Lógica de creación y muerte de forks.

Por esta razón cuando un fork muere, debe esperar a sus hijos como se muestra en el siguiente código en la línea 13.

```

1      // check if the forks should die
2      is_death = should_die(current_walker->direction, row, column);
3
4      if(!is_death)
5      {
6          // continue walking in the maze

```

```

7         take_a_step(current_walker);
8
9         row = current_walker->current_row;
10        column = current_walker->current_col;
11    }
12    else {
13        while (wait(NULL) > 0);
14        exit(0); // notify to the parent process
15    }

```

Código 11: Cuarta parte de la funcion walk_with_forks

Y de esta manera conseguimos, mediante threads y forks, ir buscando la salida por las diferentes rutas que tiene el laberinto.

Análisis de Resultados

| Actividades y Tareas | Porcentaje de completitud |
|--------------------------------------|---------------------------|
| Leer y cargar el laberinto del .txt | 100 % |
| Contar el tiempo de duración | 100 % |
| Crear un thread nuevo | 100 % |
| Crear un fork nuevo | 100 % |
| Mostrar el laberinto con sus caminos | 100 % |
| Mostrar cada proceso con un color | 100 % |
| Verificar movimientos disponibles | 100 % |
| Verificar si debe morir | 100 % |
| Verificar si encontró salida | 100 % |
| Caminar por el laberinto | 100 % |
| Mostrar resultados finales | 100 % |

Lecciones aprendidas

1. Al iniciar el código no teníamos muy claro como sería la estrategia de solución, y por empezar a programarla antes de planearla se generó un código desordenado, afortunadamente Josué logró ordenarlo y simplificarlo de la mejor manera, generando un mejor entendimiento del código.
2. Otro percance fue que por la misma razón, de empezar a programar sin plantear una solución, tuvimos que afrontar varios problemas de lógica los cuales afortunadamente logramos solucionar de manera grupal.
3. Cuando creímos que la parte de los threads estaba terminada, nos dimos cuenta que a la hora de recorrer el laberinto, y habían dos caminos, el programa primero investigaba el primer camino, y cuando ya no había más seguía con el segundo, como si el

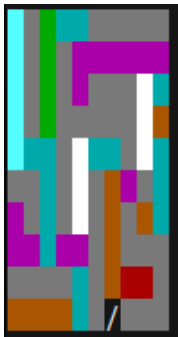


Figura 4
Resultado Prueba 1 con Forks.

Caso de prueba 2

Con el siguiente laberinto de 10x7, para verificar que con diferentes dimensiones el laberinto carga y se ejecuta bien. Además, se quiere ver si está validado si un thread no llena un espacio que coincide con un borde si se unieran. Y por último, se desea conocer



Prueba 2

La cual genera el siguiente resultado exitoso para los threads.

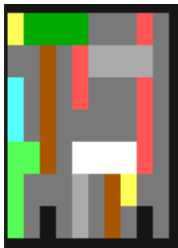


Figura 5
Resultado Prueba 2 con Threads.

Y la siguiente corrida para los forks, también es exitosa.

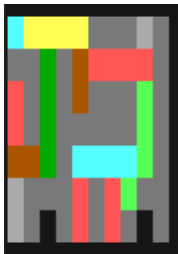
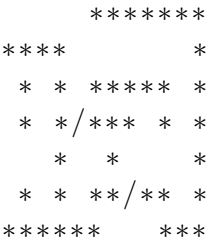


Figura 6
Resultado Prueba 2 con Forks.

Caso de prueba 3

Con el siguiente laberinto de 12x7, para verificar que cumple con dos metas y estas metas no quedan en los bordes.



Prueba 3

La cual genera el siguiente resultado exitoso para los threads.

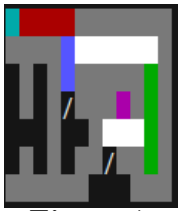


Figura 7
Resultado Prueba 3 con Threads.

Y la siguiente corrida para los forks, también es exitosa.

