Broadcast entre threads usando um buffer compartilhado (EM RUST)

Descrição dos testes realizados, problemas identificados e não resolvidos.

*por **Fernando Lima** (**2020877**)*

Nota: Todo o trabalho pode ser encontrado e reproduzido pelo repositório neste link.

...Usando a técnica de passagem de bastão e a modelagem proposta por Andrews, implemente em C e pthreads uma estrutura de dados que é um buffer limitado com N posições, usado para broadcast entre P produtores e C consumidores. Cada produtor deve depositar I ítens e depois terminar execução, e cada consumidor deve consumir P*I itens e terminar sua execução. (Os valores de N, P, C e I devem ser parâmetros de linha de comando para o programa de teste desenvolvido, nesta ordem.). Para simplificar, vamos supor que os itens de dados enviados são inteiros...

Este documento descreve testes de funcionalidade de cada da descrição enunciada acima, utilizando pthreads as funcionalidades e o padrão *Buffer* foi implementado na biblioteca *buffer.rs* que disponibiliza uma Api funcional que é consumida pelas threads de depósito e consumo nos programas principais de cada teste.

...Ao chamar deposita, o produtor deve ficar bloqueado até conseguir inserir o novo item, e ao chamar consome o consumidor deve ficar bloqueado até conseguir um item para consumir. Uma posição só pode ser reutilizada quando todos os C consumidores tiverem lido o dado. Cada consumidor deve receber as mensagens (dados) na ordem em que foram depositadas. Por favor usem o arquivo disponibilizado buffer.h para a interface das funções implementadas...

O resumo do comportamento do buffer pode ser descrito pelas regras de negócio descritas a seguir:

- Cada Buffer é inicializado com um número de Produtores e Consumidores fixos;
- O Buffer é construido através de uma fila (FIFO) de dados;
- Cada produtor observa uma fila (*nxt_free*) que disponibiliza a próxima posição disponível para escrita, garantindo a ordenação;
- Cada posição é ocupada por uma estrutura de dado com um contador (*bdata.falta_ler*) que sinaliza quantos consumidores faltam ler o dado na posição;
- Quando um produtor aloca um dado na fila o contador (*falta_ler*) é setado para o número de consumidores;
- Quando o contador chega a zero a posiçao é desalocada e inserida na fila de posições (*nxt_free*);
- Cada consumidor possui uma fila (*nxt_data[meu_id]*) preenchida com as posições em ordem de leitura;
- Quando um consumidor realiza uma leitura a posição sai da sua fila (*nxt_data[meu_id]*);
- Os produtores adicionam as posições a cada deposito em cada componente (*nxt_data[meu_id]*) para leitura;

Breve Nivelamento sobre Rust e Ownership

Rust consegue entregar programas paralelos com eficiência de uma forma segura utilizando conceitos de ***Ownership*** e ***Borrowing***. *Ownership* é o recurso mais exclusivo do Rust e permite que ele faça garantias de segurança de memória sem a necessidade de um coletor para "limpar o lixo". A memória é gerenciada por meio de um sistema de propriedade com um conjunto de regras que o compilador verifica no momento da

compilação. Nenhum dos recursos de propriedade torna seu programa lento durante a execução. Cada valor em Rust tem uma variável que é chamada de *owner*(proprietário) e só pode haver um proprietário por vez. Quando o proprietário sai do escopo, o valor é descartado.

Em linguagens com coletor de lixo (GC), o GC rastreia e limpa a memória que não está mais sendo usada, e não precisamos pensar sobre isso. Rust segue um caminho diferente: a memória é retornada automaticamente assim que a variável que a possui sai do escopo.

Rust também permite o conceito de *Borrowing*, no qual consiste em criar uma referência que se refere a um valor, mas não o possui. Por não ser o proprietário dele, o valor para o qual ele aponta não será descartado quando a referência parar de ser usada. Em Rust os dados são por padrão imutáveis, porém é possível declarar valores e referências *mutáveis* (com a cláusula *mut*). Entretanto não é permitido emprestar a ownership de uma variável mutável mais de uma vez. Essa restrição que impede múltiplas referências mutáveis aos mesmos dados ao mesmo tempo permite a mutação de uma forma muito controlada, diferente da maioria das linguagens, pois elas permitem que você mude sempre que quiser. A vantagem de ter essa restrição é que o Rust pode evitar disputas de dados em tempo de compilação.

