File : error

```

/*---------------------------------------------------------- */
/* AUTEUR : REYNAUD Nicolas                                  */
/* FICHIER : error.h                                         */
/*---------------------------------------------------------- */

#ifndef ERROR_H
#define ERROR_H

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

/**
 * If Debug Flag is on, create a maccro to print debug information
 * %param MSG : String to print
 * %param ... : List of param [ for example if want to print variable value ]
 */
#ifdef DEBUG
    #define DEBUG_MSG(MSG, ...)    \
    do {                           \
        fprintf(stderr, "\n\t[DEBUG] File : %s - Line : %d - Function : %s() : " MSG "\n",
            __FILE__, __LINE__, __func__, ## __VA_ARGS__);   \
    } while(0);
#else
    #define DEBUG_MSG(MSG, ...)
#endif

/**
 * Create a maccro for quit the program
 * %param MSG : String to print
 * %param ... : List of param [ for example if want to print variable value ]
 */
#define QUIT_MSG(MSG, ...)                           \
    do {                                             \
        DEBUG_MSG(MSG, ##__VA_ARGS__)                \
        fprintf(stderr, "[FATAL ERROR] ");           \
        fprintf(stderr, MSG, ## __VA_ARGS__);        \
        perror(NULL);                                \
        exit(EXIT_FAILURE);                          \
    }while(0);

#endif /* ERROR_H included */
```

File : main

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <curses.h>
#include <time.h>
#include <unistd.h>

#include "error.h"
#include "game.h"
#include "task.h"
#include "ncurses.h"
#include "option.h"
#include "thread.h"

int main(int argc, char* argv[]) {

    srand(time(NULL));

    clock_t time;

    Option o;
    Game* g = NULL;
    Task *t = NULL;
    ThreadInfo *ti = NULL;

    o = getOption(argc, argv);  /* Get all option     */

    if ( *o.file_path != '\0' ) /* If path file is not empty */
        if ( (g = loadBoard(o.file_path)) == NULL ) /* then use the given file [load id] */
            fprintf(stderr, "Can't load file %s\n", o.file_path);
```

```c

    if ( g == NULL ) /* If load of file fail Or no grid given */
        g = generateRandomBoard(o); /* then create one */

    if ( o.use_ncurses ) /* If we use ncurses */
        initNCurses();   /* Then we init the display */

    if ( o.nb_thread == 0 ) { /* If there is no threa given, then we use sequential version */
        t = newTask(0, g->cols - 1); /* And said to the main thread to threat all columns */
    } else {
        ti = newThreadInfo(o.nb_thread, g);
        createNThread(ti);
    }

    time = clock();
    while(o.max_tick != 0) {          /* Inifinit loop if total tick not given */

        gamePrintInfo(g, o);

        if ( o.nb_thread == 0 ) {     /* if there is 0 thread then do not use thread method */
            gameTick(g, t);             /* Lets the game tick */
        } else {
            createTask(ti, o.use_fine_grained);
            runThread(ti);
        }

        __swapGrid(g);
        --o.max_tick;

        #ifdef PRINT                  /* If we print we add some delay without it we can't see the grid */
            usleep(400000);
        #endif
    }

    time = clock() - time;
    printf("Time : %f\n", (double)(time) / CLOCKS_PER_SEC);

    if ( o.use_ncurses) /* If we use ncurses ( and then init it ) */
        endNCurses();   /* we need to clear display info */

    if ( o.save_file )
        saveBoard(g);

    if ( o.nb_thread == 0 ) {
        free(t);
    } else {
        endNThread(ti);
        freeThreadInfo(ti);
    }

    freeGame(g);            /* Free space we are not in Java */

    exit(EXIT_SUCCESS);
}
```

File : game_struct

```c
/*----------------------------------------------------------- */
/* AUTEUR : REYNAUD Nicolas                                   */
/* FICHIER : game_struct.h                                    */
/*----------------------------------------------------------- */


