WEEK #3: CONWAY'S GAME OF LIFE PROJECT IMPLEMENTATION

Subject: ESE_3025 EMBEDDED REAL-TIME OPERATING SYSTEM

Submitted by:

Name	Student ID	Submitted to
Zain Anwar Rajani	C0752681	Prof. Takis Zourntos

Software: Eclipse 06-2020
Programming Language: C Programming
Tools Used: Ncurses
Concept: pthreads implementation in RTOS
Host Machine: Linux (Ubuntu 18.04 (LTS))

Task Objective:

Using the "starter code" your task is to write three functions:

1) size_t countLiveNeighbours(size_t row, size_t col)

2) void updateCell(size_t r, size_t c)

3) void* updateCommFunc(void *param)

The third function is your thread function. Note that the display code uses ncurses and is working. All you need to do is get the game "engine" itself working. Also provided is initialization data, in the form of a text file. You simply "pipe" this data into your running program as follows:

$ cat seed_input_32_x_16.txt | ./proj_pthreads

Experiment with config_TL by making it short or long to see how it affects the code

Report Content

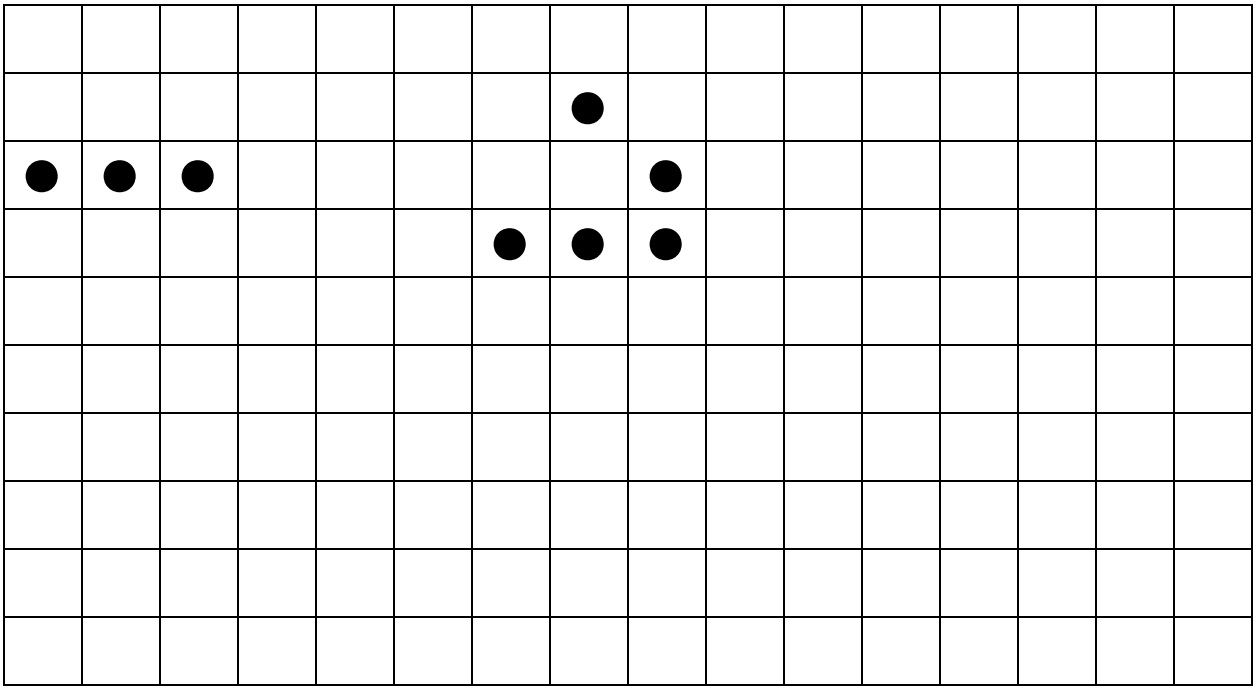
Understanding the Conway's Game of Life

The Game of life is dependent on a cell and its neighbours and the game is played in terms of generation. For a cell to go to next generation which we call a live for a second or next generation it has to see for its neighbours if there is no sufficient amount of neighbours or if it has excess of neighbours surrounding it we call as the cell as dead and it won't be present in the next generation of the game.