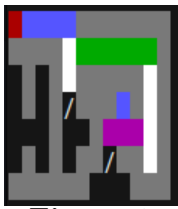


Figura 8
Resultado Prueba 3 con Forks.

Caso de prueba 4

Con el siguiente laberinto de 8x7, está lleno con espacios en blanco y sin paredes, con solo una meta.

/

Prueba 4

La cual genera el siguiente resultado exitoso para los threads.



Figura 9

Resultado Prueba 4 con Threads.

Y la siguiente corrida para los forks, también es exitosa.

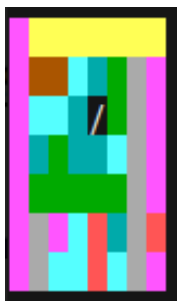


Figura 10

Resultado Prueba 4 con Forks.

Comparación

Uno de los objetivos de este trabajo también es comparar los resultados de las dos estrategias de solución, por lo que primero se presentarán los resultados obtenidos y luego un pequeño análisis sobre las comparaciones entre los forks y threads.

Comparación de resultados

El programa al terminar la ejecución de la solución tanto para los threads como para los forks, imprime la duración de la ejecución de manera que se pueda tener una alusión a al rendimiento y tiempos de la búsqueda de soluciones por medio de los dos métodos.

Los siguientes resultados son ejecuciones del laberinto 1 y del laberinto 2, los cuales se pueden consultar en el apéndice.



Figura 11

Comparación con Laberinto 1.



Figura 12

Comparación con Laberinto 2.

Para los cuales se realizó una comparación ilustrativa de los promedios para la única solución del Laberinto 1 y las dos soluciones del Laberinto 2.

| Lab1.txt Solución 1 | | Lab2.txt Solución 1 | | Lab2.txt Solución 2 | |
|---------------------|----------------|---------------------|----------------|---------------------|----------------|
| Threads | Forks | Threads | Forks | Threads | Forks |
| 0.01297 | 0.004823 | 0.00877 | 0.004814 | 0.00922 | 0.005417 |
| 0.00407 | 0.004358 | 0.007827 | 0.005125 | 0.008645 | 0.005637 |
| 0.003528 | 0.004131 | 0.007133 | 0.005762 | 0.007952 | 0.006204 |
| 0.004406 | 0.004228 | 0.006866 | 0.005656 | 0.007102 | 0.006273 |
| 0.004575 | 0.004291 | 0.016762 | 0.013507 | 0.017834 | 0.01604 |
| 0.004716 | 0.004415 | 0.008525 | 0.005125 | 0.008677 | 0.005675 |
| 0.004067 | 0.004203 | 0.005261 | 0.005106 | 0.00648 | 0.00671 |
| 0.004489 | 0.004194 | 0.007465 | 0.005204 | 0.008479 | 0.005644 |
| 0.004485 | 0.004414 | 0.007244 | 0.005599 | 0.00803 | 0.006024 |
| 0.004223 | 0.004311 | 0.007145 | 0.005368 | 0.007639 | 0.005932 |
| 0.004888 | 0.004828 | 0.010134 | 0.005022 | 0.006904 | 0.006035 |
| 0.004119 | 0.004215 | 0.007296 | 0.005035 | 0.007432 | 0.005406 |
| 0.00441 | 0.004229 | 0.004092 | 0.005333 | 0.008529 | 0.005395 |
| 0.004277 | 0.004332 | 0.007129 | 0.00561 | 0.007758 | 0.006098 |
| 0.004463 | 0.013884 | 0.016644 | 0.012006 | 0.0072 | 0.005579 |
| 0.00431 | 0.004267 | 0.008223 | 0.005249 | 0.00838 | 0.005601 |
| 0.004426 | 0.004441 | 0.007532 | 0.005825 | 0.007759 | 0.006944 |
| 0.00375 | 0.004119 | 0.010166 | 0.016122 | 0.007934 | 0.005761 |
| 0.004412 | 0.003912 | 0.00816 | 0.005399 | 0.008705 | 0.005754 |
| Promedios | | Promedios | | Promedios | |
| 0.00477 | 0.004820789474 | 0.00855 | 0.006677210526 | 0.00846 | 0.006427842105 |

