Lab(Conway's Game of Life)

ESE 3025: EmbeddedReal Time Operating Systems

Lambton College in Toronto

Instructor: Takis Zourntos

STUDENT NAME & ID:

Vishal Hasrajani(C0761544)

Parth Patel(C0764929)

Goutham Reddy Alugubelly(C0747981)

Ratnajahnavi rebbapragada(C0762196)

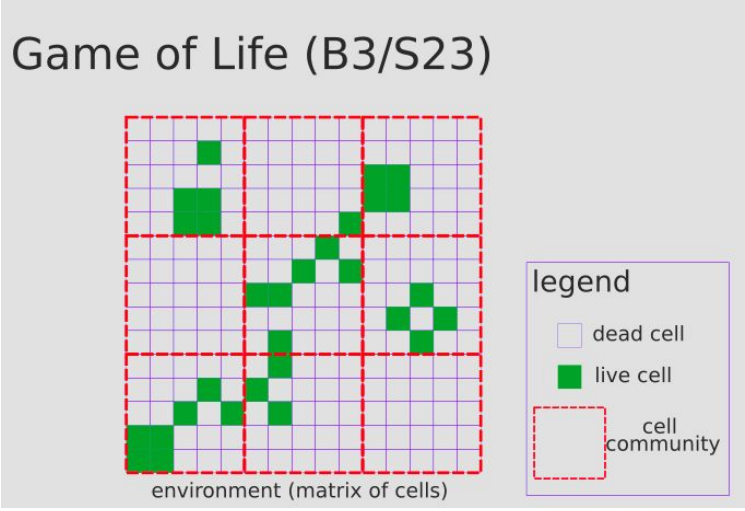
INTRODUCTION :

Conway's game of life has a 2D grid of square cells which can be either live or dead.Every cell interacts with its 8 neighbours.It consists of a collection of cells which, based on a few mathematical rules, can live, die or multiply. Depending on the initial conditions, the cells form various patterns throughout the course of the game.

DESCRIPTION:

Rules of Game of life :

- If a cell is dead but surrounded by exactly three live neighbours, it sprouts to life (birth)
- If a cell is live but has more than 3 live neighbours, it dies (overpopulation)
- If a cell is live but has fewer than 2 live neighbours, it dies (underpopulation)
- All other dead or live cells remain the same to the next generation (i.e., a live cell must have exactly three neighbours to survive)



MAIN THREAD EXPLANATION:

First we have to implement the environment as a 2D array of rows and columns in order to store the data.

```
cell_t env[config_NE][config_ME];
```

update_env is the second array to update the things and is used to store the data after applying the rules.

```
cell_t update_env[config_NE][config_ME];
```

Now coming to **gol_config.h**

```
#ifndef GOL_CONFIG_H_
#define GOL_CONFIG_H_

#include <stdlib.h>

/*
 * "community of cells" (handled by one thread) parameters
 */
#define config_NC               16 // # of cell rows in a community
#define config_MC               16// # of cell columns in a community

/*
 * overall environment parameters
 */
#define config_K                2 // # of communities "down"
#define config_L                8// # of communities "across"
#define config_NE               config_K*config_NC // number of environment rows
#define config_ME               config_L*config_MC // number of environment columns

/*
 * temporal parameters
 */
#define config_TL               80000 // microseconds between generation
#define config_TDISP    1 // number of generations between plots

/*
 * basic cell type
 */
enum cell_enum
{
        dead = 0U, live = 1U
};
typedef enum cell_enum cell_t;

/*
 * thread identifier (in units of community BLOCKS not cells!)
 */
struct threadID_struct
{
        size_t row;
        size_t col;
};
typedef struct threadID_struct threadID_t;
```

Here,all the parameters are defined for our project. config_NC is the number of cell rows in a community.Then config_MC is the number of cell columns in a community.

config_K and config_L are the communities down and across respectively which will be majorly used for our threads.Overall environment is created (i.e config_NE and config_ME)by multiplying config_K*config_NC and config_L*config_MC respectively.

config_TL is microseconds between generations.We can change this parameter as per our convenience.

cell_enum is of enum data type that is used to define name to a particular integer.
Here dead = 0U and live = 1U(means unsigned integer 0 and 1 are defined as dead and live accordingly)
A structure is created (i.e threadID_struct) to define row and column.

```
pthread_t threadptrs[config_K * config_L];
```

This is a thread handle .

```
threadID_t threadID[config_K * config_L];
```

This will be used as a thread id.