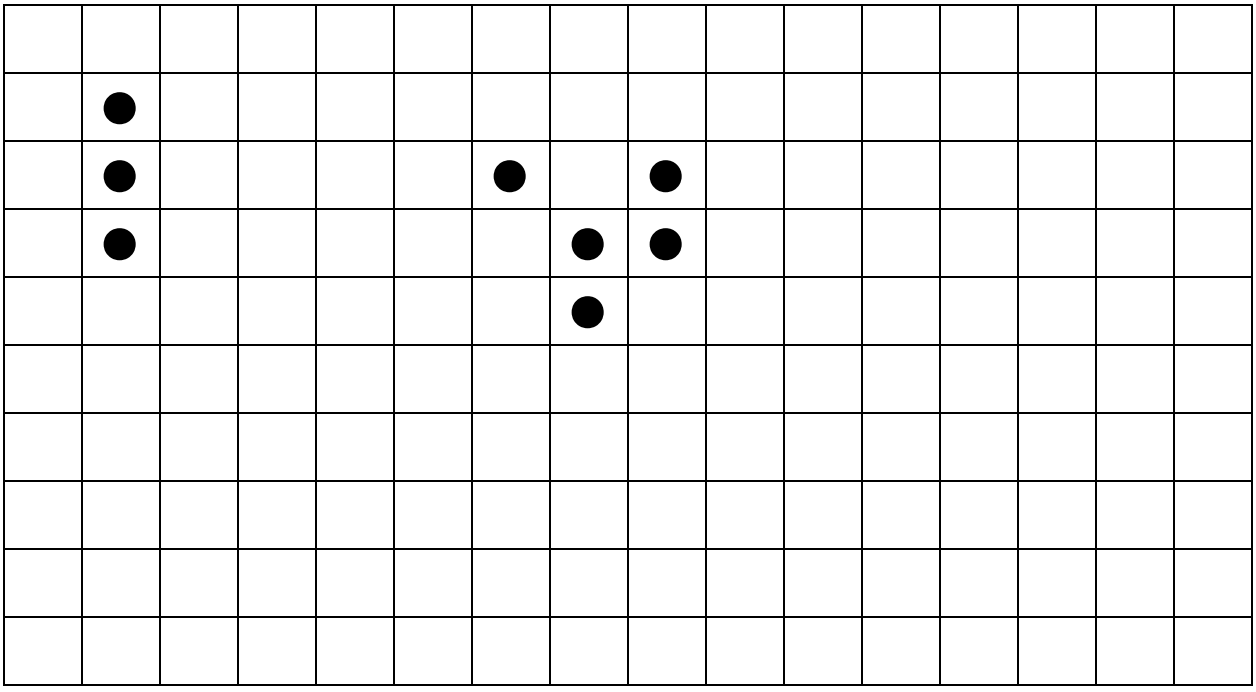
In the game thus we can have a cell live which would be denoted as logical '1' or we can have the cell dead denoted by logical '0'. The rules that decide if the cell is alive or dead are governed by the 3 rules as mentioned:

- If a cell has less than 2 neighbours i.e. just one single neighbour or no neighbour it can die of isolation. It is like having a lack of resources to survive.
- A cell can be born or live for the next generation of the game if and only if it has exactly 3 neighbours surrounding it.
- If a cell has more than 4 neighbours then the cell will be bound to die in its next generation due to overcrowding.

