Lab #1

**Game of Life**

Programming Language: C Programming

Tools Used: ncurses library as a C programming library FOR Linux providing an application programming interface that allows the programmer to write text-based user interfaces in a terminal-independent manner. It is a toolkit for developing "GUI-like" application software that runs under a terminal emulator.

Concept: pthreads implementation in RTOS

One of resources for getting into Conway's Game of Life:

https://web.stanford.edu/class/sts145/Library/life.pdf

Lab Instruction:

1. You might find three C programs of cells.c, display.c, and gol.c by cloning from:

https://github.com/takiszourntos/teaching/tree/master/lambton/2020/fall/ese3025/week\_2/project/proj\_pthreads\_template/source

1. For the cells. C code you need to complete the three following pthread functions wherever you see “**WRITE A PROPER STATEMENT HERE!**”:

size\_t countLiveNeighbours(size\_t row, size\_t col)

void updateCell(size\_t r, size\_t c)

void\* updateCommFunc(void \*param)

based on the theoretical background of the Game of Life described in the lecture.

Tip:

To assist you some part of the required functions are given below but you need to complete them:

a) 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;

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 = **WRITE A PROPER STATEMENT HERE!**;

//cell\_count=(size\_t)env[i-1][j-1]+(size\_t)env[i-1][j]+(size\_t)env[i-//1][j+1]+(size\_t)env[i][j-1]

//(size\_t)env[i][j+1]+(size\_t)env[i+1][j- //1]+(size\_t)env[i+1][j]+(size\_t)env[i+1][j+1];

}

}

}

cell\_count=**WRITE A PROPER STATEMENT HERE!**;

return cell\_count;

}

      b) 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];

size\_t live\_neighbours = countLiveNeighbours (r, c);

if (state\_cell == 0 && (live\_neighbours == 3))

{

update\_env[r][c] = state\_cell=live;

}

else if (**WRITE A PROPER STATEMENT HERE!**)

{

update\_env[r][c] =state\_cell= dead;

}

else if (**WRITE A PROPER STATEMENT HERE!**)

{

update\_env[r][c] =state\_cell= dead;

}

else

{

update\_env[r][c]=state\_cell;

}

}

c) 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)

{

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)

{

**WRITE A PROPER STATEMENT HERE!**;

}

}

}

}

}

1. Get your makefile finished and run a version of the GoL on Beaglebone (with an HDMI screen)
2. Analise the display.c and gol.c for Game of Life (GOL) project through commenting line by line