`initEnvironment()` function is used to get the data from seed_input.txt file and then store it in an 2D array.

```
size_t index;
     for (size_t i = 0; i != config_K; ++i)
     {
          for (size_t j = 0; j != config_L; ++j)
          {
               index = i * config_L + j; // map (i,j) to an
1-d index
               threadID[index].row = i;
               threadID[index].col = j;
               // the following if condition returns 0 on
the successful creation of each thread:
               if (pthread_create(&threadptrs[index], NULL,
&updateCommFunc,&threadID[index]) != 0)
               {
                    printf("failed to create the thread
%d\n", (int) index);
                    return 1;
               }
          }
     }
```

Here **index** is used to point to a particular thread that is unique.
Using if condition we are creating a thread corresponding to a particular community in an environment.

The actual execution can be seen by an example below :

|
|
|
.
.
.

thread[0].row = 0
thread[0].col = 0

---

thread[1].row = 0
thread[1].col = 1

---

thread[2].row = 0
thread[2].col = 2

---

thread[3].row = 0
thread[3].col = 3

Further, if the **reproduction_flag==true**,we can allow new generations to check in.
And when **reproduction_flag==false**,we can update the display.

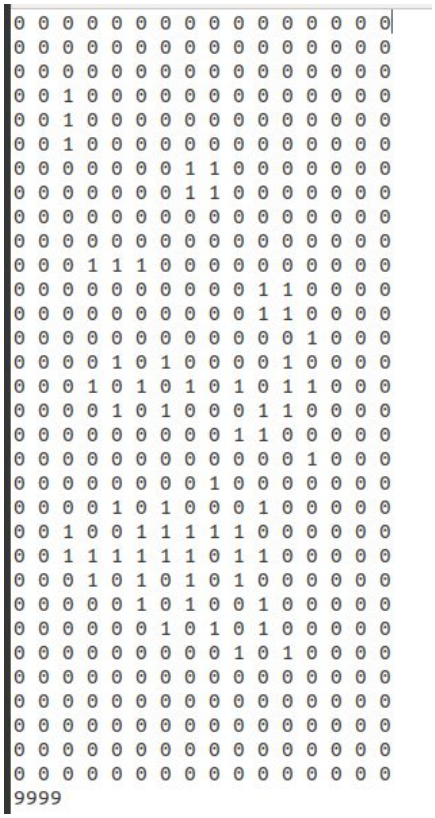
EXPLANATION OF FUNCTIONS IN **cells.c**

1) void initEnvironment(void) :

```
void initEnvironment(void)
{
     // start by reading in a single community
     int token;
     cell_t datum;
     cell_t community_init[config_NC][config_MC];

     printf("\ninitializing environment...\n");
     printf("     ... loading template community from
stdin\n");
     for (size_t i = 0; i != config_NC; ++i)
     {
          for (size_t j = 0; j != config_MC; ++j)
          {
               scanf("%d", &token);
               datum = (cell_t) token;
               community_init[i][j] = datum;
          }
     }
     printf("     ... done.\n");

     printf("     ... creating communities\n");
```

```
    // copy this community to each community in env to
initialize it
    for (size_t i = 0; i != config_K; ++i)
    {
        for (size_t j = 0; j != config_L; ++j)
        {
            transferCommunity(i, j, community_init);
        }
    }
    printf("     ... done.\n");

}
```

In this function, input (i.e is 0 or 1) is taken using scanf from the file **seed_input_32_x_16.txt**



All the data is stored in the array `community_init[i][j].`
This data is copied to each and every community in the environment using **transferCommunity function**.