#ifndef GAME_STRUCT_H
#define GAME_STRUCT_H

/**
 * Struct that represent a game
 */
typedef struct {
    char *current_board; /* The board as an array of 0's and 1's. */
    char *next_board;    /* The new board */
    unsigned int cols;   /* The number of columns. */
    unsigned int rows;    /* The number of rows. */
} Game;
#endif
```

File : game

```c
/*---------------------------------------------------------- */
/* AUTEUR : REYNAUD Nicolas                                  */
/* FICHIER : game.h                                          */
/*---------------------------------------------------------- */


#ifndef GAME_H
#define GAME_H

#include "game_struct.h"
#include "option_struct.h"

#include "task_pile_struct.h"

/**
 * First need to define all the constante
 */
#define MIN_COLS_SIZE 5
#define MIN_ROWS_SIZE 3

#define MAX_COLS_SIZE 50
#define MAX_ROWS_SIZE 30
#define POURCENT_BEEN_ALIVE 15

#define DEAD_CELL 0
#define ALIVE_CELL 1

/**
 * Given X, and Y this function output the position into the board.
 * For example POS(0,0,G) return 0, cause the cell in 0 on X, and 0 on Y is the cell 0 of
     the board
 * %param X : Position on the X coordinate
 * %param Y : Position on the Y coordinate
 * %param G : Board on which we need to compute the position
 * %return  : The associate position on the board
 */
#define POS(X, Y, G) (__position(X,Y,G))

/**
 * Function that print the board, this function determine if we need to print it or not
 * i.e if the programme is make with make display
 * This function also determine which function we need to use to display the board, and
     print the
 * number of generation left.
 *
 * %param g : The game which contains the board to print
 * %param o : Option which include the use_ncurses option
 */
void gamePrintInfo ( Game* g, Option o);

/**
 * Function that free the memory associate with a game
 * %param g : Game to free
 */
void freeGame(Game* g);

/**
 * Function that generate a random board if no are given
 * %param o : Option for generating the board
 * %return  : a random board
 */
Game* generateRandomBoard(Option o);

/**
 * Function that make the game tick, i.e function that iterate through the game board
 * and complet the other board.
 * %param g : game with contains the board on which we need to iterate
 * %param t : task that need to be done on this board
 */
void gameTick(Game *g, Task *t);

```

```c
/**
 * Private function that is used in the main program
 * This function swap the 2 board of the game
 * %param g : Game which contains the 2 board to swap
 */
void __swapGrid(Game* g);

/**
 * Load in memory a game / board contains into a file
 * %param name : path to the file to load
 * %return      : The game structure associate with the contenant of the file
 *                Or NULL if that fail [i.e the file is not valide
 */
Game* loadBoard(char* name);

/**
 * Function that save a game into a file
 * %param g : the board to save
 * %return  : true if it succeed
 *            false otherwise
 */
bool saveBoard(Game *g);

#endif
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <curses.h>

#include "error.h"
#include "game.h"
#include "game_struct.h"
#include "memory.h"

/**
 * Private function that compute the position of the board given a x and a y
 * %param x : Position on the X coordinate
 * %param y : Position on the Y coordinate
 * %param g : Game where we need to compute the cell position
 * %return  : Position of the cell associate with the X and Y coordinate
 */
int __position(unsigned int x, unsigned int y, Game* g) {
    return g->cols * y + x;
}

/**
 * Private function that print a simple line
 * %param g : Game structure which contains the information relative to the game
 * %param pf : Pointer to a print function
 */
void __printLine(Game* g, int (*pf)(const char *, ...)) {
    unsigned int i = 0;

    (*pf)("+");
    for ( i = 0; i < g->cols + 2; i++ ) /* the 2 '+'  */
        (*pf)("-");
    (*pf)("+\n");

}

/**
 * Private function that really print the board contenant
 * %param g : Game struct which contains the board to print
 * %param pf : Pointer to a printing function
 */
void __gamePrint (Game* g, int (*pf)(const char *, ...)) {
    unsigned int x, y;

    if ( *pf == printw ) /* If we use ncurses we need to replace cursor */
        move(0, 0);

    (*pf)("Board size : \n");
    (*pf)("  %d Columns\n", g->cols);
    (*pf)("  %d rows\n", g->rows);

