Unicamp
MC504 - Sistemas Operacionais

Projeto 1
Aplicação Multithread
Jogo da Vida

Antonio Gabriel da Silva Fernandes RA 231551
Daniel Mendes dos Santos RA 214752
Lindon Jonathan Sanley dos S. P. Monroe RA 220407
Lucas de Paula Soares RA 201867

Campinas
2022

1.Introdução

O projeto em questão trabalha com o processamento multithread do jogo da vida de Conway que, em síntese, consiste no processamento matricial em quadrantes para cada thread utilizada.

Diante disso, basta compreender as regras jogo da vida de Conway e como elas são representadas no projeto:

- No início da execução do código, deve ser dado o tamanho da Matriz M (m x n) separados por espaço;
- Toda célula viva é representada por “#”;
- Toda célula morta é representada por “.”;
- Qualquer célula viva com menos de dois ou com mais de três vizinhos vivos morre;
- Qualquer célula morta com exatamente três vizinhos vivos se torna uma célula viva;
- Qualquer célula viva com dois ou três vizinhos vivos continua no mesmo estado para a próxima geração;

A partir disso, estabeleceu-se que a entrada do projeto é dada aleatoriamente na forma:

```
9 9
.#..##.##
#...#.#..
.....##.#
##.#....#
.#.#.#..#
####.#.##
.#.##.#.#
#....#.#.
##..#....
```

Figura 1. Matriz aleatória

Dessa maneira a matriz pode ser lida e processada em ciclos pelo algoritmo projetado pelo grupo.

2. Algoritmo

A estratégia proposta pelo grupo fora a de, diante da matriz apresentada, criar quadrantes de tamanho até 5x5 (fixamente) para ser processado por threads individualmente e, para caso do quadrante não ser 5x5, ele é tratado como sendo um quadrante menor a x b tal que $a, b \in \mathbb{N} \mid a <= 5, b <= 5$. Ou seja, para no caso de um primeiro quadrante a ser tratado na matriz da **figura 1** teríamos:

Figura 2. Primeiro quadrante

Seguindo essa lógica, para o caso, dos extremos mais à direita e inferior da matriz teria-se quadrantes com lado menor do que o limite como por exemplo o último quadrante que será 4X4:

Figura 3. Último quadrante

A partir disso, em ambos os casos seus processamentos são dados por uma leitura da vizinhança de todas as células do quadrante ignorando os limites matriciais e a própria célula e em seguida sobrescrevendo o novo estado da célula em uma nova matriz.

3. Código e Funções

3.1 Main

Dado o algoritmo apresentado, basta apenas traduzi-lo para um código em C que utilize a pthread. Ou seja, para as definições iniciais, teremos a inclusão das bibliotecas e da definição das constantes de limitação da matriz:

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

#define QUAD_WDT 5
#define QUAD_HGT 5
```

Figura 4. Definições e chamada de bibliotecas

Com isso, deve-se analisar a função main a qual começa alocando e escaneando a matriz conforme a entrada dada na introdução e, em seguida, faz-se a divisão em quadrantes para a criação das threads matricialmente.

```c
int main() {
    int M, N;
    scanf("%d %d\n", &M, &N);
    char **matriz = malloc_matriz(M,N);

    for(int i = 0 ; i < M ; i++){
        for(int j = 0 ; j < N ; j++){
            scanf("%c ", &matriz[i][j]);
        }
    }

    // Cálculo da quantidade de quadrantes em cada dimensão:
    const int thr_num_hgt = M / QUAD_HGT + ((M % QUAD_HGT == 0) ? 0 : 1); // teto(M / QUAD_HGT)
    const int thr_num_wdt = N / QUAD_WDT + ((N % QUAD_WDT == 0) ? 0 : 1); // teto(N / QUAD_WDT)
    pthread_t thr[thr_num_hgt][thr_num_wdt];
```

Figura 5. Main (parte 1)

Em seguida, inicia-se o fluxo principal do código, onde, em uma repetição intermitente se cria a nova matriz a ser exibida no próximo ciclo e, percorre-se a matriz de threads e passando como argumentos para “f_thread” a dimensão da matriz, localização de início da thread na matriz e a matriz lida. Sendo tudo isso auxiliado por uma estrutura “f_thread_args” para maior clareza no código.