2) void transferCommunity(size_t iT, size_t jT,const cell_t data_init[config_NC][config_MC]) :

```
void transferCommunity(size_t iT, size_t jT,
          const cell_t data_init[config_NC][config_MC])
{
     size_t i_0 = iT * config_NC;
     size_t j_0 = jT * config_MC;

     printf("     ... transferring block (%d, %d)\n", (int)
(iT + 1),
               (int) (jT + 1));
     // copy this community to each community in env to
initialize it
     for (size_t i = 0; i != config_NC; ++i)
     {
          for (size_t j = 0; j != config_MC; ++j)
          {
               env[i_0 + i][j_0 + j] = update_env[i_0 +
i][j_0 + j] =
                         data_init[i][j];
          }
     }
}
```

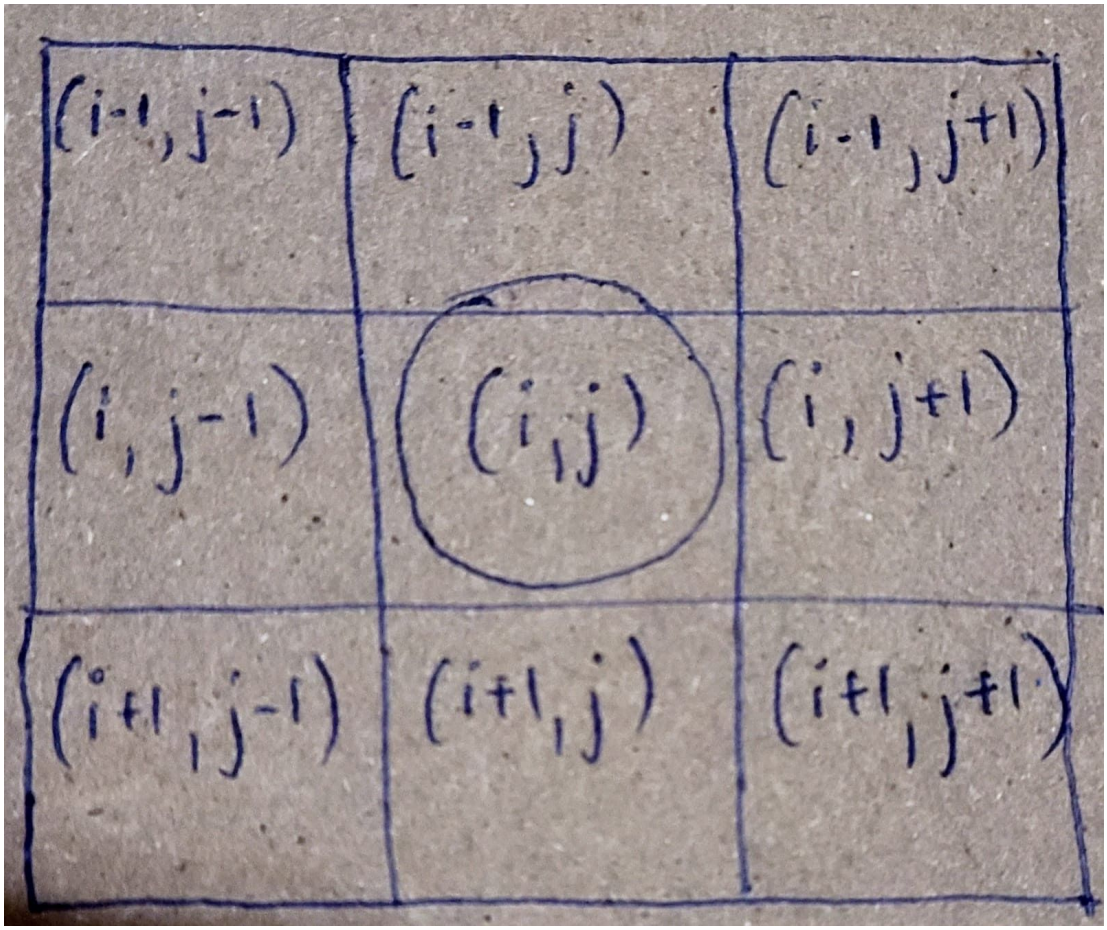
This function is used to transfer the data to each and every community using the for loop.Here,initially the env and update_env will be having the same data.