    __printLine(g, pf);

    for ( y = 0; y < g->rows; y++) {
        (*pf)("| ");
        for ( x = 0; x < g->cols; x++) {
            (*pf)("%c", ((g->current_board[POS(x, y, g)] == DEAD_CELL) ? '.' : '#'));
        }

        (*pf)(" |\n");
    }

    __printLine(g, pf);

    if ( *pf == printw ) /* If we use ncurses we need to refresh the display */
        refresh();

    DEBUG_MSG("Print board finish\n");
}

void __swapGrid(Game* g) {
    char *tmp = g->current_board;
    g->current_board = g->next_board;
    g->next_board = tmp;
}

void gamePrintInfo(Game* g, Option o) {
    #ifndef PRINT
         return;
    #endif

    int (*printFunc)(const char*, ...);
    printFunc = ( o.use_ncurses ) ? &printw : &printf;

    if ( o.max_tick >= 0 )
        printFunc("%d Generation left.\n", o.max_tick);

    __gamePrint(g, printFunc);
}

/**
 * Private function that allocate a new board
 * %param rows : Total number of rows onto the board
 * %param cols : Total number of column onto the board
 * %return     : Allocated array of char which will contains the board
 */
char* __newBoard(unsigned int rows, unsigned int cols) {
    char* board = NEW_ALLOC_K(rows * cols, char);
    return board;
}

/**
 * Private function that create a new game
 * %param rows : Total number of rows onto the new board
 * %param cols : Total number of Column onto the new board
 * %return     : Allocated game structure which contains all the information
 */
Game* __newGame(unsigned int rows, unsigned int cols) {
    Game* g = NEW_ALLOC(Game);

    g->rows = rows;
    g->cols = cols;

    g->current_board = __newBoard(rows, cols);
    g->next_board = __newBoard(rows, cols);

    return g;
}

void freeGame(Game* g)  {
    if ( g == NULL )
        return;

    free(g->current_board);
    free(g->next_board);
    free(g);
}

```

```c
Game* generateRandomBoard(Option o) {

    unsigned int rows = 0, cols = 0;
    Game* g;

    g = __newGame(o.rows, o.cols);

    DEBUG_MSG("Ligne : %d, Cols : %d\n", o.rows, o.cols);
    for (rows = 0; rows < g->rows; rows++)
        for(cols = 0; cols < g->cols; cols++)
            g->current_board[POS(cols, rows, g)] = (
                ( rand() % 100 >= POURCENT_BEEN_ALIVE ) ?
                    DEAD_CELL:
                    ALIVE_CELL
                );
    DEBUG_MSG("Generate random finish");
    return g;
}

/**
 * Private function which compute the total number of neighbour of a cell
 * %param x : X position of the cell on the board
 * %param y : Y position of the cell on the board
 * %param g : Game struct wich contains all information relative to the game
 * %return  : Total number of neighbour of this cell
 */
int __neighbourCell(unsigned int x, unsigned int y, Game *g) {
    unsigned int total = 0;
    char *b = g->current_board;

    if ( x % g->cols != g->cols - 1) {
        total += b[POS(x + 1, y,     g)]; /* Right */
        if ( y < g->rows - 1 ) total += b[POS(x + 1, y + 1, g)]; /* Right - Down */
        if ( y > 0 )           total += b[POS(x + 1, y - 1, g)]; /* Up - Right */
    }

    if ( x % g->cols != 0 ) {
        total += b[POS(x - 1, y     , g)]; /* Left */
        if ( y < g->rows - 1 ) total += b[POS(x - 1, y + 1, g)]; /* Left - Down */
        if ( y > 0 )           total += b[POS(x - 1, y - 1, g)]; /* Up - Left */
    }

    if ( y < g->rows - 1 ) total += b[POS(x     , y + 1, g)]; /* Down */
    if ( y > 0 )           total += b[POS(x     , y - 1, g)]; /* Up   */