Figura 13*Comparación de soluciones y promedios.*

Para el primer laberinto la diferencia es mínima entre los threads y los forks. Sin embargo, para el laberinto 2, que contiene dos soluciones se ve una considerable diferencia. La principal razón de este resultado, puede ser que para los threads no se implementó alguna estrategia adicional como el uso de semáforos para la los casos en los que se pueden presentar reescritura de soluciones o bien recursos compartidos, por lo que puede estar rellenar una campo del pasillo dos veces, antes de que las validaciones de moverse el hilo se consulten. Mientras que la solución para los forks se debió implementar utilizando una manera de compartir los recursos entre forks, lo cual puede arreglar el problema de los recursos compartidos.

Comparación entre Threads y Forks

Tras un pequeño análisis de los resultados obtenidos con las dos soluciones y también con el transcurso de la programación del trabajo llegamos a las siguientes conclusiones.

Los threads al ser Light Wright Processes generan menos overhead que los forks, también los threads comparten memoria, instrucciones, signals y los users ids. Por estas razones programar los threads en el trabajo fue más sencillo, ya que la forma en la que se manejan los threads es más permisiva, además la comunicación entre los hilos se realiza de con un par de instrucciones que proporciona el lenguaje C, el único aspecto a considerar es al hacer joins de los hilos ya que esto puede generar que se entre en tiempos de waiting.

También apoyándonos con una comparación de Napster (2010) encontramos lo siguiente:

- Los hilos comparten el mismo espacio de memoria, por lo que compartir datos entre ellos es realmente más rápido, lo que significa que la comunicación entre procesos (IPC) es realmente rápida.

- Si se diseñan e implementan correctamente, los hilos ofrecen más velocidad porque no hay ningún cambio de contexto a nivel de proceso en una aplicación multihilo.
- Los hilos son realmente rápidos para iniciar y terminar.

Mientras que un fork es nuevo proceso que se ve exactamente como el previo o el proceso padre, no obstante aún así es un proceso diferente con un ID de proceso diferente y que tiene su propia memoria. Por lo que su programación resulta ser un proceso más complicado cuando nunca se han utilizado ya que requiere de más cuidados para poder entrelazar la memoria compartida entre los procesos generados con el fork. Sin embargo, los forks son más fáciles de mantener y debuggear al largo plazo.

Según Napster (2010) los forks son universalmente más aceptados que los hilos por las siguientes razones:

- El desarrollo es mucho más fácil en las implementaciones basadas en forks.
- El código basado en forks es más fácil de mantener.
- La bifurcación es mucho más segura porque cada proceso bifurcado se ejecuta en su propio espacio de direcciones virtual. Si un proceso se bloquea o tiene un desbordamiento del búfer, no afecta a ningún otro proceso en absoluto.
- El código de los hilos es mucho más difícil de depurar que el de los forks.
- Los forks son más portables que los threads.
- Los forks es más rápida que los hilos en una sola cpu ya que no hay sobrecargas de bloqueo o cambio de contexto.

Manual de usuario

Para compilar y ejecutar el programa es necesario entrar en el directorio: `../LaberintoThreads-Forks/maze-threads-forks/src`

Adentro de esa carpeta src se corre el comando: *make*

Si no se tiene instalado el comando *make* se puede intalar con los siguientes comandos.

```
$ sudo apt-get update
$ sudo apt-get -y install make
```

Consola 1: Instalar Make

Una vez que se corre el programa con *make*, se solicita el nombre del archivo que contiene el laberinto. Se debe ingresar el nombre del archivo completo (nombre_archivo.txt).

```
$
$ make
gcc main.c -o program.exe -lpthread
./program.exe
Ingrese el nombre del archivo: lab1.txt
```