3) size_t countLiveNeighbours(size_t row, size_t col) :

```
size_t countLiveNeighbours(size_t row, size_t col)
{
     size_t cell_count = 0;
     size_t R, C;


          for (int i=-1;i<2;i++)
          {
               for(int j=-1;j<2;j++)
               {

                    R = (size_t) (row + i + config_NE) %
config_NE;
                    C = (size_t) (col + j + config_ME) %
config_ME;
                    cell_count = cell_count +
(size_t)env[R][C];
               }
          }

          cell_count = cell_count -  (env[row][col]==live ?
1 : 0); // works only because live = 1, dead = 0;

     return cell_count;
}
```

This function is useful in counting the live neighbours of a single cell.

To get the actual picture of how the cells looks like,you can refer the picture below:
(One cell is exactly surrounded by 8 cells)



So to count the surrounding neighbours we have used 2 for loop(one for row and one for column) from -1 to 1.

We have implemented a wrap around condition here.

```
(row + i + config_NE) % config_NE;
```

1st condition : Suppose we are at the **very start** (i.e 0th position).
At the edge **i** will be -1
So row=0 , i=-1 and config_NE=32

i.e (0 + -1 + 32) % 32
     = 31 % 32
     =31

Here we wrapped up from 0 to 31st row.

Similarly, it is the same for the columns.

2nd condition : Suppose we are at the **very end** (i.e at the 31st position)
At the edge **i** will be 1.
Sow row=31 , i=1 and config_NE=32

i.e (31 + 1 + 32) % 32
     = 64 % 32
     =0

Here we wrapped up from 31st row to 0th row.

Finally we used a counter **cell_count** to count the live cell.

4) void updateCell(size_t r, size_t c) :

```
void updateCell(size_t r, size_t c)
{
```

```
if(state_cell==0 && live_neighbours==3)
{
     update_cell[r][c]= live;

 //cell is dead and having 3 live neighbours,becomes a live
cell in next generation

}
else if(state_cell==1 &&( live_neighbours<2 ||
live_neighbours>3))
{
     update_cell[r][c] = dead;

//cell is live but has more than 3 live neighbours or less
than 2 live neighbours,then it dies in the next generation.
}
else
{
     update_cell[r][c] =state_cell;

//All others remain the same.

}
```

```
}
```

This function is used to implement the rules of Conway's Game of life.We did it using simple if else conditions.

5) void* updateCommFunc(void *param) :

This function updates all the cells for a thread (corresponding to one community)

```
void* updateCommFunc(void *param)
{
// If the reproduction flag is true means we can allow new
generations to check in ..

     if(reproduction_flag==true)

     {
          // *testing is a pointer pointing to the same
location as param
          threadID_t  *testing = param;

          //getting the block pair corresponding to a
thread.
          size_t i_t = testing->row;
          size_t j_t = testing->col;

//multiplying it with config_NC and config_MC to get exact
position of a row and column in a particular community
          size_t i_0 = i_t * config_NC;
          size_t j_0 = j_t * config_MC;

//Using FOR loop for updating all the cells corresponding
to a particular community
               for (size_t i = 0; i != config_NC; ++i)
               {
                    for (size_t j = 0; j != config_MC; ++j)
                    {
                    updateCell(i+i_0,j+j_0);
```

```
                    }
                }
            }



}
```