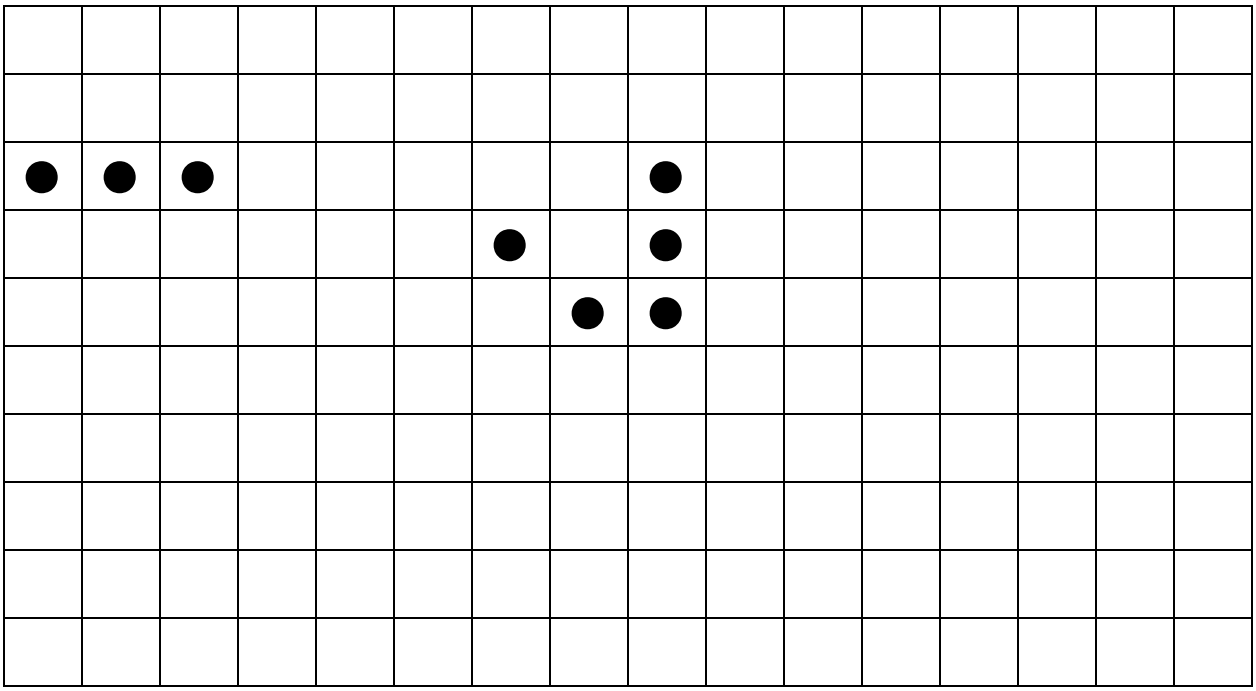
Lets illustrate this concept using a simple example of a glider as John Conway the inventor calls it. Consider we have a matrix of 10x16 and the initial or starting pattern as shown.



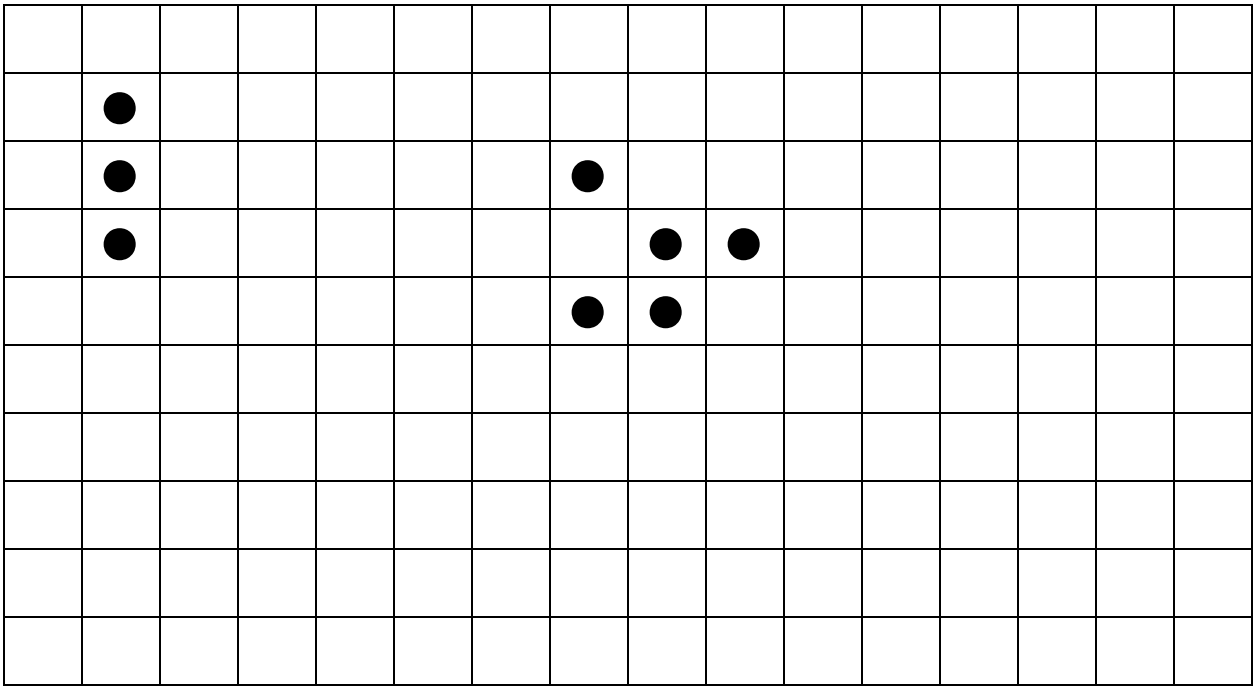
Second Generation of the Game:



