Atividade Threads

Samuel Cavalcanti

2 de março de 2021

Testando o algoritmo de Fibonacci

Para resolver o problema proposto, foi inicialmente implementado um algoritmo que gera a sequência de Fibonacci dado o ultimo index desejado. e foi feito um teste simples que comprar a lista gerada pelo algoritmo com uma lista de valores esperados retirados do wikipedia (2008). Tanto o algoritmo e o seu teste pode ser visto no lib.rs. Durante os testes do algoritmo implementado foi descoberto que a partir do index 47, o algoritmo falha, uma vez que seu valor é muito maior que é possível armazenar em uma variável inteira de 32bits e foi adicionado um teste para exemplificar esse fato. Uma vez sabendo que o algoritmo funciona dada a sua limitação, foi decidido que essa parte do exercício foi resolvida.

Listing 1: lib.rs

```rust
use std::collections::LinkedList;

pub fn fibonacci(number: usize) -> LinkedList<i32> {
    let mut fibonacci_numbers: LinkedList<i32> = LinkedList::new();
    fibonacci_numbers.push_back(0);
    if number == 1 {
        return fibonacci_numbers;
    }

    fibonacci_numbers.push_back(1);
    let mut size = fibonacci_numbers.len();
    if number == size {
        return fibonacci_numbers;
    }

    let mut n_element: i32 = 0;

    let mut n_1_element: i32 = 1;

    while size < number {
        size = fibonacci_numbers.len();

        let option = n_1_element.checked_add(n_element);

        match option {
            Some(next_fibonacci_number) => {
```

```
                fibonacci_numbers.push_back(next_fibonacci_number);
                n_element = n_1_element;
                n_1_element = next_fibonacci_number;
            }
            None => {
                eprintln!(
                    "Nao foi possivel adicionar, variavel inteira
                        de 32 bits atingiu seu limite!!"
                );
                return fibonacci_numbers;
            }
        }
    }

    return fibonacci_numbers;
}

mod test_fibonacci {
    use std::collections::LinkedList;

    use super::fibonacci;

    #[test]
    fn test_fibonacci() {
        let mut expected = LinkedList::new();

        expected.push_back(0);
        assert_eq!(expected, fibonacci(1));
        expected.push_back(1);
        assert_eq!(expected, fibonacci(2));

        for element in vec![
            1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
                987, 1597, 2584, 4181, 6765,
        ] {
            expected.push_back(element);
        }

        assert_eq!(expected, fibonacci(20))
    }

    #[test]
    fn test_overflow_fibonacci() {
        let fibonacci_numbers = fibonacci(100);

        assert_eq!(fibonacci_numbers.len(), 47);
    }
}
```

Criando uma Command line interface (CLI)

Em Rust, uma das bibliotecas mais utilizadas cara criar CLIs é o command line Argument Parser (clap), um App clap criado possui a seguintes propriedades: autor, sobre, argumentos, versão. No author foi colocado o meu nome e meu e-mail, sobre, foi escrito uma breve descrição da Atividade e para essa aplicação, só é necessário um único argumento, que representa quantos números da sequencia de Fibonacci será exibida. Essa parte do código foi recortada e pode ser vista separadamente. 2

Listing 2: CLAP

```
  let matches = clap_app!(tarefa_threads =>
  (author: "Samuel Cavalcanti <scavalcanti111@gmail.com>")
  (about: "Tarefa Threads, o usuario digitara na linha de comandos
      a quantidade de numeros de Fibonacci que o programa deve
      gerar")
  (@arg Number: +required "Digite a quantidade de numeros de
      fibonacci")
  (version: "1.0")
).get_matches();
```