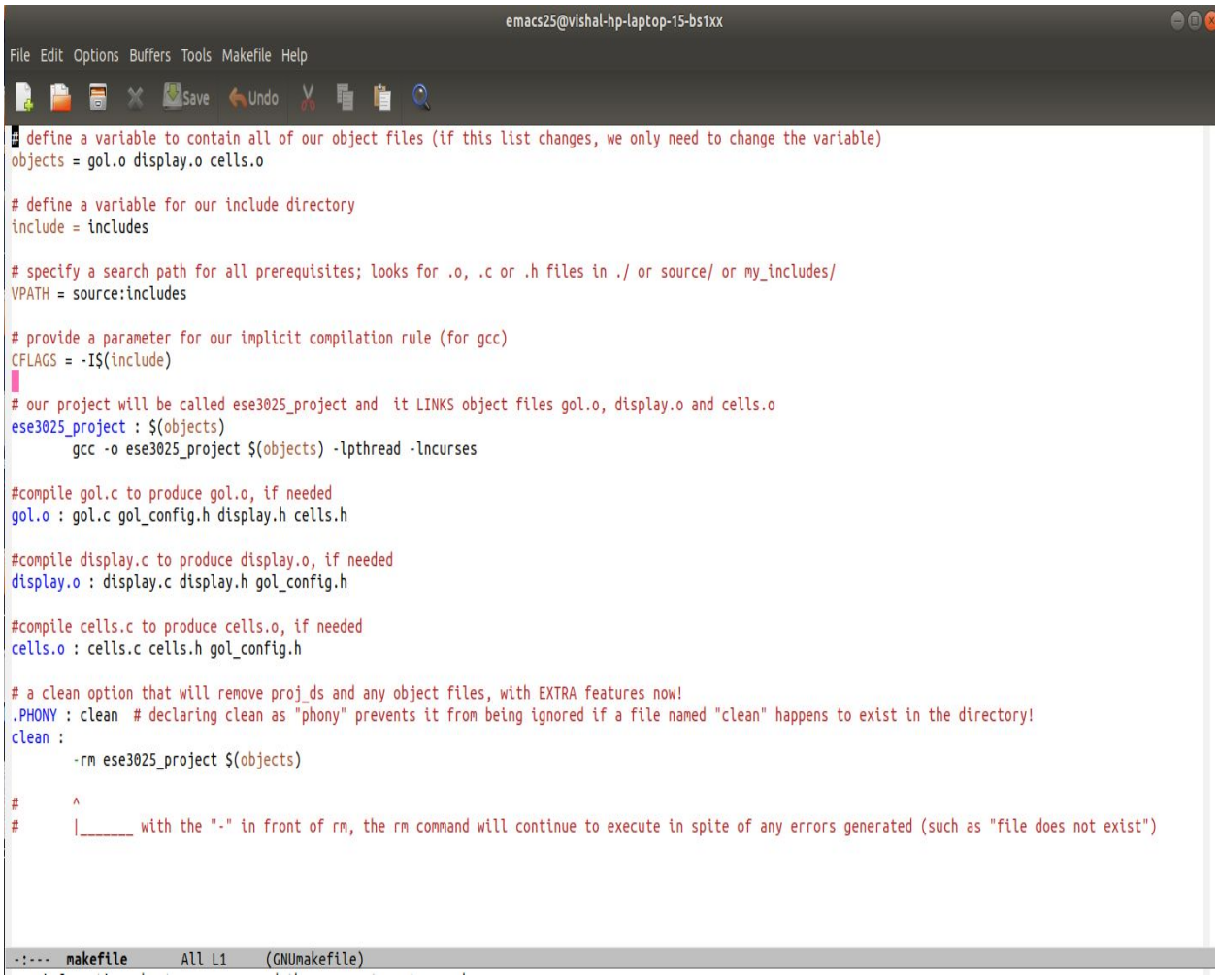
6) void copyEnvironment(void) :

```
void copyEnvironment(void)
{
     // copy this community to each community in env to
initialize it
     for (size_t i = 0; i != config_NE; ++i)
     {
          for (size_t j = 0; j != config_ME; ++j)
          {
               env[i][j] = update_env[i][j];
          }
     }
}
```

This function is simply used to copy the values from update_env to env variable .

display.c is used for displaying the (*) in place of 1 on our console using the ncurses.

Makefile for this project :



Implementation video for host machine :

https://www.youtube.com/watch?v=BZRz1BMnj6o

Implementation video for Beaglebone Black :

https://www.youtube.com/watch?v=dY0gpgX12Cw

CONCLUSION :

To sum up, we learned many new things like using pthreads for a particular community and how to distribute a big program into smaller sections or functions to make it easy to understand.At last, we implemented Conway's Game Of Life using pthreads.

