Universidad Nacional de Rosario

Facultad de ciencias exactas, ingeniería y agrimensura

Sistemas operativos I

Trabajo práctico 1

Alumnos:

Petruskevicius Ignacio - Trincheri Lucio

Abril 2021

SOI - Informe TP 1

Petruskevicius Ignacio - Trincheri Lucio, LCC UNR FCEIA 16/04/2021

1. Introducción

En este informe contaremos detalles sobre la implementación del Game of Life de Conway propuesto en el TP1 de Sistemas Operativos y aclaraciones sobre como compilarlo y ejecutarlo. Comenzando por las estructuras de datos elegidas para almacenar la información necesaria, luego por las decisiones que tomamos sobre como hacer el programa concurrente y por ultimo problemas y comentarios que nos resultaron interesantes de presentar.

2. Compilación y ejecución

El programa se compila simplemente escribiendo el comando **make** en la consola de comandos, luego se procede a ejecutar el archivo **./simulador** seguido por el archivo semilla.

Para limpiar los archivos de compilación y el ejecutable, está a disposición el comando **make clean**.

3. Estructuras de datos

3.1. Game:

Para almacenar la información sobre el game, decidimos generar una struct de la siguiente forma:

Listing 1: Struct game

```
1    typedef struct _game{
2       board_t *board;
3       int ciclos;
4    }game_t;
```

en la cual tenemos un tablero (board) y la cantidad de ciclos pedidos a realizar, al comienzo el tablero sera el pasado por el archivo, y se le aplicaran a este la cantidad de generaciones como ciclos se hayan indicado en el archivo semilla.

3.2. Board:

Listing 2: Struct board

```
1    typedef struct _board{
2        unsigned int columnas;
3        unsigned int filas;
4        char ** grilla;
5    }board_t;
```

Para almacenar un tablero, decidimos utilizar una array bidimensional de chars, lo cual facilita ingresar a cada celda del tablero mediante coordenadas, además agregamos la cantidad de columnas y filas a la estructura para tener limites en a la grilla y no sobrepasarlos. En este punto tuvimos en cuenta varias opciones, como trabajar directamente con una array unidimensional y usar los tamaños de fila y columna para acceder a cada elemento, pero vimos que complicaba innecesariamente la escritura del programa.

4. Programa concurrente

La idea que tomamos para implementar el Game of Life de manera concurrente fue la siguiente, dividir el tablero en tantas partes como unidades de procesamiento tengamos y generar tantos threads como unidades de procesamiento que se encarguen de actualizar las células que les corresponden. Utilizamos una barrera para que los threads esperan a que todos los demás threads terminen la generación actual antes de pasar a la siguiente.

Veamos en detalle como hicimos cada parte. Primero necesitamos dos tableros, el que contiene la generación actual y el que contendrá la nueva. Esto es para no modificar el tablero que estamos leyendo lo que podría traer una mezcla en el resultado de la generación. Además esto nos permite quitar los posibles problemas de generar una zona critica, pues estaríamos leyendo y escribiendo con distintos threads el mismo espacio de memoria, vimos en clase los problemas que esto trae (nos ahorramos locks, que harían que los threads se detengan si otro esta accediendo a la zona crítica).

Recapitulando, tenemos 2 tableros uno con la generación actual y otro que contendrá la nueva generación. Cada hilo actualiza una zona del tablero, lee lo que necesite del viejo (que al no ser modificado no hay problemas) y modifica solo la zona que él puede actualizar que no coincide con la zona de ningún otro thread. Luego de que todos los threads hayan actualizado su zona, nos aseguramos de esto con una barrier de POXIS, los hilos se esperan a que todos hayan actualizado. En ese momento se intercambia el tablero de la vieja generación con el de la nueva, para realizar otra generación más.

Con esto logramos ”reducir” el tamaño del tablero a actualizar pues cada proceso tendrá un tablero de aproximadamente

$$cant.celdas = \frac{(columnas * filas)}{cant.unidadesDeProcesamiento} \quad (1)$$

Mientras los hilos ejecutan la cantidad de ciclos que se indicaron en la semilla el main espera a que terminen, cuando lo hacen, imprime en el archivo el tablero final.