Programação paralela em Rust

Assim como **C++**, rust possui smart pointers, esses ponteiros especiais desalocam a memória automáticamente, quando nenhum ponteiro está apontando para ela, em rust esse ponteiros são chamados de **Arc** Klabnik and Nichols (2019). Como por padrão a linguagem Rust não permite o compartilhamento de variáveis entre diferentes threads, se faz necessário o uso desse ponteiro **Arc**, a qual podemos utilizar o método **clone** e passar uma referência para a outra thread. Existem diferentes formas seguras de compartilhar memória entre threads, uma delas é através de um **Mutex**, um Mutex, é uma abreviação de mutual exclusion, que basicamente, permite que apenas uma thread tem acesso a uma variável por vez e para ter acesso a essa variável é preciso chamar uma função especial chamada **lock**, que faz basicamente, retorna a variável e impede que qualquer outra thread tenha acesso a essa variável. Como dito anteriormente, Rust não permite que uma mesma variável esteja presente em duas threads, por isso também se faz o uso o **move**, que *move* as variáveis da função main, para a **função anônima** que é passada na função **thread::spawn**, que *spawna* uma thread. Funções anônimas são presentes em várias linguagens de programação como Kotlin, python e Javascript, basicamente são funções que não possui nomes, em Rust outra diferença que ao invés de utilizar **()**, usa-se as barras: ||, observe que utilizando o **move**, não foi necessário passar valores, como parâmetro, mas implicitamente, estou passando um referência do **Mutex** e a variável **input**.

Listing 3: Programação paralela

```
    let mutex = Arc::new(Mutex::new(LinkedList::new()));

    let mutex_thread_child = Arc::clone(&mutex);
    let handle = thread::spawn(move || {
        let mut mutex_fibonacci_numbers = mutex_thread_child.lock()
            .unwrap();
        *mutex_fibonacci_numbers = fibonacci(input);
    });
    handle.join().unwrap();
```

Tratamento de Erro em Rust

Em Rust existe dois tipos de Erros, irrecuperáveis e os recuperáveis. Erros recuperáveis são por exemplo, arquivos não encontrados, http requests, **parse** de strings. Já erros irrecuperáveis, são os erros sinônimos de bugs como, acesso indevido de endereço de memória, estouro da pilha e outros, quando acontece esses erros não é possível trata-los em tempo de execução. Nó código apresentado, existem diferentes situações onde pode ocorrer falhas, a primeira delas é no **parse** da string que supostamente deveria ser um número positívo. Em Rust, quando algo é passível de erro, é retornado ou uma variável do tipo **Result** ou do tipo **Option**. No caso do **Result** ele pode ser de dois tipos de estruturas, **Ok** ou **Error**. No caso do **parse** , o **Result** do tipo **Ok** *retorna* o valor de uma variável inteira de 32bits sem sinal, se for do tipo **Error**, ele retorna o erro do parse. Já o **Option** pode ser de dois tipos: **Some** ou **None**, onde **Some** pode ser qualquer tipo de objeto **enum** e **None** seria quando não retorna nada. Verificar erros em Rust pode ser algo bem cansativo e as vezes desnecessário, por isso tanto **Result** quando **Option**, possui métodos como **unwrap** ou **expect**, o **unwrap** retorna o valor dentro do tipo **Ok** ou **Some**, mas caso não seja desses tipos, a aplicação se encerra com uma mensagem de erro e caso queria editar essa mensagem, pode utilizar o **expect**.

Listing 4: Momentos que foi utilizando unwrap e expect

```
let input = matches
      .value_of("Number")
      \\ compilador nao sabe se he possivel
      \\ achar o argumento Number
      .unwrap()
      .parse::<usize>()
      \\ encerra o programa caso nao seja possivel
      \\ fazer o parse do argumento da funcao.
      .expect("\nError nao foi possivel fazer o parse do input.
          Um numero positivo era o esperado.\n");


\* o compilador nao sabe dizer se he possivel fazer o lock da
    lista ligada */
let mut mutex_fibonacci_numbers = mutex_thread_child.lock().
    unwrap();

\\ o compilador nao sabe dizer se he possivel esperar a thread.
handle.join().unwrap();
```

Apesar, de que em quase todo o código é possível não tratar o erro, dado que se o argumento não for transformando para o tipo **usize**, não há nada o que fazer ou quando sabemos de antemão que não irá haver dead-locks, ou erros que impeçam o **join**. Houve uma vez que foi preciso tratar o erro, o erro do overflow 5.