APPENDIX :

gol_config.h

```
/*
 * gol_config.h
 *
 *  Created on: May 30, 2020
 *      Author: takis
 */

#ifndef GOL_CONFIG_H_
#define GOL_CONFIG_H_

#include <stdlib.h>

/*
 * "community of cells" (handled by one thread) parameters
 */
#define config_NC       16 // # of cell rows in a community
#define config_MC       16// # of cell columns in a
community

/*
 * overall environment parameters
 */
#define config_K        2 // # of communities "down"
#define config_L        8// # of communities "across"
#define config_NE       config_K*config_NC // number of
environment rows
#define config_ME       config_L*config_MC // number of
environment columns
```

```
/*
 * temporal parameters
 */
#define config_TL       80000 // microseconds between
generation
#define  config_TDISP  1 // number of generations between
plots

/*
 * basic cell type
 */
enum cell_enum
{
     dead = 0U, live = 1U
};
typedef enum cell_enum cell_t;

/*
 * thread identifier (in units of community BLOCKS not
cells!)
 */
struct threadID_struct
{
     size_t row;
     size_t col;
};
typedef struct threadID_struct threadID_t;

/*
 * a neighbour type for cells... here, X represents the
cell:
 *
```

```
 *                a b c
 *              d X e
 *              f g h
 *
 */
enum neighbour_enum
{
     a_posn=0U,
     b_posn,
     c_posn,
     d_posn,
     e_posn,
     f_posn,
     g_posn,
     h_posn
};
typedef enum neighbour_enum neighbour_t;
```

display.h

```
/*
 * display.h
 *
 *  Created on: May 30, 2020
 *      Author: takis
 */

#ifndef DISPLAY_H_
#define DISPLAY_H_

#include <ncurses.h>
```

```
#include <stdbool.h>

// window parameters
#define CELL_CHAR '*'
#define TIME_OUT  300

/*
 * functions
 */
void initDisplay(void);

void updateDisplay(void);

#endif /* DISPLAY_H_ */
```

cells.h

```
/*
 * cells.h
 *
 *  Created on: May 30, 2020
 *      Author: takis
 */

#ifndef CELLS_H_
#define CELLS_H_

#include "gol_config.h"
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <stdbool.h>

/*
 * functions
 */
void initEnvironment(void);

void copyEnvironment(void);

void* updateCommFunc(void*);

#endif /* CELLS_H_ */
```

gol.c

```
/*
 * gol.c
 *
 *  Created on: May 30, 2020
 *      Author: takis
 */

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <pthread.h>
#include <unistd.h>
#include <ncurses.h>
#include "gol_config.h"
#include "cells.h"
#include "display.h"
```

```
/*
 * global variables
 */
cell_t env[config_NE][config_ME];
cell_t update_env[config_NE][config_ME];
bool reproduction_flag = false; // is high when it's mating
season

int STARTX = 0;
int STARTY = 0;
int ENDX = config_ME;
int ENDY = config_NE;
WINDOW *win;
/*
 * main code
 */
int main(void)
{
     pthread_t threadptrs[config_K * config_L]; // our
thread handles
     threadID_t threadID[config_K * config_L]; // thread ID

     // initialize workspace
     initEnvironment();

     // create the threads
     printf("\ncreating threads...\n");
     size_t index;
     for (size_t i = 0; i != config_K; ++i)
     {
          for (size_t j = 0; j != config_L; ++j)
          {
               index = i * config_L + j; // map (i,j) to an
```

```
1-d index
                threadID[index].row = i;
                threadID[index].col = j;
                // the following if condition returns 0 on
the successful creation of each thread:
                if (pthread_create(&threadptrs[index], NULL,
&updateCommFunc,
                              &threadID[index]) != 0)
                {
                     printf("failed to create the thread
%d\n", (int) index);
                     return 1;
                }
           }
     }

     // initialize display with ncurses
     initDisplay();

     unsigned short int ctr = 0;
     while (1)
     {
          reproduction_flag = true;
          usleep(config_TL / 2); // allow new generation to
check in
          reproduction_flag = false;
          usleep(config_TL / 2); // put a hold on
reproduction to update display
          if (++ctr == config_TDISP)
          {
               ctr = 0;
               updateDisplay();
          }
```

```
            copyEnvironment(); // write changes to the
environment, env, from update_env
        }

        // should never arrive here;
        return 1;
}
```