    return total;
}

/**
 * Private function which process a cell, i.e update the cell on the other board according
     to ome rules
 * %param x : Position on X of the cell on the board
 * %param y : Position on Y of the cell on the board
 * %param g : Game struct which contains all information relative to the game
 * %return  : New state of the cell in x / y coordinate.
 */
char __process(unsigned int x, unsigned int y, Game* g) {
    unsigned int neightbour = __neighbourCell(x, y, g);

    if ( neightbour < 2 || neightbour > 3 ) return DEAD_CELL;
    else if ( neightbour == 3 )             return ALIVE_CELL;
    else                                    return g->current_board[POS(x, y, g)];
}

void gameTick(Game *g, Task* t) {

    unsigned int x, y;

    for (y = 0; y < g->rows; y++)
        for(x = t->min; x ≤ t->max; x++)
            g->next_board[POS(x, y, g)] = __process(x, y, g);

    DEBUG_MSG("Game tick finish");
}

Game* loadBoard(char* name) {
    char reader = ' ';
    unsigned int rows = 0, cols = 0;
    FILE* fp = NULL;
    Game *g = NULL;

    if ( (fp = fopen(name, "r")) == NULL ) return NULL;
    if ( fscanf(fp, "Rows : %d\nCols : %d\n", &rows, &cols) != 2) { fclose(fp); return NULL
    ; }

    g = __newGame(rows, cols);

    DEBUG_MSG("Rows : %d, Cols : %d\n", rows, cols);
    rows = 0; cols = 0; /* Reinit variable */

    while ( (reader = fgetc(fp)) != EOF ) {
        if ( reader == '.' ) reader = DEAD_CELL;
        if ( reader == '#' ) reader = ALIVE_CELL;

        if ( reader == '\n') ++rows;
        else g->current_board[POS(cols, rows, g)] = reader;

        if ( ++cols > g->cols ) cols = 0; /* We are going to go over cols due to \n */
    }

    fclose(fp);

    if ( cols != g->cols && rows != g->rows ) { freeGame(g); return NULL; }
    return g;
}

bool saveBoard(Game *g) {
    unsigned int i;
    FILE *fp = NULL;

    if ( (fp = fopen("output.gol", "w")) == NULL ) return false;

    fprintf(fp, "Rows : %d\nCols : %d\n", g->rows, g->cols);
    for ( i = 0; i < g->cols * g->rows; i++ ) {

        fprintf(fp, "%c", ((g->current_board[i]) ? '#' : '.') );
        if ( i % g->cols == g->cols - 1 ) fprintf(fp, "\n");
    }

    #ifdef PRINT
        printf("File saved into : output.gol\n");
    #endif

    fclose(fp);
    return true;
}
```

File : memory

```c
/*----------------------------------------------------------- */
/* AUTEUR : REYNAUD Nicolas                                   */
/* FICHIER : memory.h                                         */
/*----------------------------------------------------------- */


#ifndef MEMORY_H
#define MEMORY_H

#include <stdlib.h>

/**
 * Function that allocate a single object
 * %param OBJECT : Object type to allocate
 * %return       : Pointer in memory associate with the object Type.
 */
#define NEW_ALLOC(OBJECT) (NEW_ALLOC_K(1, OBJECT))

/**
 * Function that allocate an array of the same Object
 * %param K      : Total number to allocate
 * %param OBJECT : Object type to allocate
 * %return       : Pointer in memory associate with the object type.
 */
```

```c
#define NEW_ALLOC_K(K, OBJECT) (__memAlloc(K, sizeof(OBJECT)))

/**
 * Private function that shouldn't be used
 * The definition of this function is in memory.c
 */
void *__memAlloc(int total, size_t object_size);

#endif
```

```c
#include "error.h"
#include "memory.h"

/**
 * Private function that board the allocation of an object
 * %param total : Total number of object that we need to allocate
 * %param object_size : Size of the object which we need to allocate
 * %return : Pointer on the memory associate with the new object
 */
void *__memAlloc(int total, size_t object_size) {

    void *p = calloc(total, object_size);

    if ( p == NULL )
        QUIT_MSG("Canno't allocate new object\n");

    return p;

}
```

File : ncurses

```c
/*------------------------------------------------------------ */
/* AUTEUR : REYNAUD Nicolas                                    */
/* FICHIER : ncurses.h                                         */
/*------------------------------------------------------------ */


