Sistemas Operacionais
Roteiro de Laboratório 4

1 Objetivos

O objetivo deste laboratório é ter um contato prático com a biblioteca Pthreads para criação de *threads* no Linux.

2 *Threads*

Uma *thread* (*light-weight process*) é uma unidade básica de utilização de CPU, sendo formada por um *thread* ID, um conjunto de registradores e uma pilha. Os demais recursos tais como área de código, área de dados e outros recursos do SO são compartilhados por todas as *threads* que fazem parte de um mesmo processo. Um processo tradicional (*heavy-weight*) começa com uma única *thread* de controle (`main()`) que pode criar outras *threads*. Nesta aula de laboratório vamos aprender a usar a biblioteca Pthreads (POSIX *threads*).

3 POSIX *threads*

Em arquiteturas multi-processador (ou multi-*core*), *threads* podem ser usadas para implementar paralelismo. Historicamente, cada fabricante de *hardware* implementava a sua própria versão de biblioteca de *threads*, o que limitava a portabilidade dos programas. Para sistemas UNIX, uma API unificada foi definida pelo padrão IEEE POSIX 1003.1c. Implementações que seguem esse padrão são chamadas de POSIX *threads* ou Pthreads.

Ao contrário de outras bibliotecas (como por exemplo, OpenMP), Pthreads é uma biblioteca de *multi-threading* explícito. Isso quer dizer que o programador é responsável por gerir todos os aspectos de manipulação/uso de *threads* de forma explícita, isto é, escrevendo código dedicado para tal fim. Assim, se quisermos usar padrões de programação paralela, nós devemos escrever explicitamente o código correspondente a esses padrões.

Um padrão recorrente no uso de Pthreads é o padrão de *fork-join*, ilustrado abaixo.

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

// Child thread will run this function.
void *greetings(void *param) {
    printf("Greetings from the child thread!\n");
    pthread_exit(0); // Terminate thread normally.
}

int main(void) {
    pthread_t tid; // Thread identifier.
    pthread_attr_t attr; // Set of thread attributes.

    // Get the default attributes.
```

```
    pthread_attr_init(&attr);
    // Create the thread.
    pthread_create(&tid, &attr, greetings, NULL);
    // Wait for the thread to exit.
    pthread_join(tid, NULL);

    return 0;
}
```

Exemplo de compilação e execução.

```
$ gcc -o thread0 thread0.c -lpthread
$ ./thread0
Greetings from the child thread!
```

No próximo exemplo, a *thread* principal cria um filho para realizar a soma de um vetor.

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

#define ARRAY_SIZE 10

// Global data is shared by all threads.
int nums[ARRAY_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum;

// Child thread will run this function.
void *do_sum(void *param) {
    for (int i = 0; i < ARRAY_SIZE; i++) {
        sum += nums[i];
    }
    pthread_exit(0); // Terminate thread normally.
}

int main(void) {
    pthread_t tid; // Thread identifier.
    pthread_attr_t attr; // Set of thread attributes.

    // Get the default attributes.
    pthread_attr_init(&attr);
    // Create the thread.
    pthread_create(&tid, &attr, do_sum, NULL);
    // Wait for the thread to exit.
    pthread_join(tid, NULL);

    printf("Sum = %d\n", sum);
    return 0;
}
```

Exemplo de compilação e execução.

```
$ gcc -o thread1 thread1.c -lpthread
$ ./thread1
Sum = 55
```

Por definição do padrão, a função que vai executar na *thread* só recebe um único argumento `void*`. Caso seja necessário passar mais de um argumento, temos que criar uma estrutura dedicada para tal, como mostra o exemplo abaixo.

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

// Struct for parameter passing to threads.
typedef struct {
  int *nums;
  int size;
} Parameters;

// Global data is shared by all threads.
int sum;

// Child thread will run this function.
void *do_sum(void *params) {
    int *nums = ((Parameters*) params)->nums;
    int size = ((Parameters*) params)->size;
    for (int i = 0; i < size; i++) {
        sum += nums[i];
    }
    pthread_exit(0); // Terminate thread normally.
}

int main(void) {
    int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    Parameters params;
    params.nums = nums;
    params.size = 10;

    pthread_t tid; // Thread identifier.
    pthread_attr_t attr; // Set of thread attributes.

    // Get the default attributes.
    pthread_attr_init(&attr);
    // Create the thread.
    pthread_create(&tid, &attr, do_sum, &params);
    // Wait for the thread to exit.
    pthread_join(tid, NULL);

    printf("Sum = %d\n", sum);
    return 0;
}
```

O próximo exemplo ilustra uma primeira tentativa para um modelo de paralelismo de dados, aonde criamos uma *thread* para realizar cada passo da soma `c[i] = a[i] + b[i]`. No entanto, o programa abaixo possui um erro sutil. Procure encontrá-lo.

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

#define ARRAY_SIZE 10

// Global data is shared by all threads.
int a[ARRAY_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int b[ARRAY_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int c[ARRAY_SIZE];

#define NUM_THREADS 10

// Child threads will run this function.
void *do_sum(void *param) {
    int i = *((int*) param);
    c[i] = a[i] + b[i];
    pthread_exit(0); // Terminate thread normally.
}

int main(void) {
    pthread_t workers[NUM_THREADS]; // Array of worker threads.
    pthread_attr_t attr; // Set of thread attributes.
    int i;

    // Get the default attributes.
    pthread_attr_init(&attr);
    // Create the threads.
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_create(&workers[i], &attr, do_sum, &i);
    }
    // Wait for the threads to exit.
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(workers[i], NULL);
    }

    // Print the result.
    printf("C[]:");
    for (i = 0; i < ARRAY_SIZE; i++) {
        printf(" %d", c[i]);
    }
    printf("\n");

    return 0;
}
```

