Broadcast entre threads usando um buffer compartilhado

Descrição dos testes realizados, problemas identificados e não resolvidos.

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

...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.1h* 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 possui um contador (*falta_ler[pos]*) 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;

Testes de Inicialização (OK)

O teste de inicialização teste somente a inicialização e finização do buffer, dado pelo trecho:

```
tbuffer* buffer = iniciabuffer(N, P, C);
...
finalizabuffer(buffer);
```

Para reproduzir o teste utilize o comando:

```
make inicializa.test PARAMS="16 2 2 4"
```

O resultado esperado é descrito pelo log a seguir, o buffer é inicializado com 16 posições disponíveis 2 produtores e 2 consumidores, o programa também testará a inserção de 4 items, não contemplada nesse teste

```
./bin/test/inicializa.test 16 2 2 4
Parametros N, P, C, I = 16 2 2 4
Buffer [ * * * * * * * * * * * * * * * *  ] ( free slots: 16 next free: 0 )
```

Testes de Produção (OK)

O teste de produção testa a criação de uma thread para cada produtor que inicializa a função de depósito em um semáforo através da API do buffer, dado pelo trecho:

```
pthread_t* prod_thds = (pthread_t*)malloc(sizeof(pthread_t) * P);
...
//Deposita
deposita_args* d_arg = (deposita_args *) malloc(sizeof(deposita_args) * P);
for (int i = 0; i < P; i++)
{
    //Produz
    d_arg[i].buffer = buffer;
    d_arg[i].item = rand() % 100;
    d_arg[i].insertions = I;
    pthread_create(&(prod_thds[i]), NULL, deposita_thread, &d_arg[i]);
}

for (int i = 0; i < P; i++) {pthread_join(prod_thds[i], NULL);}
```

Para reproduzir o teste utilize o comando:

```
make produz.test PARAMS="16 2 2 4"
```

O resultado esperado é descrito pelo log a seguir, o buffer é inicializado com 16 posições disponíveis 2 produtores e 2 consumidores, o programa também testará a inserção de 4 items aleatórios pelos 2 produtores, totalizando 8 items nas primeiras posições.

```
./bin/test/produz.test 16 2 2 4
Parametros N, P, C, I = 16 2 2 4
```

```
5372::7998 -> Buffer[ 7998 * * * * * * * * * * * * * * * *  ] ( free slots: 15 next
free: 1 )
5372::7396 -> Buffer[ 7998 7396 * * * * * * * * * * * * * *  ] ( free slots: 14
next free: 2 )
5371::4067 -> Buffer[ 7998 7396 4067 * * * * * * * * * * * *  ] ( free slots: 13
next free: 3 )
5372::5332 -> Buffer[ 7998 7396 4067 5332 * * * * * * * * * * *  ] ( free slots:
12 next free: 4 )
5371::7470 -> Buffer[ 7998 7396 4067 5332 7470 * * * * * * * * * *  ] ( free
slots: 11 next free: 5 )
5372::5418 -> Buffer[ 7998 7396 4067 5332 7470 5418 * * * * * * * * *  ] ( free
slots: 10 next free: 6 )
5371::2158 -> Buffer[ 7998 7396 4067 5332 7470 5418 2158 * * * * * * * *  ] (
free slots: 9 next free: 7 )
5371::2158 -> Buffer[ 7998 7396 4067 5332 7470 5418 2158 2158 * * * * * * *  ] (
free slots: 8 next free: 8 )
```

Cada inserção é caracterizada pelo log:

```
{thread_id}::{item} -> Buffer[ {item} * * * * * * * * * * * * * * * *  ...] ( free
slots: N - 1 next free: pos )
```

Testes de Consumo (OK)

Neste teste o Buffer se inicializa pré-preenchido com 5 items para teste isolado de consumo:

```
pthread_t* cons_thds = (pthread_t*)malloc(sizeof(pthread_t) * C);
int start[] = {100, 200, 300 , 400 , 500, 600, 700};
tbuffer* buffer = iniciabuffer_pre(N, P, C, NELEMS(start), start);
```

O teste de consumo testa a criação de uma thread para cada consumidor C que inicializa a função de consumo em um semáforo através da API do buffer, dado pelo trecho:

```
consome_args* c_arg = (consome_args *) malloc(sizeof(consome_args) * C);
for (int i = 0; i < C; i++)
{
    //Consome
    c_arg[i].buffer = buffer;
    c_arg[i].id = i;
    c_arg[i].consome = P * I;
    pthread_create(&(cons_thds[i]), NULL, consome_thread, &c_arg[i]);
}

for (int i = 0; i < C; i++) {pthread_join(cons_thds[i], NULL);}
```

Para reproduzir o teste utilize o comando:

```
make consome.test PARAMS="16 1 2 4"
```