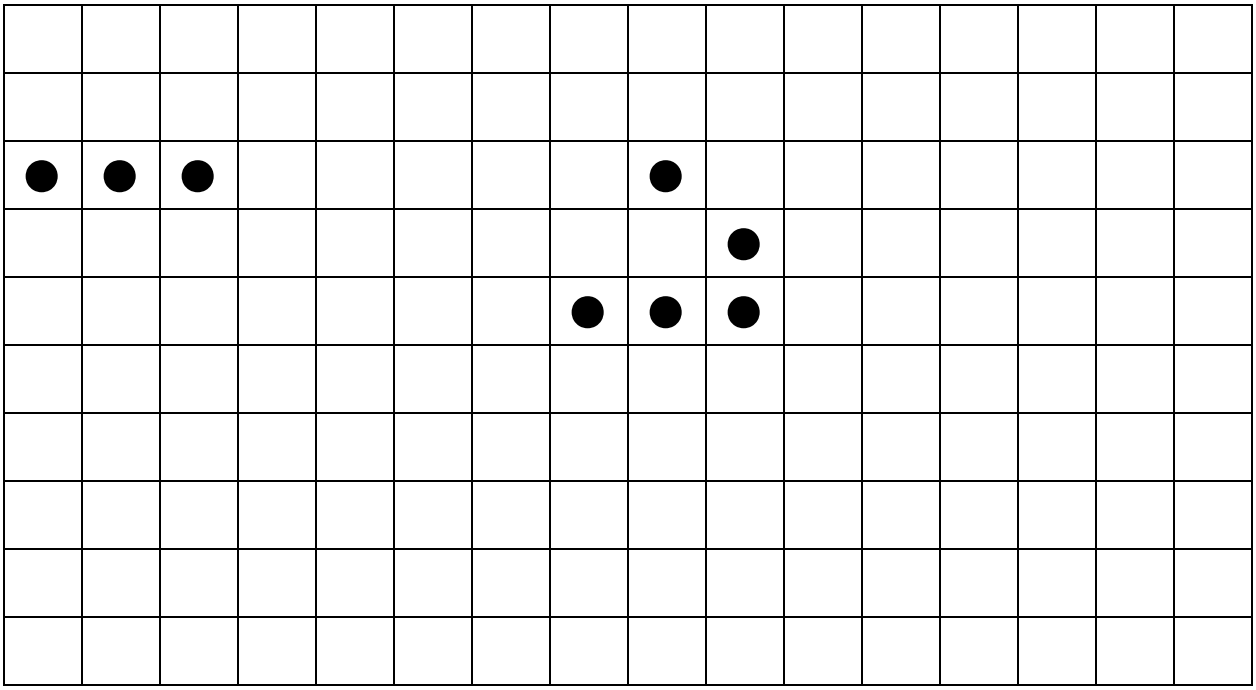
Third Generation of the Game:



Fourth Generation of the Game:



Fifth Generation of the Game:



As you can see in the example in the fifth generation we get back to initial started pattern and also the pattern the glider pattern on the right side is just shifted and it will be seen like crawling after each of the 5 generations of the game and the simple 3 horizontal pattern keeps on oscillating in alternative pattern thus called as the oscillatory pattern. After having the theoretical background of the game now let us look at how they accomplished the task given in the next section.