4.1. Trabajo de cada thread:

Para que cada thread pueda realizar la tarea es necesario que pueda acceder a cierta información, para ello creamos la siguiente estructura que es pasa como argumento a cada hilo:

Listing 3: Struct argshilo

```
1    typedef struct _argshilo{
2     int inicio;
3     int final;
4     board_t *viejo;
5     board_t *nuevo;
6     int ciclos;
7  } argshilo;
```

La misma contiene el intervalo que le toca a cada hilo actualizar, el tablero inicial y el nuevo en donde se actualizara la generación nueva y la cantidad de ciclos a realizar.

Listing 4: Función que ejecuta cada thread

```
1  void *trabajo_thread(argshilo *arg){
2      board_t *nuevoThread = arg->nuevo;
3      board_t *viejoThread = arg->viejo;
4      board_t *aux;
5
6      for (int i = 0; i < arg->ciclos; i++) {
7          // intervambiamos viejo con nuevo y llamamos a nueva generacion, cuando
8          // todos los threads lo hagan seguimos con el ciclo.
9
10         aux = nuevoThread;
11         nuevoThread = viejoThread;
12         viejoThread = aux;
13
14         nueva_generacion_tablero(arg->inicio, arg->final, viejoThread, nuevoThread↩
               );
15         pthread_barrier_wait(&barrier);
16     }
17     free(arg);
18
19     pthread_exit(0);
20 }
```

Cada thread guarda la dirección de cada uno de los dos tableros, luego los intercambia y actualiza con la función nueva generación tablero las celdas del intervalo indicado. Cuando esta función termina, el hilo espera a que los demás hilos terminen la ejecución de la actualización actual, para luego proceder a continuar. Una vez finalizados todos los ciclos, se libera la estructura

argshilos y se sale del thread.

Aunque esta sea la implementación final, hubo diferentes iteraciones de la misma, las cuales se detallan a continuación:

4.2. 1ra implementación

Por otra parte, en cuanto a la idea de threads pensamos en dos implementaciones previas a la solución final a la cual llegamos. La primera de ellas se trataba de crear múltiple threads y utilizar semáforos para restringir la cantidad en uso concurrente. La idea surgió al intentar utilizar todas las herramientas vistas en el dictado de la materia, pero fue rechazada al no encontrarle necesidad al uso de los semáforos.

4.3. 2da implementación

La segunda implementación fue la de realizar un loop en la función main() de nuestro programa, donde creábamos una cantidad de threads igual a la devuelta por "get_nprocs()", calcular la actualización del tablero y finalizar retornando al main(), donde los threads sería destruidos y se iteraría en el loop. Esta idea fue descartada ya que nos dimos cuenta que era posible crear los threads una sola vez y hacer que ellos recorran el loop con las iteraciones pedidas. Esto dio lugar a la implementación final que se encuentra en el TP.

4.4. Otras posibles implementaciones contempladas

Teniendo en cuenta que el programa solo devuelve el resultado luego de x ciclos, pensamos en diferentes posibilidades, ya que podemos adelantar la ejecución en ciertas zonas del tablero (es decir que zonas ejecuten más ciclos que otras). Esta idea, además de ser engorroso de leer, nos pareció demasiado complejo para la idea pedida y los conceptos que abarcaba este TP.

5. Entrada-salida

Nos gustaría realizar una aclaración sobre la carga de datos, como grupo decidimos no enfocar mucho esfuerzo en asegurarnos de la sanitidad de la entrada, ya que no es el centro de la materia ni de la Practica 1. Por lo tanto suponemos que el archivo de entrada es correcto, es decir, la cantidad de lineas es igual a las de las filas, el orden es el correcto, cada fila tiene la cantidad indicada de caracteres. Por otro lado, para almacenar el numero de caracteres 'X' o 'O' se utiliza una array de 512 caracteres, suponemos que es suficiente (el tablero seria extremadamente grande si una fila necesita mas de un numero de 512 dígitos, dado que el tamaño solo es necesario para colocar la células vivas, visto que el plano es infinito para luego calcular el juego).