O resultado esperado é descrito pelo log a seguir, o buffer é inicializado com 16 posições disponíveis 2 produtores e 2 consumidores, o programa também testará a inserção de 4 items aleatórios pelos 2 produtores, totalizando 8 items nas primeiras posições. Cada consumidor lê P * I items e armazena totalizando 10 leituras e retirando P * I posições do buffer.

```
./bin/test/consome.test 16 1 2 4
Parametros N, P, C, I = 16 1 2 4
Consumer 1 data: [ 100 ] ( free slots: 9 next data: 1 )
5880::100 <- Buffer[ 100[1] 200[2] 300[2] 400[2] 500[2] 600[2] 700[2] *[-1] *[-1]
*[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 9 next free: 7 )
Consumer 0 data: [ 100 ] ( free slots: 10 next data: 1 )
5879::100 <- Buffer[ *[-1] 200[2] 300[2] 400[2] 500[2] 600[2] 700[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 10 next free: 7 )
Consumer 1 data: [ 100 200 ] ( free slots: 10 next data: 2 )
5880::200 <- Buffer[ *[-1] 200[1] 300[2] 400[2] 500[2] 600[2] 700[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 10 next free: 7 )
Consumer 0 data: [ 100 200 ] ( free slots: 11 next data: 2 )
5879::200 <- Buffer[ *[-1] *[-1] 300[2] 400[2] 500[2] 600[2] 700[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 11 next free: 7 )
Consumer 1 data: [ 100 200 300 ] ( free slots: 11 next data: 3 )
5880::300 <- Buffer[ *[-1] *[-1] 300[1] 400[2] 500[2] 600[2] 700[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 11 next free: 7 )
Consumer 1 data: [ 100 200 300 400 ] ( free slots: 11 next data: 4 )
5880::400 <- Buffer[ *[-1] *[-1] 300[1] 400[1] 500[2] 600[2] 700[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 11 next free: 7 )
Consumer 0 data: [ 100 200 300 ] ( free slots: 12 next data: 3 )
5879::300 <- Buffer[ *[-1] *[-1] *[-1] 400[1] 500[2] 600[2] 700[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 12 next free: 7 )
Consumer 0 data: [ 100 200 300 400 ] ( free slots: 13 next data: 4 )
5879::400 <- Buffer[ *[-1] *[-1] *[-1] *[-1] 500[2] 600[2] 700[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 13 next free: 7 )
```

Cada consumo é caracterizada pelo log:

```
Consumer {meu_id} data: [ item ... ] ( free slots: 9 next data: 1 )
{thread_id}::{item} <- Buffer[ {item[0]}[{falta_ler[0]}] {item[1]}[{falta_ler[1]}]
*[-1] *[-1] ] ( free slots: N - 2 next free: pos )
```

Onde * caracteriza um slot disponível e *[-1]* siginifica que o contador *falta_ler* não foi inicializado.

Note que cada Consumidor termina com P * I items armazenados

```
...
Consumer 1 data: [ 100 200 300 400 ] ( free slots: 11 next data: 4 )
```

```
...
Consumer 0 data: [ 100 200 300 400 ] ( free slots: 13 next data: 4 )
...
```

Teste Geral (OK)