#ifndef NCURSES_H
#define NCURSES_H

/**
 * Function that initialize NCurses
 */
void initNCurses();

/**
 * Function that end NCurses, i.e free memory associate with NCurse
 */
void endNCurses();

#endif
```

```c
#include <stdio.h>
#include <curses.h>

void initNCurses() {
    initscr();
    noecho();
}

void endNCurses() {
    printw("End of generation\nUse anykey for leave\n");
    refresh();
    getch();
    endwin();
}
```

File : option_struct

```c
/*------------------------------------------------------------ */
/* AUTEUR : REYNAUD Nicolas                                    */
/* FICHIER : error.h                                           */
/*------------------------------------------------------------ */

#ifndef OPTION_STRUCT
#define OPTION_STRUCT
```

```c

#include <stdbool.h>

/**
 * Structure that will contains all of the option
 */
typedef struct Option {
    int max_tick;              /* How much tick we need to do               - Default : 100 */
    char* file_path;           /* Path to the file to load                  - Default : "" */
    unsigned int nb_thread;    /* Total number of thread to use             - Default : 0  */
    unsigned int rows;         /* Number of rows to generate                - Default : Random
         */
    unsigned int cols;         /* Number of columns to generate             - Default : Random
         */
    bool use_ncurses;          /* Do we use ncurses for the display ?       - Default : false
         */
    bool use_fine_grained;     /* Do we use fine grain in multi thread ? - Default : false
         */
    bool save_file;            /* Do we need to save the last grid ?        - Default : false
         */
} Option;

#endif
```

File : option

```c
/*---------------------------------------------------------- */
/* AUTEUR : REYNAUD Nicolas                                  */
/* FICHIER : error.h                                         */
/*---------------------------------------------------------- */


#ifndef OPT
#define OPT

#include "option_struct.h"

/* List of possible option */
#define OPT_LIST "hf:t:np:r:c:gs"

/** Use the definition defined by David Titarenco
 *  On StackOverFlow http://stackoverflow.com/questions/3437404/min-and-max-in-c
 */
#define MAX(a,b) \
   ({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a > _b ? _a : _b; })

/**
 * Print the usage of the program
 * %param name : name of the program
 */
void usage(char* name);

/**
 * Function that get all command line option and return those one into a structure
 * %param argc : Total number of argument onto the command line
 * %param argv : Contenant of all the command line
 * %return      : Structure which contains all option given onto command line into this
       structure
 */
Option getOption(int argc, char** argv);

#endif
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

#include "game.h"
#include "option.h"

void usage(char* name) {
    printf("%s [-h]\n\t\t [-f <filePath>] [-t <maxTick>] [-c <number cols] [-r <number rows
        ] [-g] [-n] [-s]\n\n", name);
    printf("\t\t -h : print this help\n");
    printf("\t\t -f filePath : path to the file to use for the grid\n");
    printf("\t\t -t maxTick : max time to make the game tick, set it to negatif for
        infinite tick\n");
    printf("\t\t -c : total numner of column\n");
    printf("\t\t -r : total number of rows\n");
    printf("\t\t -n : use ncurses for the display\n");
    printf("\t\t -g : if -g set then we use fine grained method\n");
    printf("\t\t -s : if -s is use the final grib will be saved\n");

    exit(EXIT_SUCCESS);
}

/**
 * Private function that define the default value for the option
 * %return : The option struct with the default value
 */
Option __setDefaultValue() {
    Option o;

    o.use_fine_grained = false;
    o.use_ncurses = false;
    o.file_path = "\0";
    o.max_tick = 100;
    o.nb_thread = 0;
    o.save_file = false;

    o.rows = rand() % ( MAX_ROWS_SIZE - MIN_ROWS_SIZE) + MIN_ROWS_SIZE;
    o.cols = rand() % ( MAX_COLS_SIZE - MIN_COLS_SIZE) + MIN_COLS_SIZE;

    return o;
}

Option getOption(int argc, char **argv) {
    int opt = 0;
    Option o = __setDefaultValue();