Task Accomplishment/ Proposal

Let us briefly see the outlook and then look at the actual code on how it was implemented.

1) size_t countLiveNeighbours(size_t row, size_t col)

In this we count the number of live neighbours around a given cell. Thus we already know that for a given cell there will be exactly 8 neighbours surrounding it. Thus we take each cell and use a counter to keep a track of all the neighbours surrounding it. For the cells that lie along the edges we use the wrap around condition that we think of the board as a circular pattern and to accomplish this task we use the modular division operation. We could have to make it simpler to also consider or assume that for the cells that lie on the edges the neighbours are all dead always. But in the implementation we used the wrap around condition. The code for the same looks as follows.

```
size_t countLiveNeighbours(size_t row, size_t col)
{
      size_t cell_count = 0; // Neighbour Count Variable

     // Loop for counting neighbours
      for (size_t i=row-1; i<=row+1;i++)
      {
            for (size_t j=col-1;j<=col+1;j++)
            {
                  // Loop Condition to make sure you don't count the
cell as its own neighbour
```

```
                if (i!=0 && j!=0)
                {
                      cell_count = cell_count +(size_t)env
[(i+config_NC)%config_NC][(j+config_MC)%config_MC];
                }
           }
     }

     return cell_count;
}
```

2) void updateCell(size_t r, size_t c)

As the name of the function says we need to update the given cell that we are taking into account. Now the point to note here is that from the previous function above the cell_count variable is used here. This function decides whether a cell stays alive or dead in the next generation based on the rules that we have already discussed earlier. The implementation is simple using loops i.e. if-else loop. Also we use the reproduction flag, that is this flag would go high only when the new cell will become alive. The above mentioned is implemented using the code shown below.

```
void updateCell(size_t r, size_t c)
{
     cell_t state_cell = env[r][c];  // Value Read
     // Function Call
     size_t live_neighbours = countLiveNeighbours(r, c);
     // Making neighbours dead or live as per the game rules
     if (state_cell==0 && live_neighbours==3)
     {
          update_env[r][c]=state_cell=live;
     }
```

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

3) void* updateCommFunc(void *param)

This is the last task of this project and this uses the concept of pthreads. Here we first take the param function and de-reference it using the dot operator and thus now we get the rows and columns of the matrix and achieve all this by type_casting the variables as if you refer the appendix at the end of this we would see that the param was of type thread_t and we have converted it into type of size_t and used it in our code. Using the for loop we access the values of the matrix which has 32 rows and 16 columns and each of them are replicated at 2 rows and 4 columns. In simple words we have a 2x4 matrix and each element of the matrix contains a 32x16 matrix. In this we later call the UpdateCell function and this function is only called when the reproduction flag is high. We have achieved this in the below shown code.

```
void* updateCommFunc(void *param)
{
      while (1) // Forever loop until 1==0
      {
            if (reproduction_flag) // enters if true
            {
                  // De-references
                  threadID_t *var=param;
                  size_t i_0=var->row;
```

```
                size_t j_0=var->col;
                size_t a=i_0*config_NC;
                size_t b=j_0*config_MC;
               // Loop to access cell data
                for (size_t i=0;i!=config_NC;++i)
                {
                      for (size_t j=0;j!=config_MC;++j)
                      {
                            // Function Call
                            updateCell(i+a,j+b);
                      }
                }
          }
     }
}
```