display .c

```
/*
 * display.c
 *
 *  Created on: May 30, 2020
 *      author: takis
 *      note: a lot of this code adapted from the TDLP
tutorial on ncurses,
 *      by Pradeep Padala
 */

#include "gol_config.h"
#include <unistd.h>
#include <ncurses.h>
#include "display.h"

/*
 * important variables, defined elsewhere
 */
extern cell_t env[config_NE][config_ME];
extern int STARTX;
extern int STARTY;
extern int ENDX;
```

```
extern int ENDY;
extern WINDOW *win;

/*
 * PRIVATE FUNCTIONS
 */
void create_newwin(int height, int width)
{
     win = newwin(height, width, STARTY, STARTX);
     box(win, 0, 0); /* 0, 0 gives default characters
      * for the vertical and horizontal
      * lines */
     wrefresh(win); /* show that box */

     return;
}

/*
 * PUBLIC FUNCTIONS
 */
void initDisplay(void)
{
     printf("\ninitializing display...\n");
     usleep(2 * config_TL);
     initscr();
     cbreak();
     timeout(TIME_OUT);
     keypad(stdscr, TRUE);
     create_newwin(config_NE, config_ME);
}

void updateDisplay(void)
{
```

```
//   ENDX = COLS - 1;
//   ENDY = LINES - 1;

     int i, j;
     wclear(win);
     for (i = STARTX; i != config_ME; ++i)
          for (j = STARTY; j != config_NE; ++j)
               if (env[j][i] == live)
                    mvwaddch(win, j, i, CELL_CHAR);
     wrefresh(win);
}

/*

**************************************************************
********************************* reference
 */
```

