Sistemas Operacionais
Produtor/Consumidor

Prof. Dr. Gerson Cavalheiro [1],
Guilherme de Souza[1]

[1]

1. Execução

Para este trabalho foi fornecido um *Makefile* junto ao código, para facilitar a sua execução. Com isso basta executar o comando:

make

Logo após para executar o programa deve-se utilizar o seguinte comando seguido dos parâmetros esperado por ele:

./producer_consumer <num_it><num_prod><num_cons><tam_buffer>

- **num_it**: Número de iterações passado para os produtores.
- **num_prod**: Número de produtores.
- **num_cons**: Número de consumidores.
- **tam_buffer**: Tamanho maximo do buffer.

2. Relatório

Para implementação deste trabalho, parti da leitura da teoria/funcionabilidade do produtor/consumidor. O que não foi uma tarefa complicada, fazer-se entender foi o passo mais simples. Em seguida para que a implementação do mesmo fosse possivel, o conhecimento de pthreds se fez necessario (como pedido pelo trabalho, dado que não poderia fazer uso do threads do C++ 11).

Em um primeiro momento realizei a implementação de um pseudo código com o que eu realmente gostaria de programar.

Data: numero_iteracoes, tamanho_buffer
while $\Delta i < numero\_iteracoes$ **do**
  if *buffer == numero_iteracoes* **then**
    espera até mudar;
  end
  secao_critica;
  buffer.push(rand());
  fim_secao_critica;
end

Algorithm 1: Pseudo código para o produtor

```
while true do
    if buffer == vazio then
        espera até mudar;
    end
    secao_critica;
    if buffer.front() == -1 then
        buffer.pop();
        break;
    end
    e_primo(buffer.front());
    if e_primo then
        imprime(thread_id : numero_tirado_do_buffer)
    end
    buffer.pop();
    fim_secao_critica;
end
```

Algorithm 2: Pseudo código para o consumidor

Partindo do pseudo código pude ver alguns erros como: lugar onde seria necessario a seção critica e como a lista devia se comportar para que ela seguisse o padrão FIFO. Para solucionar a parte da lista, bastou fazer uso correto das funções ja dadas pelo *list* do C++ 11, inserindo atrás e removendo na frente.

Quanto a seção critica, para um melhor aproveitamento, fez-se uso de variaveis locais, que são visiveis somente ao escopo da função. Dessa forma pude paralelizar a parte de verificar se um numero é primo ou não e a parte de imprimir os que são dados como primos.

Para realizar o que era necessario algumas funções espesíficas foram necesárias:

- **pthread_cond_await()**: dessa forma conseguia esperar o sinal de que havia algo no buffer, ou que tinha espaço para inserir mais itens. Dessa forma colocava a thread em espera de forma correta, conseguindo fazer a liberação do mutex, assim não deixando ninguem presso a espera dele.
- **pthred_cond_signal()**: para que as threads em espera pudessem sair do seu estado de *sleep*, toda vez que era terminado de inserir e remover um item do buffer, um sinal era mandado. Dessa forma tenho um controle maior sobre o que esta acontecendo com o buffer.
- **pthread_mutex_lock()** e **pthread_mutex_unlock()**: assim conseguia criar a seção critica tanto para ver se estava cheio ou vazio o buffer, como para inserir ou remover o item do mesmo. Assim evitando os problemas comum do produtor/consumidor, que é inserir tendo um item no local, ou consumir nem um item, ou até duas ou mais *threads* consumirem o mesmo.

Enquanto o código estava em contrução pude observar alguns pontos, como o uso correto das funções da biblioteca e observar os pontos onde realmente é necessário uma seção critica ou não. parece meio obvio falando assim, mas uma programação mais no *"hardcoding"* mostra que o desempenho melhora e muito.

Porém dado o número de iterações pequeno para um determinado numero de

threads, a eficiência não é demonstrada. Dado o custo de criação de uma *thread* e sua execução. Agora quando o numero de iterações é grande suficiênte, é possivel ver ganhos reais.