Consola 2: Ejemplo de ingreso de mapa

El programa buscar los archivos de los laberintos en `../LaberintoThreads-Forks/maze-threads-forks/files/maps`. Por lo que si se quiere integrar un mapa nuevo se debe agregar en ese directorio. Además el txt del archivo debe tener el siguiente formato:

- En la primera línea puede o no contener el número de filas y columnas, ambas separadas por un espacio. Es preferible escribir el número de filas y columnas en el archivo de texto.
- En las siguientes líneas se ingresa la forma del laberinto, representada con paredes (asteriscos) y pasillos (espacios).
- Se debe verificar que esté en el rango correcto de filas y columnas la forma del laberinto.
- Debe guardarse en un archivo de texto (.txt)

```

1  10 10
2  *  **/**
3  *  *
4  *  *  ***
5  *  *****
6  |  |  *
7  ** * * *
8  *  *  *  *
9  |  |  */***
10 ***** *
11

```

Figura 14
Ejemplo de laberinto válido

En el apéndice se adjuntan las capturas de pantallas de una ejecución completa con un laberinto de ejemplo.

Bitácora

- 06-09-21: Nos reunimos para seccionar el proyecto y poder asignar tareas a cada uno de los integrantes.
- 07-09-21: Se empezó y se finalizó de programar la estrategia para la solución del laberinto por medio de threads.
- 08-09-21: Se organizo y se limpio el código de la solución del laberinto por medio de threads.
- 10-09-21: Se empezó a programar la estrategia para la solución del laberinto por medio de forks.
- 12-09-21: Se finalizó de programar la estrategia para la solución del laberinto por medio de forks y además se organizo y se limpio el código.
- 13-09-21: Se empezó y finalizó de programar una manera de hacer pruebas y comparaciones entre las dos estrategias.
- 16-09-21: Se empezó a documentar de manera detallada las estrategias planteadas.
- 18-09-21: Se encontró un problema con la lógica de la creación y muerte de los threads y forks.
- 20-09-21: Se solucionó el problema de la lógica de la creación y muerte de los threads y forks y además se concluyó con la documentación.

Referencias

- GeekforGeeks. (2019). *fork() in C*. <https://www.geeksforgeeks.org/fork-system-call/>
- Ippolito, G. (2004). *POSIX thread (pthread) libraries*. <https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>
- Kerrisk, M. (2021). *pthread_join(3) - Linux manual page*. https://man7.org/linux/man-pages/man3/pthread_join.3.html
- Napster. (2010). *Forking vs Threading*. https://view.officeapps.live.com/op/view.aspx?src=https%3A%2F%2Fwww.dsi.fceia.unr.edu.ar%2Fimages%2FForking_vs_Threading.doc&wdOrigin=BROWSELINK
- POSIX Threads*. (s.f.). <http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html>

Apéndice

Ejemplo de archivos con los laberintos

Laberinto 1

```

10 10
 *      *****
 *  *
 *  *  ****
 *  *  ****
      *      *
**  *  *      *
  *  *  *  *
      *  ***
****  *      *
      */****

```

lab1.txt

Laberinto 2

```

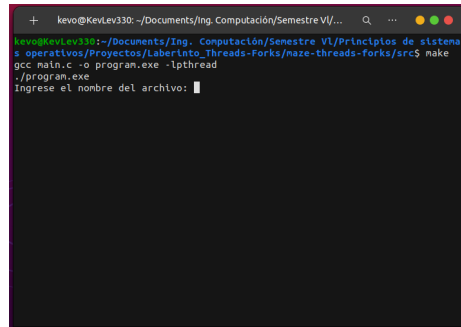
      *           * * * * *
          *               *
    *   *   * * *   *
      *   *       *   *
*       *         *   *
* * * * * *   *   *
      *   *       *   *
          *   * * * *
*   * * *   *       * *
*   *   *   *   *   /
*/ * * * * * * * *

```

lab2.txt

Ejemplo de ejecución de un programa

Se ingresa la instrucción `make` en el directorio donde tenemos el `src` del proyecto.