    while ( (opt = getopt(argc, argv, OPT_LIST)) != -1 ) {
        switch(opt) {
            case '?':
            case 'h':
                usage(argv[0]);
                break;
            case 'f':
                if ( optarg != 0 )
                    o.file_path = optarg;
                break;
            case 't':
                o.max_tick = atoi(optarg);
                break;
            case 'n':
                o.use_ncurses = true;
                break;
            case 'p':
                o.nb_thread = MAX(atoi(optarg), 0);
                break;
            case 'r':
                o.rows = MAX(atoi(optarg), MIN_ROWS_SIZE);
                break;
            case 'c':
                o.cols = MAX(atoi(optarg), MIN_COLS_SIZE);
                break;
            case 'g':
                o.use_fine_grained = true;
                break;
            case 's':
                o.save_file = true;
                break;
            default:
                exit(EXIT_FAILURE);
        }
    }

    if ( argc == 1)
        fprintf(stderr, "Remember to use -h for help\n");

    return o;
}
```

File : task_pile_struct

```c
/*------------------------------------------------------------ */
/* AUTEUR : REYNAUD Nicolas                                    */
/* FICHIER : task_pile_struct.h                                */
/*------------------------------------------------------------ */

#ifndef TASK_PILE_STRUCT
#define TASK_PILE_STRUCT

/**
 * A task consist into a start column and an ending column
 */
typedef struct Task {
    unsigned int min;        /* Where the task need to start, i.e. column where start on
        board */
    unsigned int max;        /* Where the task need to end,   i.e column where stop on
        board   */

    struct Task* next_task;  /* Pointer to the next task */
} Task;

/**
 * A task pile is a stack of task, i.e FILO
 */
typedef struct TaskPile {
  Task *first;               /* Pointer on the first task of the task pile */
} TaskPile;


#endif
```

File : task

```c
/*------------------------------------------------------------ */
/* AUTEUR : REYNAUD Nicolas                                    */
/* FICHIER : task.h                                            */
/*------------------------------------------------------------ */

#ifndef TASK
#define TASK

#include "task_pile_struct.h"

/**
 * Function that create a new task
 * %param min : Column where the task will start
 * %param max : Column where the task will end
 * %return    : A new allocated task
 */
Task *newTask(int min, int max);

/**
 * Function which add a task to the pile
 * %param tpi : task pile where we need to add the task
 * %param t   : The task to add
 */
void insertTask(TaskPile *tpi, Task *t);

/**
 * Function which check if the pile is empty
 * %param tp : Task pile to check
 * %return   : true if the pile is empty
 *             false otherwise
 */
bool isEmpty(TaskPile* tp);

/**
 * Function which get a task
 * %param tp : Task pile where we need to get a task
 * %return   : A task of the task pile
 */
Task *getTask(TaskPile* tp);

/**
 * Private function that shouldn't be used [only use in thread.c]
 * The definition of this function is in task.c [all information are there]
 */
```

```c
void __freeTaskPile(TaskPile* tp);

#endif
```

```c
#include <stdbool.h>

#include "error.h"
#include "task.h"
#include "memory.h"

Task *newTask(int min, int max) {
    Task *t;

    if ( min > max )
        QUIT_MSG("max need to greater than min\n");

    t = NEW_ALLOC(Task);
    t->min = min;
    t->max = max;
    t->next_task = NULL;

    return t;
}

void insertTask(TaskPile *tp, Task *t) {
    DEBUG_MSG("We add task from : %d to %d |Taks : %p - Next : %p|\n", t->min, t->max, t, t
        ->next_task);

    if ( tp->first != NULL )
        t->next_task = tp->first;

    tp->first = t;
}

bool isEmpty(TaskPile *tp) {
    return tp->first == NULL;
}

Task *getTask(TaskPile *tp) {
    Task *t = NULL;
    if ( isEmpty(tp) )
        return NULL;

    t = tp->first;
    tp->first = t->next_task;