cells .c

```
/*
 * cells.c
 *
 *  Created on: May 30, 2020
 *      Author: takis
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "gol_config.h"
#include "cells.h"
```

```
/*
 * declare important variables (defined in main file as
global variables)
 */
//extern cell_t **env;
//extern cell_t **update_env;
extern cell_t env[config_NE][config_ME];
extern cell_t update_env[config_NE][config_ME];
extern bool reproduction_flag;

/*
 * PRIVATE FUNCTIONS
 */

/*
 * transfer a single community identified by the block-pair
(iT,jT) to env and
 * update_env using data_init[][]
 */
void transferCommunity(size_t iT, size_t jT,
         const cell_t data_init[config_NC][config_MC])
{
     size_t i_0 = iT * config_NC;
     size_t j_0 = jT * config_MC;

     printf("     ... transferring block (%d, %d)\n", (int)
(iT + 1),
               (int) (jT + 1));
     // copy this community to each community in env to
initialize it
     for (size_t i = 0; i != config_NC; ++i)
     {
```

```
          for (size_t j = 0; j != config_MC; ++j)
          {
               env[i_0 + i][j_0 + j] = update_env[i_0 +
i][j_0 + j] =
                         data_init[i][j];
          }
     }
}

/*
 * function counts the number of live neighbours of a cell
located
 * at row r and column c of the env array
 *
 * for reference, neighbours are designated as follows:
 *                a b c
 *              d X e
 *              f g h
 *
 *
 */
size_t countLiveNeighbours(size_t row, size_t col)
{
     size_t cell_count = 0;
     size_t R, C;

     // your code goes here
          for (int i=-1;i<2;i++)
          {
               for(int j=-1;j<2;j++)
               {

                    R = (size_t) (row + i + config_NE) %
```

```
config_NE;
                    C = (size_t) (col + j + config_ME) %
config_ME;
                    cell_count = cell_count +
(size_t)env[R][C];
                }
            }

            cell_count = cell_count -  (env[row][col]==live ?
1 : 0); // works only because live = 1, dead = 0;

        return cell_count;
}

/*
 * update cell located at row r and column c in env
(indicated by X):
 *
 *                  a b c
 *                  d X e
 *                  f g h
 *
 * with nearest neighbours indicated as shown from a, b,
..., h.
 *
 * this function features Conway's rules:
 *          - if a cell is dead but surrounded by exactly
three live neighbours, it sprouts to life (birth)
 *          - if a cell is live but has more than 3 live
neighbours, it dies (overpopulation)
 *          - if a cell is live but has fewer than 2 live
neighbours, it dies (underpopulation)
 *          - all other dead or live cells remain the same to
```

```
the next generation (i.e., a live cell must
 *          have exactly three neighbours to survive)
 *
 */
void updateCell(size_t r, size_t c)
{
     cell_t state_cell = env[r][c];
     size_t live_neighbours = countLiveNeighbours(r, c);


     if(state_cell==0 && live_neighbours==3)
     {
          update_env[r][c] = live;
     }
     else if(state_cell==1 &&( live_neighbours<2 ||
live_neighbours>3))
     {
          update_env[r][c] = dead;
     }
     else
     {
          update_env[r][c]=state_cell;
     }


     // your code goes here

}

/*
 * PUBLIC FUNCTIONS
 */
/*
```

```
 * seed environment on a community-by-community basis,
 * from standard input; we assume that the seed input is
exactly
 * the size of a community; 9999 indicates end of file;
 * run this before started ncurses environment;
 */
void initEnvironment(void)
{
     // start by reading in a single community
     int token;
     cell_t datum;
     cell_t community_init[config_NC][config_MC];

     printf("\ninitializing environment...\n");
     printf("     ... loading template community from
stdin\n");
     for (size_t i = 0; i != config_NC; ++i)
     {
          for (size_t j = 0; j != config_MC; ++j)
          {
               scanf("%d", &token);
               datum = (cell_t) token;
               community_init[i][j] = datum;
          }
     }
     printf("     ... done.\n");

     printf("     ... creating communities\n");
     // copy this community to each community in env to
initialize it
     for (size_t i = 0; i != config_K; ++i)
     {
          for (size_t j = 0; j != config_L; ++j)
```

```
          {
               transferCommunity(i, j, community_init);
          }
     }
     printf("     ... done.\n");

}
/*
 * write changes to the environment, env, from update_env
 */
void copyEnvironment(void)
{
     // copy this community to each community in env to
initialize it
     for (size_t i = 0; i != config_NE; ++i)
     {
          for (size_t j = 0; j != config_ME; ++j)
          {
               env[i][j] = update_env[i][j];
          }
     }
}

/*
 * this function updates all the cells for a thread
(corresponding to one community)
 */
void* updateCommFunc(void *param)
{
while(1){
     if(reproduction_flag)
          {
          threadID_t  *testing = param;
```

```
          size_t i_t = testing->row;
          size_t j_t = testing->col;

          size_t i_0 = i_t * config_NC;
          size_t j_0 = j_t * config_MC;

               for (size_t i = 0; i != config_NC; ++i)
               {
                    for (size_t j = 0; j != config_MC; ++j)
                    {
                    updateCell(i+i_0,j+j_0);
                    }
               }
          }



}


}
```

Makefile :

```
# define a variable to contain all of our object files (if
this list changes, we only need to change the variable)
objects = gol.o display.o cells.o

# define a variable for our include directory
include = includes

# specify a search path for all prerequisites; looks for
.o, .c or .h files in ./ or source/ or my_includes/
VPATH = source:includes

# provide a parameter for our implicit compilation rule
(for gcc)
CFLAGS = -I$(include)

# our project will be called ese3025_project and  it LINKS
object files gol.o, display.o and cells.o
ese3025_project : $(objects)
     gcc -o ese3025_project $(objects) -lpthread -lncurses

#compile gol.c to produce gol.o, if needed
gol.o : gol.c gol_config.h display.h cells.h

#compile display.c to produce display.o, if needed
display.o : display.c display.h gol_config.h

#compile cells.c to produce cells.o, if needed
cells.o : cells.c cells.h gol_config.h

# a clean option that will remove proj_ds and any object
files, with EXTRA features now!
.PHONY : clean  # declaring clean as "phony" prevents it
```

```
from being ignored if a file named "clean" happens to exist
in the directory!
clean :
     -rm ese3025_project $(objects)

#       ^
#       |_______ with the "-" in front of rm, the rm
command will continue to execute in spite of any errors
generated (such as "file does not exist")
```