O teste geral 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 log:

```
./bin/test/completo.test 16 2 2 4
Parametros N, P, C, I = 16 2 2 4
6568::7998 -> Buffer[ 7998 * * * * * * * * * * * * * * * *  ] ( free slots: 15 next
free: 1 )
6568::7396 -> Buffer[ 7998 7396 * * * * * * * * * * * * * *  ] ( free slots: 14
next free: 2 )
6567::4067 -> Buffer[ 7998 7396 4067 * * * * * * * * * * * *  ] ( free slots: 13
next free: 3 )
6568::5332 -> Buffer[ 7998 7396 4067 5332 * * * * * * * * * *  ] ( free slots:
12 next free: 4 )
6567::7470 -> Buffer[ 7998 7396 4067 5332 7470 * * * * * * * * *  ] ( free
slots: 11 next free: 5 )
6568::5418 -> Buffer[ 7998 7396 4067 5332 7470 5418 * * * * * * * *  ] ( free
slots: 10 next free: 6 )
Consumer 1 data: [ 7998 ] ( free slots: 10 next data: 1 )
6588::7998 <- Buffer[ 7998[1] 7396[2] 4067[2] 5332[2] 7470[2] 5418[2] *[-1] *[-1]
*[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 10 next free: 6 )
Consumer 0 data: [ 7998 ] ( free slots: 11 next data: 1 )
6587::7998 <- Buffer[ *[-1] 7396[2] 4067[2] 5332[2] 7470[2] 5418[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 11 next free: 6 )
Consumer 1 data: [ 7998 7396 ] ( free slots: 11 next data: 2 )
6588::7396 <- Buffer[ *[-1] 7396[1] 4067[2] 5332[2] 7470[2] 5418[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 11 next free: 6 )
Consumer 0 data: [ 7998 7396 ] ( free slots: 12 next data: 2 )
6587::7396 <- Buffer[ *[-1] *[-1] 4067[2] 5332[2] 7470[2] 5418[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 12 next free: 6 )
Consumer 1 data: [ 7998 7396 4067 ] ( free slots: 12 next data: 3 )
6588::4067 <- Buffer[ *[-1] *[-1] 4067[1] 5332[2] 7470[2] 5418[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 12 next free: 6 )
Consumer 1 data: [ 7998 7396 4067 5332 ] ( free slots: 12 next data: 4 )
6588::5332 <- Buffer[ *[-1] *[-1] 4067[1] 5332[1] 7470[2] 5418[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 12 next free: 6 )
Consumer 0 data: [ 7998 7396 4067 ] ( free slots: 13 next data: 3 )
6587::4067 <- Buffer[ *[-1] *[-1] *[-1] 5332[1] 7470[2] 5418[2] *[-1] *[-1] *[-1]
*[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 13 next free: 6 )
Consumer 1 data: [ 7998 7396 4067 5332 7470 ] ( free slots: 13 next data: 5 )
6588::7470 <- Buffer[ *[-1] *[-1] *[-1] 5332[1] 7470[1] 5418[2] *[-1] *[-1] *[-1]
*[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 13 next free: 6 )
Consumer 1 data: [ 7998 7396 4067 5332 7470 5418 ] ( free slots: 13 next data: -1
)
6588::5418 <- Buffer[ *[-1] *[-1] *[-1] 5332[1] 7470[1] 5418[1] *[-1] *[-1] *[-1]
*[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 13 next free: 6 )
```

```
Consumer 1 data: [ 7998 7396 4067 5332 7470 5418 ] ( free slots: 13 next data: -1
)
6588::-1 <- Buffer[ *[-1] *[-1] *[-1] 5332[1] 7470[1] 5418[1] *[-1] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 13 next free: 6 )
Consumer 1 data: [ 7998 7396 4067 5332 7470 5418 ] ( free slots: 13 next data: -1
)
6588::-1 <- Buffer[ *[-1] *[-1] *[-1] 5332[1] 7470[1] 5418[1] *[-1] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 13 next free: 6 )
6567::1909 -> Buffer[ * * * 5332 7470 5418 1909 * * * * * * * * *  ] ( free slots:
12 next free: 7 )
Consumer 0 data: [ 7998 7396 4067 5332 ] ( free slots: 13 next data: 4 )
6587::5332 <- Buffer[ *[-1] *[-1] *[-1] *[-1] 7470[1] 5418[1] 1909[2] *[-1] *[-1]
*[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 13 next free: 7 )
Consumer 0 data: [ 7998 7396 4067 5332 7470 ] ( free slots: 14 next data: 5 )
6587::7470 <- Buffer[ *[-1] *[-1] *[-1] *[-1] *[-1] 5418[1] 1909[2] *[-1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 14 next free: 7 )
6567::166 -> Buffer[ * * * * * 5418 1909 166 * * * * * * * * *  ] ( free slots: 13
next free: 8 )
Consumer 0 data: [ 7998 7396 4067 5332 7470 5418 ] ( free slots: 14 next data: 6 )
6587::5418 <- Buffer[ *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] 1909[2] 166[2] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 14 next free: 8 )
Consumer 0 data: [ 7998 7396 4067 5332 7470 5418 1909 ] ( free slots: 14 next
data: 7 )
6587::1909 <- Buffer[ *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] 1909[1] 166[2] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 14 next free: 8 )
Consumer 0 data: [ 7998 7396 4067 5332 7470 5418 1909 166 ] ( free slots: 14 next
data: -1 )
6587::166 <- Buffer[ *[-1] *[-1] *[-1] *[-1] *[-1] *[-1] 1909[1] 166[1] *[-1] *
[-1] *[-1] *[-1] *[-1] *[-1] *[-1] *[-1]  ] ( free slots: 14 next free: 8 )
```

Desta da vez o consumidor 1 terminou com *6* items armazenados. Essa peculiaridade acontenceu pois como as threads de consumo e produção são inicializadas com uma pseudo-aleatoriedade, os produtores inseriram somente 6 items no buffer, tendo somente estes disponíveis para consumo pelos dois consumidores.

```
...
Consumer 1 data: [ 7998 7396 4067 5332 7470 5418 ] ( free slots: 13 next data: -1
)
...
```

Algum tempo depois, mais dois items foram adicionados ao buffer, possibilitando a leitura de um dos consumidores para os P * I items pré-estabelecidos.

```
...
6567::1909 -> Buffer[ * * * 5332 7470 5418 1909 * * * * * * * * *  ] ( free slots:
12 next free: 7 )
...
6567::166 -> Buffer[ * * * * * 5418 1909 166 * * * * * * * * *  ] ( free slots: 13
next free: 8 )
...
```

Com isso o consumidor 0 termina sua leitura com P * I items armazenados.

```
...
Consumer 0 data: [ 7998 7396 4067 5332 7470 5418 1909 166 ] ( free slots: 14 next
data: -1 )
...
```