O problema fica mais visível quando compilamos e rodamos o programa.

```
$ gcc -o thread3_errado thread3_errado.c -lpthread
$ ./thread3_errado
C[]: 2 0 6 0 10 12 14 16 18 0
$ ./thread3_errado
C[]: 2 0 6 0 10 12 14 16 18 20
```

Algumas posições do vetor não estão sendo preenchidas. Isso ocorre porque estamos passando o endereço para a variável de índice `i`, e nada garante que no momento que uma *thread* estiver executando, o valor de `i` ainda seja o mesmo que quando a *thread* foi criada.

Para consertar o programa anterior, precisamos garantir que cada *thread* receba uma cópia individual da variável `i`. Isso leva a um padrão comum de criação de parâmetros das funções de *threads* como abaixo.

```
for (i = 0; i < NUM_THREADS; i++) {
    int *local_i = malloc(sizeof *local_i);
    *local_i = i; // Create a local copy.
    pthread_create(&workers[i], &attr, do_sum, local_i);
}
```

Uma vez que estamos fazendo alocação de memória com `malloc`, precisamos garantir que não haja vazamento de memória. Por conta disso, a função das *threads* fica assim agora:

```
void *do_sum(void *param) {
    int i = *((int*) param);
    c[i] = a[i] + b[i];
    free(param); // To avoid memory leak.
    pthread_exit(0); // Terminate thread normally.
}
```

É interessante notar que não é obrigatório o uso de alocação no *heap*. Basta garantir que os argumentos de cada *thread* sendo criada fiquem imutáveis. O código abaixo também resolve o problema.

```
int local_i[NUM_THREADS];
for (i = 0; i < NUM_THREADS; i++) {
    local_i[i] = i;
    pthread_create(&workers[i], &attr, do_sum, &local_i[i]);
}
```

4 Tarefa

Um jogo de *Sudoku* utiliza um *grid* 9×9 aonde em cada linha e coluna, bem como em cada um dos nove 3×3 *subgrids*, todos os nove dígitos $(1, \ldots, 9)$ devem aparecer. As macros em C abaixo mostram uma solução correta e uma incorreta para o Sudoku, implementadas como uma matriz de duas dimensões.

```
#define GOOD_SOLUTION {\
  {6, 2, 4, 5, 3, 9, 1, 8, 7},\
  {5, 1, 9, 7, 2, 8, 6, 3, 4},\
  {8, 3, 7, 6, 1, 4, 2, 9, 5},\
  {1, 4, 3, 8, 6, 5, 7, 2, 9},\
  {9, 5, 8, 2, 4, 7, 3, 6, 1},\
  {7, 6, 2, 3, 9, 1, 4, 5, 8},\
  {3, 7, 1, 9, 5, 6, 8, 4, 2},\
  {4, 9, 6, 1, 8, 2, 5, 7, 3},\
  {2, 8, 5, 4, 7, 3, 9, 1, 6}\
}

#define BAD_SOLUTION {\
  {6, 2, 4, 5, 3, 9, 1, 8, 7},\
  {5, 1, 9, 7, 2, 8, 6, 3, 4},\
  {8, 3, 7, 6, 1, 4, 2, 9, 5},\
  {1, 4, 3, 8, 6, 5, 7, 2, 9},\
  {9, 5, 8, 2, 1, 7, 3, 6, 1},\
  {7, 6, 2, 3, 9, 1, 4, 5, 8},\
  {3, 7, 1, 9, 5, 6, 8, 4, 2},\
  {4, 9, 6, 1, 8, 2, 5, 7, 3},\
  {2, 8, 5, 4, 7, 3, 9, 1, 6}\
}

char s[9][9] = GOOD_SOLUTION;
// char s[9][9] = BAD_SOLUTION;
```

A tarefa deste laboratório consiste em desenvolver um programa *multi-threaded* que determina se uma solução para o Sudoku é válida ou não. Você deve gerar 27 *threads* no total, divididas como a seguir.

- Nove *threads* para analisar cada uma das linhas da solução.
- Nove *threads* para analisar cada uma das colunas da solução.
- Nove *threads* para analisar cada um dos *subgrids* 3×3.

Será necessário criar três funções distintas para cada um dos tipos de *threads* acima. A *thread* principal deve criar as 27 *threads* de computação (`workers`), passando para cada uma a região do Sudoku que deve ser analisada. A forma mais simples para se passar esses parâmetros é através de uma estrutura como abaixo.

```
typedef struct {
    int row;
    int column;
} Parameters;
```

Veja os exemplos anteriores para determinar como passar os parâmetros individuais de cada *thread* corretamente. Se você utilizar alocação dinâmica (`malloc`), certifique-se que o seu programa não vaza memória. (Use, por exemplo, a ferramenta `valgrind` para testar.)

Envie a sua implementação pelo AVA na tarefa correspondente. O prazo (inadiável) para submissão é dia 03/10/2017 (terça-feira) às 23:55. Essa tarefa vale um *bit* (1 – fez corretamente dentro do prazo, 0 – caso contrário) na nota dos exercícios de laboratório.