    DEBUG_MSG("Get : %p | tp->first : %p\n", t, tp->first);
    return t;
}

/**
 * Private function that free the task pile and all remaining task if there is some
 * %param tp : Pointer of the pile which we need to free
 */
void __freeTaskPile(TaskPile *tp) {
    while (!isEmpty(tp))
        free(getTask(tp));

    free(tp);
}
```

File : thread

```c

    TaskPile *task_pile; /* list of all task (could be empty is keep_task is set */
    pthread_t *plist;
} ThreadInfo;

#endif
```

```c
/*---------------------------------------------------------- */
/* AUTEUR : REYNAUD Nicolas                                  */
/* FICHIER : thread.h                                        */
/*---------------------------------------------------------- */

#ifndef THREAD
#define THREAD
```

```c

#include <stdbool.h>

#include "game_struct.h"
#include "thread_struct.h"

/**
 * Function that create new thread information, this function also set the real number of
     usefull thread
 * %param n : Total number of thread that need to be create
 * %param g : Game struct which contains all information relative to the game where thread
     going to iterate
 * %return  : Thread information containing all revelent information
 */
ThreadInfo *newThreadInfo(unsigned int n, Game *g);

/**
 * Function which free the thread info
 * %param ti : Thread info to free
 */
void freeThreadInfo(ThreadInfo *ti);

/**
 * Function which create a new task
 * %param ti : thread information which contains lock and all revelent information about
     thread
 * %param fine_grained : Bool which say if we use the fine grained method or not
 */
void createTask(ThreadInfo *ti, bool fine_grained);

/**
 * Function that make the thread run [.ie broadcast a start message ]
 * %param ti : Thread information which contains all revelent information about thread [
     Mutex etc ]
 */
void runThread(ThreadInfo *ti);

/**
 * Function that create N thread, according to the thread info struct
 * %param ti : thread information structure which contains information about the thread
 *            [ total number of thread / mutex etc ]
 */
void createNThread(ThreadInfo *ti);

/**
 * Function which stop all thread, this function make them run after said that they need to
      stop
 * then it wait for all thread
 * %param ti : Thread information struct which contains the list of thread
 */
void endNThread(ThreadInfo *ti);