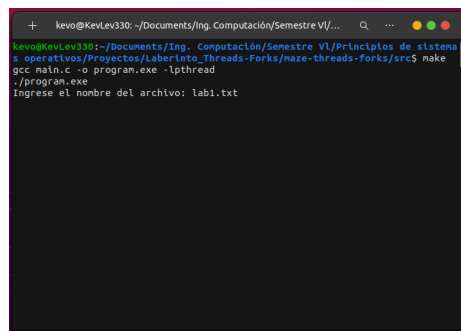


```
kevo@KevLev330: ~/Documents/Ing. Computación/Semestre VI/...  
kevo@KevLev330:~/Documents/Ing. Computación/Semestre VI/Principios de sistema  
operativos/Proyectos/Laberinto_Threads-Forks/maze-threads-forks/src$ make  
gcc main.c -o program.exe -lpthread  
./program.exe  
Ingrese el nombre del archivo: 
```

Figura 15

Ejecución del programa.

Luego, se ingresa el archivo de texto con el laberinto que se desea ejecutar.



```
kevo@KevLev330:~/Documents/Ing. Computación/Semestre VI/...  
kevo@KevLev330:~/Documents/Ing. Computación/Semestre VI/Principios de sistema  
operativos/Proyectos/Laberinto_Threads-Forks/maze-threads-forks/src$ make  
gcc main.c -o program.exe -lpthread  
./program.exe  
Ingrese el nombre del archivo: lab1.txt
```

Figura 16

Selección del archivo con el laberinto.

Seguidamente, se muestra el proceso que se sigue para buscar la meta.

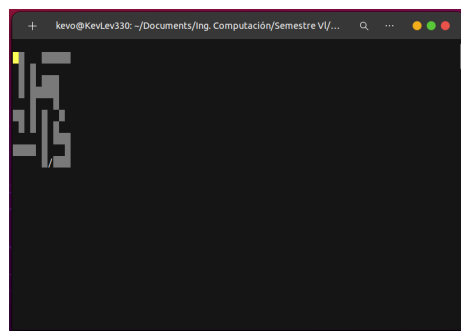


Figura 17

Primer paso de la solución.

Luego se pide que se digite un *Enter* para continuar con el proceso de Forks.

Luego de que imprime el último paso de la solución con forks, el programa imprime la duración de ambos procesos en segundos y finaliza su ejecución.

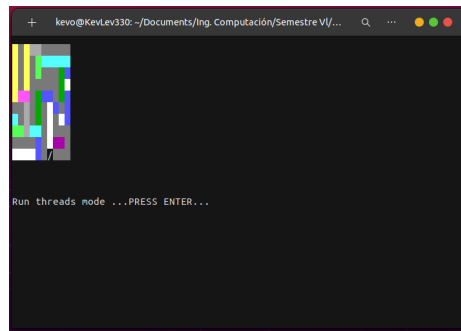


Figura 18
Solución final de proceso con Threads.

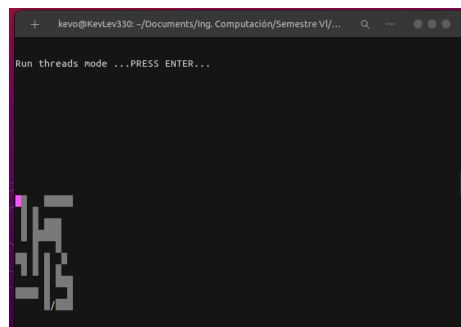


Figura 19
Primer paso del Fork.

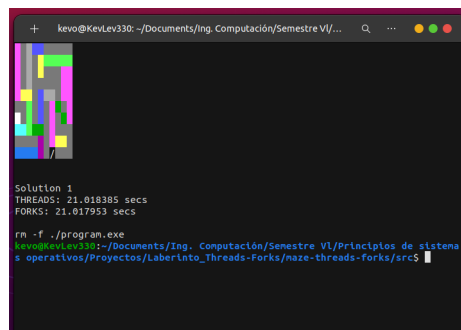


Figura 20
Impresión de tiempos y finalización.