Há casos em que um único valor pode ter vários proprietários. Por exemplo, em estruturas de dados de grafos, várias arestas podem apontar para o mesmo nó, e esse nó é conceitualmente de propriedade de todas as arestas que apontam para ele. Um nó não deve ser limpo, a menos que não tenha nenhuma aresta apontando para ele. Para se ter múltiplos proprietários, Rust possui uma estrutura chamada Rc, que é uma abreviatura para contagem de referência. O tipo Rc rastreia o número de referências a um valor para determinar se o valor ainda está em uso ou não. Se houver zero referências a um valor, o valor pode ser limpo sem que nenhuma referência se torne inválida.

Este trabalho irá explorar as vantagens descritas acima para acessar a estrutura do Buffer Compartilhado de forma segura e com eficiência.

Tarefas de Consumo e Produção

Na implementação dos *handlers* de produção e consumo, não há diferença de paradigma, somente diferenças de implementação das instruções para Rust, em relação a um código feito C. A programação é funcional, o *handler* de produção se encarrega da tarefa de inserir I inserções, enquanto o de consumo consome um determinado número de dados no buffer para um consumidor específico com identificador *my_id*. As funções abaixo descrevem a tarefa pela superfície, os conceitos envolvidos serão detalhados adiante.

Produção

```
fn deposit_handler(mutex: &mut Arc<Mutex<SBuffer<i32>>>, mut insertions: i32)
{
    let mut buff = mutex.lock().unwrap();
    while insertions > 0
    {
        thread::sleep(time::Duration::from_secs(rand::random::<u64>() % 3));
        buff.push(rand::random::<i32>());
        insertions -= 1;
    }
    println!("{}", buff);
}
```

Consumo

```
fn consome_handler(mutex: &mut Arc<Mutex<SBuffer<i32>>>, my_id: usize, mut
consumes: u32)
{
    let mut buff = mutex.lock().unwrap();
    let mut data: Vec<i32> = vec![];
    while consumes > 0
    {
        thread::sleep(time::Duration::from_secs(rand::random::<u64>() % 2));
        data.push(buff.pop(my_id).unwrap());
        consumes -=1;
    }
    print!("Data consumed by {}: ( ", my_id);
    for elem in data.iter() {
        print!("{}; ", elem);
    }
    println!(")");
}
```

Testes de Unidade

Rust e seu gerenciador de pacotes Cargo permitem a execução de testes unitários com simplicidade. Foram executados os mesmos testes da aplicação em C em forma de testes de unidade.

Cada seção abaixo descreverá os testes, o log a seguir mostra o resultado dos testes, atestando que todos passam pelas avaliações (ou *asserts*) propostos por cada teste.Para reporduzir os testes basta executar o comando ***cargo test*** na pasta raiz do projeto.

```
Finished test [unoptimized + debuginfo] target(s) in 6.74s
Running unittests (target/debug/deps/rust_broadcast_buffer-0ba1d236c5b63c12)

running 5 tests
test test::tests::test_create_buffer ... ok
test test::tests::test_consume ... ok
test test::tests::test_pop_from_buffer ... ok
test test::tests::test_push_to_buffer ... ok
test test::tests::test_deposit ... ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.01s
```

Testes de Inicialização (OK)

O teste de inicialização teste somente a inicialização e finização do buffer, dado pelo código a seguir. O buffer é inicializado com 10 posições disponíveis 2 produtores e 2 consumidores.

```
#[test]
/**
* Creates a i32 SBuffer with:
*  {capacity} positions;
*  1 producer slot;
*  1 consumer slot;
*
* For filling with BData.
*/
fn test_create_buffer() {
    let numprod = 2;
    let numcons = 2;
    let capacity = 10;
    let buffer: SBuffer<i32> = SBuffer::with_capacity(capacity, numprod, numcons);
    assert_eq!(buffer.numpos(), capacity);
    assert_eq!(buffer.numcons(), 2);
    assert_eq!(buffer.numprod(), 2);
}
```

Testes de Push (OK)

O teste de produção testa a criação do buffer e a inserção de três items:

```
#[test]
/** Push to Buffer
    *
*/
fn test_push_to_buffer() {
    let numprod = 2;
    let numcons = 2;
    let capacity = 10;
    let mut buffer: SBuffer<i32> = SBuffer::with_capacity(capacity, numprod,
numcons);
    buffer.push(100);
    buffer.push(200);
    buffer.push(300);
    assert_eq!(buffer.data(0).unwrap(), 100);
    assert_eq!(buffer.data(1).unwrap(), 200);
    assert_eq!(buffer.data(2).unwrap(), 300);
}
```

Testes de Pop (OK)