6. Dificultades y errores

Un problema que surgió y nos causó confusión en su momento fue el hecho que Valgrind reportaba memoria que aún podía ser accedida pero no fue liberada. **A veces.**

Tras investigar el problema, este comportamiento parece estar relacionado con el hecho que al finalizarse los threads, parte de las estructuras creadas siguen disponibles durante el resto del proceso, ya que estas pueden ser reutilizadas. Valgrind detecta a esta asignación de memoria para los threads, pero como no es liberada antes de que el programa termine, la marca como "still reachable". Esto no es un memory leak en sí, pero nos parecía necesario aclarar la situación.

Por otra parte, hay ejecuciones del programa en las que la memoria se libera antes de que finalice el main(), por lo cual Valgrind no indica memoria aún asignada. Adjuntamos fotos del comportamiento a continuación, y el comando utilizado:

valgrind –error-exitcode=1 –tool=memcheck –gen-suppressions=all –leak-check=full –leak-resolution=med –track-origins=yes –vgdb=no –show-reachable=yes ./simulador

Ejemplo de funcionamiento sin memoria restante:

```
lucio-unix@DESKTOP-HFQ30MG:/mnt/c/Users/lucio/Documents/Facultad/3er año/1er cuatrimestre/Sistemas Operativos I/ConwayGoF
ll --leak-check=full --leak-resolution=med --track-origins=yes --vgdb=no --show-reachable=yes -s ./main
==9251== Memcheck, a memory error detector
==9251== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9251== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==9251== Command: ./main
==9251==
==9251==
==9251== HEAP SUMMARY:
==9251==     in use at exit: 0 bytes in 0 blocks
==9251==   total heap usage: 39 allocs, 39 frees, 5,094 bytes allocated
==9251==
==9251== All heap blocks were freed -- no leaks are possible
==9251==
==9251== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ejemplo de funcionamiento con memoria restante:

```
==9247== 1,198 bytes in 1 blocks are still reachable in loss record 4 of 4
==9247==    at 0x483DD99: calloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==9247==    by 0x400D273: _dl_new_object (dl-object.c:89)
==9247==    by 0x4006E96: _dl_map_object_from_fd (dl-load.c:997)
==9247==    by 0x400A61A: _dl_map_object (dl-load.c:2236)
==9247==    by 0x4015D36: dl_open_worker (dl-open.c:513)
==9247==    by 0x49D98B7: _dl_catch_exception (dl-error-skeleton.c:208)
==9247==    by 0x40155F9: _dl_open (dl-open.c:837)
==9247==    by 0x49D8860: do_dlopen (dl-libc.c:96)
==9247==    by 0x49D98B7: _dl_catch_exception (dl-error-skeleton.c:208)
==9247==    by 0x49D9982: _dl_catch_error (dl-error-skeleton.c:227)
==9247==    by 0x49D8994: dlerror_run (dl-libc.c:46)
==9247==    by 0x49D8994: __libc_dlopen_mode (dl-libc.c:195)
==9247==    by 0x486899A: pthread_cancel_init (unwind-forcedunwind.c:53)
==9247==
{
   <insert_a_suppression_name_here>
   Memcheck:Leak
   match-leak-kinds: reachable
   fun:calloc
   fun:_dl_new_object
   fun:_dl_map_object_from_fd
   fun:_dl_map_object
   fun:dl_open_worker
   fun:_dl_catch_exception
   fun:_dl_open
   fun:do_dlopen
   fun:_dl_catch_exception
   fun:_dl_catch_error
   fun:dlerror_run
   fun:__libc_dlopen_mode
   fun:pthread_cancel_init
}
==9247== LEAK SUMMARY:
==9247==    definitely lost: 0 bytes in 0 blocks
==9247==    indirectly lost: 0 bytes in 0 blocks
==9247==      possibly lost: 0 bytes in 0 blocks
==9247==    still reachable: 1,654 bytes in 4 blocks
==9247==         suppressed: 0 bytes in 0 blocks
==9247==
==9247== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

7. Bibliografía

- Documentación de la librería PTHREAD.
- https://es.wikipedia.org/wiki/Juego_de_la_vida
- https://es.wikipedia.org/wiki/Run-length_encoding
- Teoría proveída por la cátedra.