```
while(1){
    char **prox_matriz = malloc_matriz(M, N);

    for(int i = 0 ; i < thr_num_hgt ; i++){
        for(int j = 0 ; j < thr_num_wdt ; j++){
            f_thread_args* args = malloc(sizeof(f_thread_args));
            if(args == NULL) return 1;
            args->matriz = matriz;
            args->prox_matriz = prox_matriz;
            args->M = M;
            args->N = N;
            args->i = i;
            args->j = j;
            pthread_create(&thr[i][j], NULL, f_thread, (void*) args);
        }
    }
```

Figura 6. Main (parte 2)

```
// Struct usada para enviar os argumentos da função f_thread
typedef struct {
    char **matriz;
    char **prox_matriz;
    int M;
    int N;
    int i;
    int j;
} f_thread_args;
```

Figura 7. Main (parte 3)

Feita essas execuções da função, a matriz já estará processada (função melhor explicada no tópico seguinte) e basta apenas lidar com a impressão da matriz e sua desalocação e com a anulação das matriz de threads para um novo ciclo a ser executado em seguida.

```
            for(int i = 0 ; i < thr_num_hgt ; i++){
                for(int j = 0 ; j < thr_num_wdt ; j++){
                    pthread_join(thr[i][j], NULL);
                }
            }

            print_matriz(matriz, M, N);
            free_matriz(matriz, M);
            matriz = prox_matriz;

            sleep(1);
        }

        free_matriz(matriz, M);
        return 0;
```

Figura 8. Main (parte 4)

3.2 Funções

Para criação e tratamento da matriz de entrada, criou-se as funções de alocação e desalocação de matriz, as quais basicamente alocam memória dinamicamente com o "malloc" e liberam-na com o "free".

```
char** malloc_matriz(int M, int N){
    char **matriz;
    int i;

    matriz = (char**) malloc(sizeof(char*) * M);
    if(matriz == NULL){
        printf("Memoria insuficiente.\n");
        exit(1);
    }
    for(i = 0; i < M; i++){
        matriz[i] = (char*) malloc(sizeof(char) * N);
        if(matriz[i] == NULL){
            printf("Memoria insuficiente.\n");
            exit(1);
        }
    }
    return matriz;
}

// Libera a memória alocada para uma matriz com M linhas
void free_matriz(char **matriz, int M){
    for(int i = 0 ; i < M ; i++){
        free(matriz[i]);
    }
    free(matriz);
}
```

Figura 9. Alocação e desalocação

Por fim, basta-se entender as 2 funções na qual todo o código se sustenta, pois elas fazem o processamento das threads:

- **get_vizinhos_vivos:** Analisa toda a vizinhança da célula em questão e verifica se sua condição no próximo ciclo será de viva ("#") ou morta (".").

```c
int get_vizinhos_vivos(char **matriz, int M, int N, int i, int j){
    int count = 0;

    for(int x = i - 1 ; x <= i+1 ; x++){
        for(int y = j - 1 ; y <= j+1 ; y++){
            if(x < 0 || x >= M || y < 0 || y >= N) continue; // Célula fora da matriz
            if(x == i && y == j) continue; // Própria célula
            count += (matriz[x][y] == '#' ? 1 : 0);
        }
    }

    return count;
}
```

Figura 10. get_vizinhos_vivos

- **f_thread:** Aqui é onde se localiza a parte central do código, onde basicamente, com os argumentos recebidos pela estrutura percorre-se o quadrante (delimitado pelos argumentos) e analisa como o auxílio da função **get_vizinhos_vivos** o próximo estado da célula.

```c
// Processa o quadrante da matriz associado à ID (i, j) da thread
void* f_thread(void *void_args) {
    f_thread_args *args = (f_thread_args*) void_args;
    for(int x = args->i * QUAD_HGT ; x < (args->i + 1) * QUAD_HGT ; x++){
        for(int y = args->j * QUAD_WDT ; y < (args->j + 1) * QUAD_WDT ; y++){
            if(x < 0 || x >= args->M || y < 0 || y >= args->N) continue; // Célula fora da matriz
            int vizinhos = get_vizinhos_vivos(args->matriz, args->M, args->N, x, y);
            if(args->matriz[x][y] == '#'){
                if(vizinhos == 2 || vizinhos == 3) args->prox_matriz[x][y] = '#';
                else args->prox_matriz[x][y] = '.';
            }
            else{
                if(vizinhos == 3) args->prox_matriz[x][y] = '#';
                else args->prox_matriz[x][y] = '.';
            }
        }
    }

    free(args);
    return NULL;
}
```

Figura 11. get_vizinhos_vivos