Neste teste o Buffer se inicializa pré-preenchido e o teste é realizado retirnado items do buffer:

```
#[test]
/** Pop from Buffer
```

```
 *
*/
fn test_pop_from_buffer() {
    let numprod = 4;
    let numcons = 2;
    let insertions = 2;
    let capacity = numprod * insertions;
    let mut buffer: SBuffer<i32> = SBuffer::with_capacity(capacity, numprod,
numcons);

    for i in 0..capacity
    {
        buffer.push(((i+1) * 100) as i32);
    }

    let mut data;
    for i in 0..capacity
    {
        for c in 0..numcons
        {
            data = buffer.pop(c);
            assert_eq!(data.unwrap(), ((i+1) * 100) as i32);
        }
    }
    assert_eq!(buffer.is_empty(), true);
}
```

Testes de Produção (OK)

A forma de invocar threads em Rust possui diferenças significativas a forma como C implementa. Com o Pthreads, é necessário criar a estrutura da thread e utilizar a API para criar e realizar o join das threads, passando a referência da estrutura criada, com a função que será executada e os argumentos passados a função, como mostrado abaixo.

```
pthread_create(&prod_thd, NULL, deposita_thread, &d_arg[i]);
...
pthread_join(prod_thd, NULL);
```

Em Rust a bibliotéca thread cria as threads através de *closures*, funções anônimas que são passadas como parâmetro a função *spawn*, servindo como handlers que são executados pelas threads e podem retornar resultados a partir do join. Os sistemas de Ownership é uma forma poderosa de ajudar a gerenciar a segurança da memória e problemas de simultaneidade. Aproveitando o Ownership e a verificação de tipo, muitos erros de concorência são erros de tempo de compilação no Rust, em vez de erros de tempo de execução.

O a diretiva *move* é freqüentemente usada com closures junto com *thread ::spawn* porque permite que você use dados de um thread em outro thread. Rust não pode dizer por quanto tempo o thread gerado será executado, então ele não sabe se a referência de uma variável passada de outra thread sempre será válida. Essa diretiva passa a Ownership de uma variável de uma thread para a outra.

A biblioteca padrão Rust fornece canais para passagem de mensagens e tipos de ponteiro inteligente, como Mutex e Arc, que são seguros para uso em contextos simultâneos. O sistema de tipo e o verificador de empréstimo garantem que o código que usa essas soluções não termine com disputas de dados ou referências inválidas. A estrutura Mutex é usada para garantir a exclusão mútua. Para acessar os dados dentro do mutex, usamos o método lock para adquirir o lock. Esta chamada irá bloquear o thread atual para que ele não possa fazer nenhum trabalho até que seja nossa vez de bloqueá-lo.

Em tempo de compilação, Rust não permite mover a variável Mutex para várias threads, para isso o tipo Arc fornece propriedade compartilhada de um valor de tipo T, alocado no heap. Invocar clone no Arc produz uma nova instância Arc, que aponta para a mesma alocação no heap que o Arc de origem, enquanto aumenta uma contagem de referência de uma forma atômica, sendo assim segura para threads.

```
#[test]
fn test_deposit()
{
    let data = rand::random::<i32>();
    let shared_buffer: Arc<Mutex<SBuffer<i32>>> =
Arc::new(Mutex::new(SBuffer::with_capacity(10, 1, 1)));
    let my_buffer = Arc::clone(&shared_buffer);
    let handle = thread::spawn(move || {
        let mut buff = my_buffer.lock().unwrap();
        buff.push(data);
    });
    handle.join().unwrap();
    assert_eq!(shared_buffer.lock().unwrap().data(0).unwrap(), data);
}
```

O código acima acima descreve a utilização das estruturas Mutex e Arc para clonar e envolver a estrutura do buffer e passa-la por threads. Como o Mutex guarda a referência do buffer é possível depositar dados via thread após adquirir o lock. No final do teste é checado se o dado inserido via thread está correto.

Testes de Consumo (OK)

Seguindo o mesmo princípio de invocação de threads. O teste de consumo abre threads para consumir dados de um buffer pre-preenchido e validada o consumo desses dados após o join.

```
#[test]
fn test_consume()
{
    let shared_buffer: Arc<Mutex<SBuffer<i32>>> =
Arc::new(Mutex::new(SBuffer::with_capacity(10, 1, 1)));
    shared_buffer.lock().unwrap().push(100);
    shared_buffer.lock().unwrap().push(200);

    let my_buffer = Arc::clone(&shared_buffer);
    let handle = thread::spawn(move || -> Option<i32> {
        let mut buff = my_buffer.lock().unwrap();
        buff.pop(0)
    });
```

```
    let res = handle.join().unwrap();
    assert_eq!(res.unwrap(), 100);
}
```