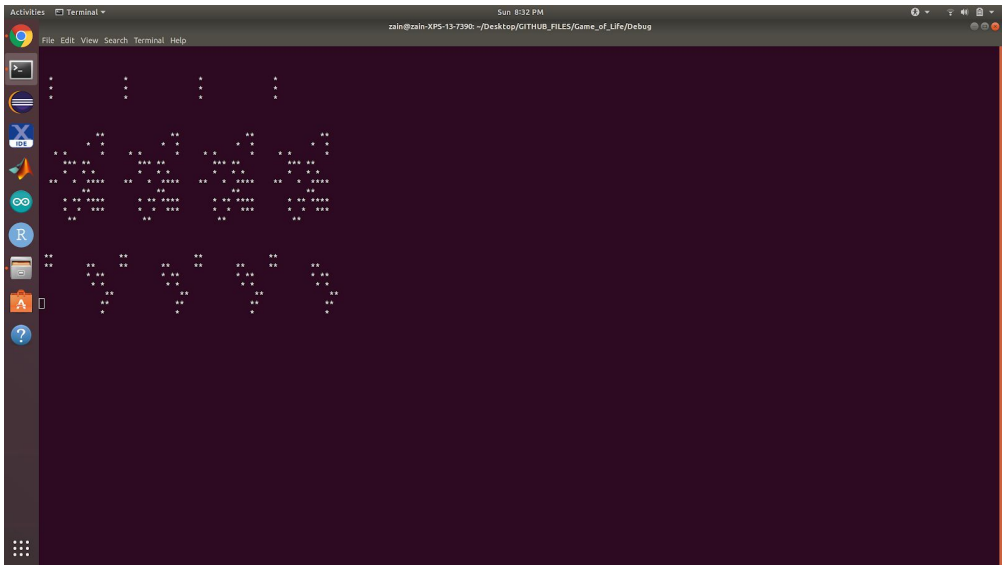
Lastly, we successfully were able to connect all the functions and a output was seen using the ncurses library in the linux terminal as follows. The screenshot below shows just one generation. A full video of the run can be found on the link mentioned below.

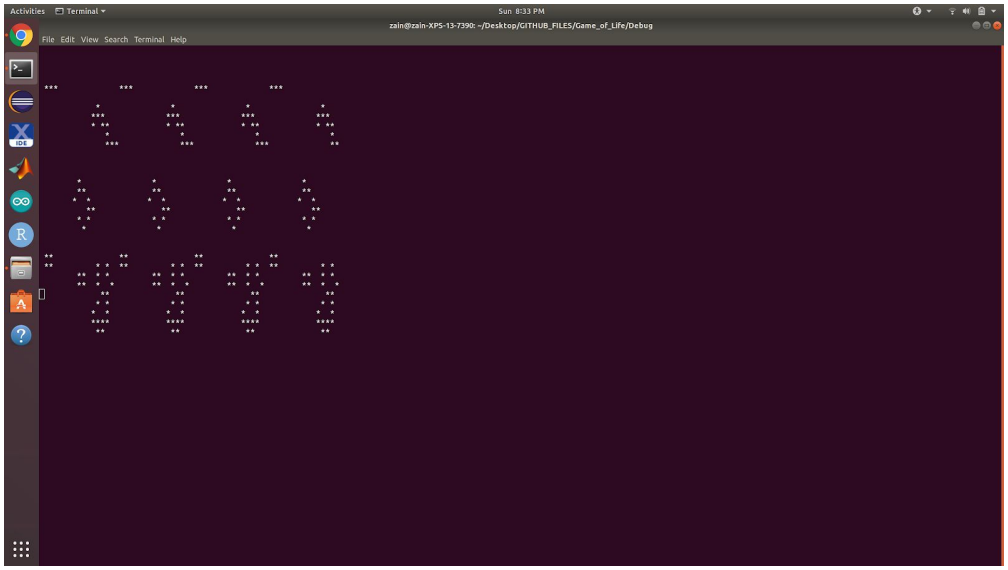
Full Video of the run: https://www.youtube.com/watch?v=_GVBSyjA3VA