3. hardware usado

Para obtenção das infromações sobre o *hardware* usado para algumas execuções, usou-se o comando **lscpu**, como pode ser visto abaixo:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
Address sizes:         36 bits physical, 48 bits virtual
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    2
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz
Stepping:              9
CPU MHz:               1197.298
CPU max MHz:           3200.0000
CPU min MHz:           1200.0000
BogoMIPS:              5188.11
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              3072K
NUMA node0 CPU(s):     0-3
```

Listing 1. informações CPU

4. Visualização usando HTOP

De forma que fosse possivel visualizar as *threads* criadas e, se de fato estava em funcionamento (partindo é claro do que o *hardware* nos possibilita), algumas execuções foram realizadas usando o **htop** como ferramenta de visualização.

Em um primeiro momento executou-se número pequenos de iterações e threads, porém a visualização nem era percebivel de fato (algo sobre já foi comentado em seções anteriores). Com isso partimos de execuções com números grandes. Primeiro podemos visualizar o uso da máquina na figura 1 antes da execução do código.

Com isso executou-se o código com os seguintes parâmetros demonstrados na tabela 1. Usou-se um tamanho de buffer pequeno, para desta forma fosse possivel ver *threads* paradas, com isso a visualização no htop se torna melhor. Mas denovo dado o limite da máquina os valores não crescem tanto, pela administração do próprio sistema perante as *threads* exigidas ou umfato não averiguado no momento. Mas é possivel ver

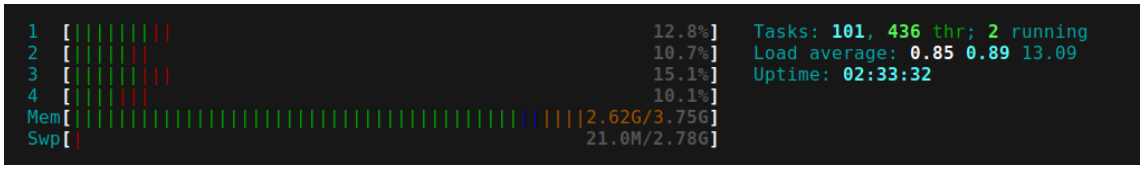

Figure 1. Visualização htop antes da execução do código

que a máquina encontra-se com 100% de uso, como demonstra a figura 2 e o numero de *threads* cresce também.

Parâmetros	Valores
Número de iterações	10000
Número de Produtores	500
Número de Consumidores	500
Tamanho do buffer	100

Table 1. Tabela com parâmetros passados para primeira execução do código

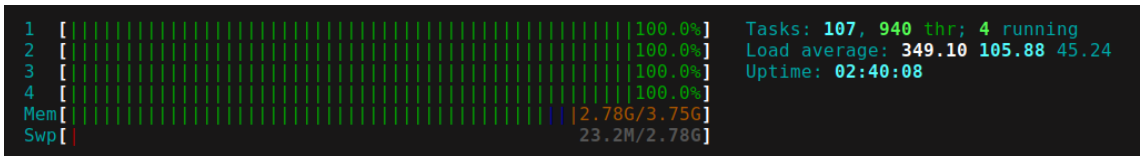

Figure 2. Visualização via htop durante execução do código

Com isso, aumentou-se o numero de *threads* em uso, para ver se esse "up" seria visivel via htop. Segui os mesmos passo anterior, podendo obervar os parâmetros na tabela 2 e o número de *threads* antes e depois da execução do código na figura 3 e 4 respectivamente.

Parâmetros	Valores
Número de iterações	10000
Número de Produtores	1000
Número de Consumidores	1000
Tamanho do buffer	100

Table 2. Tabela com parâmetros passados para segunda execução do código

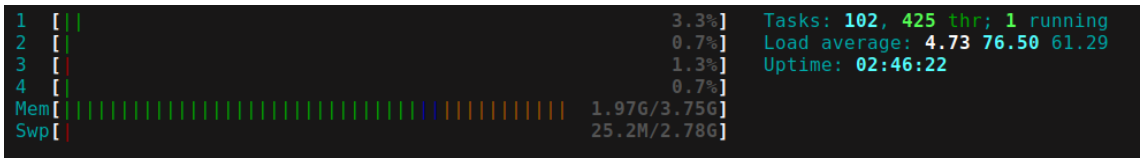

Figure 3. Visualização via htop antes da execução do código

Através das figuras, tornou-se possivel observar que o número de *threads* se manteve quase sempre pela metade do que era pedido, possivelmente dado o *hardware* em

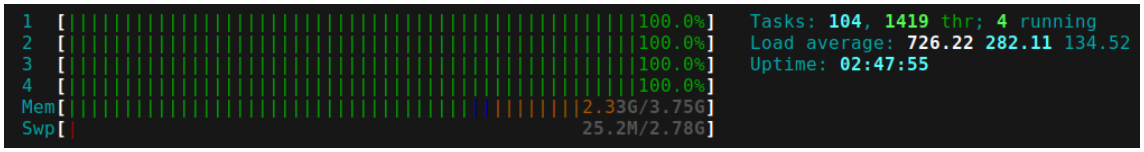

Figure 4. Visualização via htop durante execução do código

uso e, o próprio controle do sistema como ja comentado. Revisão no código foram feitas e comparações com de colegas para encontrar possíveis erros, porém, nada foi visivel, logo fica meu questionamento.