Tarefa Geral (OK)

O teste geral, reproduzido no crate binário ***main.rs***, une todas as características descritas nos testes acima e as testa, abrindo simultaneamente várias threads de deposito e consumo, por fim finalizando o buffer. O resultado é descrito pelo bash:

```
Finished dev [unoptimized + debuginfo] target(s) in 0.47s
Running `target/debug/rust-broadcast-buffer`

Buffer[ 2058827439(3); 1637138227(3); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);
0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);  ] free_slots: 14 next_free: 2
Buffer[ 2058827439(3); 1637138227(3); 1091656967(3); 409624372(3); 0(0); 0(0);
0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);  ] free_slots: 12
next_free: 4
Buffer[ 2058827439(3); 1637138227(3); 1091656967(3); 409624372(3); 856994756(3);
1649947986(3); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);  ]
free_slots: 10 next_free: 6
Buffer[ 2058827439(3); 1637138227(3); 1091656967(3); 409624372(3); 856994756(3);
1649947986(3); 1716518158(3); 1460852491(3); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);
0(0); 0(0);  ] free_slots: 8 next_free: 8

Data consumed by 0: ( 2058827439; 1637138227; )
Data consumed by 2: ( 2058827439; 1637138227; 1091656967; 409624372; )
Data consumed by 1: ( 2058827439; 1637138227; 1091656967; )

Buffer[ 0(-1); 0(-1); 1091656967(1); 409624372(2); 856994756(3); 1649947986(3);
1716518158(3); 1460852491(3); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);  ]
free_slots: 10 next_free: 8
```

Na inicialização do programa são invocadas threads de produção para inserir aleatórios dados no buffer de forma paralela. Cada produtor P insere I itena no buffer. Como mostrado pela primeira parte do log:

```
...
Buffer[ 2058827439(3); 1637138227(3); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);
0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);  ] free_slots: 14 next_free: 2
Buffer[ 2058827439(3); 1637138227(3); 1091656967(3); 409624372(3); 0(0); 0(0);
0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);  ] free_slots: 12
next_free: 4
Buffer[ 2058827439(3); 1637138227(3); 1091656967(3); 409624372(3); 856994756(3);
1649947986(3); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);  ]
free_slots: 10 next_free: 6
Buffer[ 2058827439(3); 1637138227(3); 1091656967(3); 409624372(3); 856994756(3);
1649947986(3); 1716518158(3); 1460852491(3); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);
0(0); 0(0);  ] free_slots: 8 next_free: 8
...
```

No caso, o buffer tem capacidade de 16 posições e 4 produtores inserem 2 items no buffer deixando 8 posições livres. Pela premissa do broadcast, cada posição ocupada tem um contador com o número de consumidores, quando cada consumidor é executado, o contador da posição decrementa até que não haja mais consumidores e a posição é liberada para uma nova inserção. No código, 3 consumidores são acionados em 3 threads para paralelizar o consumo de dados. A segunda parte do log abaixo mostra o quanto cada consumidor consumui do buffer:

```
...
Data consumed by 0: ( 2058827439; 1637138227; )
Data consumed by 2: ( 2058827439; 1637138227; 1091656967; 409624372; )
Data consumed by 1: ( 2058827439; 1637138227; 1091656967; )
...
```

O consumidor com id 0 consumiu dois dados, assim como 1 e 2 consumiram 3 e 4, respectivamente, em sequência, já que o buffer é uma fila. Isso significa que os dois primeiros dados foram consumidos por todos os consumidores em suas threads. A última parte do log, mostra exatamente a premissa do broadcast no buffer. Os dois primeiros dados, nos quais foram consumidos por todos os consumidores, são retirados do buffer com o contador chegando a zero e suas posições ficam livres, enquanto nas outras posições consumidas o contador decrementa mas não chegam a zero, então dos dados nas posições 3 e 4 são mantidos.

```
...
Buffer[ 0(-1); 0(-1); 1091656967(1); 409624372(2); 856994756(3); 1649947986(3);
1716518158(3); 1460852491(3); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0); 0(0);  ]
free_slots: 10 next_free: 8
```

Com esses resultados, é possível demostrar o funcionamento do buffer em broadcast para casos em uma ou várias threads atuando como produtores e consumidores. A implementação em Rust mostra-se bastante similar em termos de dificuldade com a implementação em C, entretanto as premissas intrínsicas da linguagem Rust permitem, em tempo de compilação, garantir a segurança no gerenciamento da memória *heap* e também o síncronismo para evitar a corrida pelo acesso aos dados em rotinas paralelas.