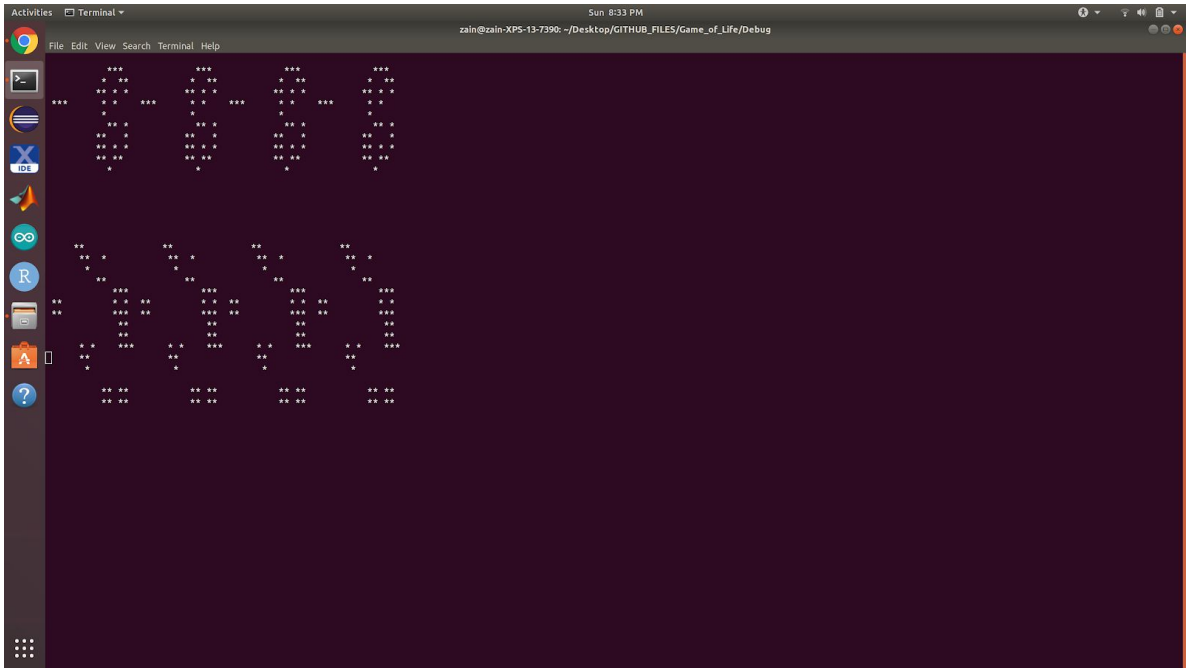
Project Outcome

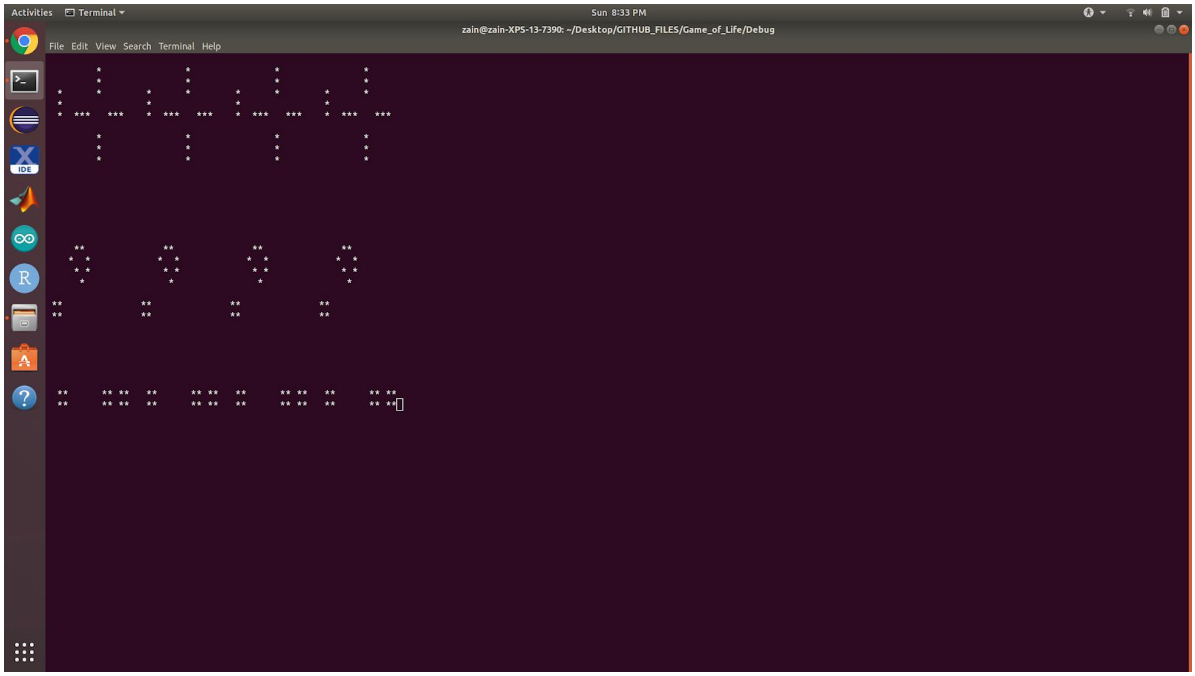
Full Video of the run: https://www.youtube.com/watch?v=_GVBSyjA3VA