#endif
```

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#include "error.h"
#include "memory.h"
#include "thread.h"
#include "game.h"
#include "task.h"

ThreadInfo *newThreadInfo(unsigned int n, Game *g) {
    ThreadInfo *ti;

    ti = NEW_ALLOC(ThreadInfo);

    ti->g = g;
    ti->n = n;
    ti->total_end = 0;
    ti->should_end = false;
    ti->keep_task = false;
    ti->lock_work = (pthread_mutex_t) PTHREAD_MUTEX_INITIALIZER;
    ti->lock_end =  (pthread_mutex_t) PTHREAD_MUTEX_INITIALIZER;
    ti->lock_end_cond = (pthread_cond_t) PTHREAD_COND_INITIALIZER;

    ti->task_pile = NEW_ALLOC(TaskPile);
    ti->plist =  NEW_ALLOC_K(n, pthread_t);

    if ( n > g->cols ) { /* If there is more thread than needed, then adjust the value */
        #ifdef PRINT
            fprintf(stderr, "[INFO] %d thread is/are useless\n", n - g->cols );
        #endif
        ti->n = g->cols; /* change default value */
    }

    return ti;
}

void freeThreadInfo(ThreadInfo *ti) {
    __freeTaskPile(ti->task_pile);
    pthread_mutex_destroy(&ti->lock_work);
    pthread_mutex_destroy(&ti->lock_end);
    pthread_cond_destroy(&ti->lock_end_cond);
    free(ti->plist);
    free(ti);
}

void createTask(ThreadInfo *ti, bool fine_grained) {
    unsigned int i;
    unsigned int j = 0;
    int slice_size = 0;
    Task *t = NULL;

    if ( ti->keep_task )
        return;

    pthread_mutex_lock(&ti->lock_work);

    slice_size = (!fine_grained ) ? ((int) ti->g->cols / ti->n) - 1 : 0; /* Calculate slice
         size */

    DEBUG_MSG("Slice Size : %d \n", slice_size + 1 );
    for ( i = 0; i < ti->g->cols; i = (t->max - t->min + 1) + i, ++j ) { /* J contains
        number of thread used */
        t = NEW_ALLOC(Task);

        t->min = i;                   /* The start of slice start at the last one done  */
        t->max = t->min + slice_size; /* And end at : The start + the slice size */

        if ( !fine_grained && j == ti->n - 1 ) /* If we are at the last thread available in
             average grained */
            t->max += ti->g->cols % ti->n;      /* Then give it the remaining column */

        if ( !fine_grained && t->max >= ti->g->cols ) t->max = ti->g->cols - 1; /* If we
            don't use the fine grained, then add missing column to */

        insertTask(ti->task_pile, t);
    }

    pthread_mutex_unlock(&ti->lock_work);

    /* set it to true when we've got some task and enought thread for never switch them */
    if ( (ti->g->cols == ti->n || !fine_grained) && !ti->keep_task )
        ti->keep_task = true;
}

/**
 * Private function which will lock the mutex, take a task and free the mutex
 * %param ti  : ThreadInfo struct which contains all of the mutex / lock etc
 * %return    : A Task
 */
Task *__threadGetTask(ThreadInfo *ti) {
    Task *t = NULL;

    pthread_mutex_lock(&ti->lock_work);
    t = getTask(ti->task_pile);
    pthread_mutex_unlock(&ti->lock_work);

    return t;
```

```c
}

/**
 * Private function which wait that all thread have finish there task
 * %param ti : ThreadInfo structu which contains all information relative to thread
 */
void __waitTickEnd(ThreadInfo* ti) {
    while (ti->total_end != ti->n) { usleep(5000); }
}

void runThread(ThreadInfo* ti) {
    __waitTickEnd(ti); /* Wait that all have end there task before restart */
    pthread_mutex_lock(&ti->lock_end);

    ti->total_end = 0;
    pthread_cond_broadcast(&ti->lock_end_cond);
    pthread_mutex_unlock(&ti->lock_end);

    __waitTickEnd(ti); /* Wait all have finish before give hand to main */
}

/**
 * Private function that wait for all thread
 * %param ti : Thread info struct which contains all information relative to thread
 */
void __waitAllTick(ThreadInfo* ti) {
    pthread_mutex_lock(&ti->lock_end);
    ++ti->total_end;

    DEBUG_MSG("%d out of %d have finish, wait all others\n", ti->total_end, ti->n);
    pthread_cond_wait(&ti->lock_end_cond, &ti->lock_end);

    pthread_mutex_unlock(&ti->lock_end);
}

/**
 * Private function that process a thread action
 * When this function start, it wait for a broadcast to start
 * Then process is task and either wait or process a new task
 * %param ti : Thread info struct wich contains all information relative to thread
 */
void __processThread(ThreadInfo* ti) {
    Task *t = NULL;
    __waitAllTick(ti);

    while (!ti->should_end) {

        if ( t == NULL )
            t = __threadGetTask(ti);

        if ( t != NULL )
            gameTick(ti->g, t);

        if ( !ti->keep_task ) {
            free(t);
            t = NULL;
        }

        if ( ti->keep_task || isEmpty(ti->task_pile) )
            __waitAllTick(ti);
    }

    /* If we keep task then free that one are associated with the thread */
    if ( ti->keep_task )
        free(t);

    pthread_mutex_lock(&ti->lock_end);
    ++ti->total_end;
    pthread_mutex_unlock(&ti->lock_end);
}

void createNThread(ThreadInfo *ti) {
    unsigned int i = 0;

    for ( i = 0; i < ti->n; i++)
        if ( pthread_create(&ti->plist[i], NULL, (void*) __processThread, ti) ) /* Create
```

```
                the thread here */
                QUIT_MSG("Can't create thread %d\n", i);
}

void endNThread(ThreadInfo *ti) {
    unsigned int i;
    ti->should_end = true;
    runThread(ti);
    for ( i = 0; i < ti->n; i++) {
        if ( pthread_join(ti->plist[i], NULL) )
            QUIT_MSG("Error while join thread %d\n", i);
    }

    DEBUG_MSG("All thread have finish they work\n");
}
```