Listing 5: Momento que foi necessário tratar o Option

```
let option = n_1_element.checked_add(n_element);

match option {
    Some(next_fibonacci_number) => {
        fibonacci_numbers.push_back(next_fibonacci_number);
        n_element = n_1_element;
        n_1_element = next_fibonacci_number;
    }
    None => {
        eprintln!(
            "Nao foi possivel adicionar, variavel inteira de 32
                bits atingiu seu limite!!"
        );
        return fibonacci_numbers;
    }
}
```

Neste caso, após observar que a aplicação se encerrava ao pedir a sequencia de Fibonacci até 50 elemento, sentiu-se a necessidade de *checar* cada vez que se somava o número inteiro I_{n+1} com o I_n. para isso foi utilizando o método **checked_add**, que ao invés de apenas somar o valor, ele verifica se o novo valor é possível de ser representado como inteiro de 32bits, caso não seja possível é retornado a lista até onde foi possível gerar e uma mensagem de error, dizendo que houve um overflow.

Exibindo valores da Sequência de Fibonacci

Após a execução da thread, é recuperado a lista ligada por meio do **mutex**, foi feito um laço **for** pela lista ligada e exibido na tela o número de Fibonacci e seu respectivo valor. Por meio do **Macro println!**, em Rust **Macros** são funções que executadas em tempo de compilação, desde modo o parsing da formatação da mensagem e geração de tipos ocorre em tempo de compilação. Na prática a sua usabilidade se assemelha as função **println** do Dart, kotlin, Java.

Listing 6: Exibindo valores da Sequência de Fibonacci

```
  let mutex_fibonacci_numbers = mutex.lock().unwrap();

  println!("Sequencia: ");

  for (index, number) in mutex_fibonacci_numbers.iter().enumerate()
       {
      println!("F({}) - {}", index, number);
  }
```

Listing 7: main.rs

```
use clap::clap_app;
use tarefa_threads::fibonacci;

use std::thread;
use std::{
    collections::LinkedList,
    sync::{Arc, Mutex},
    usize,
};

fn main() {
    let matches = clap_app!(tarefa_threads =>
        (author: "Samuel Cavalcanti <scavalcanti111@gmail.com>")
        (about: "Tarefa Threads, o usuario digitara na linha de
            comandos a quantidade de numeros de Fibonacci que o
            programa deve gerar ")
        (@arg Number: +required "Digite a quantidade de numeros de
            fibonacci")
        (version: "1.0")
    ).get_matches();

    let input = matches
        .value_of("Number")
        .unwrap()
        .parse::<usize>()
        .expect("\nError nao foi possivel fazer o parse do input.
            Um numero positivo era o esperado.\n");

    println!("Por favor aguarde ...");

    let mutex = Arc::new(Mutex::new(LinkedList::new()));

    let mutex_thread_child = Arc::clone(&mutex);
    let handle = thread::spawn(move || {
        let mut mutex_fibonacci_numbers = mutex_thread_child.lock()
            .unwrap();
        *mutex_fibonacci_numbers = fibonacci(input);
    });
    handle.join().unwrap();

    let mutex_fibonacci_numbers = mutex.lock().unwrap();

    println!("Sequencia: ");

    for (index, number) in mutex_fibonacci_numbers.iter().enumerate
        () {
        println!("F({}) - {}", index, number);
    }
}
```

Referências

Klabnik, S. and Nichols, C. (2019). *The Rust Programming Language (Covers Rust 2018)*. No Starch Press.

wikipedia (2008). Fibonacci wikipedia. `https://en.wikipedia.org/wiki/Fibonacci_number`. [Online; accessed 2-march-2021].