Screenshot of some of the generation:









This is the stabilized generation with only an oscillatory pattern present.

Note: The entire code of the project is mentioned in the appendix just the source files no header files are included.

Conclusion/ Future Work:

The implementation of the game as whole was challenging at the first look but provides a deep insight into the working of pthreads and gives a good coding practice. This game stands as base and can be improved as well. The way it stands out is by the usage of threads in the game implementation. The features for the future could be:

- Implement this game involving more users

  This game is currently played for a single or better to say 0 player. But you can involve multiple players in this game. Like setting a fixed pattern and if the player touches a cell which is dead it will die off and repeated three die would mean you lose the game. Or we could also have a challenge game where the user decides the pattern and if the challenged player presses the live cell or opposite he loses the game.

- Use this for a screen saver pattern on computers/ mobile devices, etc

  We could have the patterns running on the screen when the screen goes off while the user is not working on the screen

- Used as an indication pattern

  Some patterns can be used to indicate direction for the driver on the screen using some LCD Displays

- Improving the Interface

  The display of the game in the current state using ncurses does not look good thus we can make it better using some other alternative using OPENGL or other kinds of alternatives. The best would be using a coloured screen with some animated figures.

APPENDIX

Header file : cell.h

```
#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_ */
```

Header file 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       32 // # of cell rows in a community
#define config_MC       16 // # of cell columns in a community

/*
 * overall environment parameters
 */
#define config_K        1 // # of communities "down"
#define config_L        4 // # of communities "across"
#define config_NE             config_K*config_NC // # of environment rows
#define config_ME             config_L*config_MC // # of environment col.

/*
 * temporal parameters
 */
#define config_TL       262144.5 // 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;


#endif /* GOL_CONFIG_H_ */
```

Header file: display .h

```
#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_ */
```

display.c file used ncurses here:

```
/*
 * display.c
 *
 *  Created on: May 30, 2020
 *      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 where all the operation regards to cells occur:

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

#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;

  for (size_t i = row - 1; i <= row + 1; i++)
    {
      for (size_t j = col - 1; j <= col + 1; j++)
      {
// To make sure that you don't count the cell whose neighbours are counted
        if (i != 0 && j != 0)
          {
            cell_count = cell_count + (size_t) env[(i + config_NC) %
config_NC][(j + config_MC)% config_MC];
          }
      }
    }
  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 4 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] = state_cell = live;
    }
  else if (state_cell == 1 && (live_neighbours >= 4 || live_neighbours <
2))
    {
      update_env[r][c] = state_cell = dead;
    }
}

/*
 * 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 *var = param;
        size_t i_0 = var->row;
        size_t j_0 = var->col;
        size_t a = i_0 * config_NC;
        size_t b = j_0 * config_MC;
        for (size_t i = 0; i != config_NC; ++i)
          {
            for (size_t j = 0; j != config_MC; ++j)
            {
              updateCell (i + a, j + b);
            }
          }
      }
    }
}
```

gol.c the main file of the game where the initial patterns are passed and set accordingly:

```
#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
      printf("\ncreating threads...%d\n",(int)index);
      threadID[index].row = i;
      threadID[index].col = j;
     //if condition returns 0 on the successful creation of 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();
      }
      // write changes to the environment, env,from update_env
      copyEnvironment();     }

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