Sistemas Distribuídos – Trabalho 2

A linguagem utilizada para a implementação de programas *multithreading* foi a C++.

**Primeiro programa –**

**Segundo programa – producerCunsumer.cpp**

O segundo programa propôs a implementação *multithraeded* de produtores e consumidores, onde o primeiro escreveria o produto numa memória compartilhada limitada e o segundo faria a leitura. A sincronização e coordenação do acesso à memória compartilhada deveria ser feito por meio de semáforos.

As primeiras funções implementadas são auxiliares à atividade principal do programa, sendo utilizadas pela função *main* e suas threads:

* Int **isPrime**(int number): Tenta dividir um número especificado em seu argumento por todos os números do intervalo 2 até o número que antecede ele mesmo. Se alguma divisão proceder, o número não é primo e a função retorna o valor 0. Será primo caso contrário, retornando o valor 1.
* int **getRandomNumber**(int min, int max): Faz uso das funções *srand* e *rand* da biblioteca *stdlib* para obter um número aleatório. Para que o número aleatório esteja entre a faixa especificada nos argumentos da função, ele é dividido pelo valor da faixa e depois o módulo da divisão é somado ao valor mínimo, formando o número a ser retornado.
* int **getFirstFreePosition**(vector<int> memory): Itera todas as posições de um vetor até encontrar a primeira livre, aqui representada pelo valor 0. Retorna o número de tal posição e -1 caso não haja nenhuma posição livre.
* int **getFirstFullPosition**(vector<int> memory): Itera todas as posições de um vetor até encontrar a primeira cheia, aqui representada por um valor diferente de 0. Retorna o número de tal posição e -1 caso não haja nenhuma posição cheia.
* bool **isTotallyFree**(vector<int> memory): Faz uso da função getFirstFullPosition e retorna se todas as posições do vetor estão livres ou não.
* bool **isTotallyFull**(vector<int> memory): Faz uso da função getFirstFreePosition e retorna se todas as posições do vetor estão cheias ou não.

A função ***main*** do programa recebe como parâmetros Np e Nc, que indicam o número de threads do produtor e do consumidor, respectivamente. O número total de threads é a soma desses dois. Fazendo uso da biblioteca *thread*, primeiramente foi criado um container para todas as *threads*:

thread allThreads[totalNumberOfThreads];

E depois, para cada posição do container, foi definida qual função implementaria o comportamento das *threads* (**consumer** ou **producer**) através da função *thread*:

for (int i = 0; i < Np; ++i)

{

allThreads[i] = thread(producer);

}

for (int i = Np; i < Np+Nc; ++i)

{

allThreads[i] = thread(consumer);

}

Ainda usando a biblioteca *thread,* todas as *threads* foram incorporadas à função *main,* através da função *join*:

for (int i = 0; i < Np+Nc; ++i)

{

allThreads[i].join();

}

A função que implementa a *thread* produtora começa com um *while* que limita a produção de produtos para 10000. O produto é produzido fazendo uma chamada a função auxiliar getRandomNumber fora da região crítica, pois nenhuma variável é compartilhada com outras *threads* e, portanto, não há problemas caso ocorra troca de contexto. O semáforo foi implementado fazendo uso das bibliotecas *mutex* e *condition\_variable*:

unique\_lock<mutex> lock(semaphoreMutex);

if(semaphoreEmpty.wait\_for(lock, chrono::milliseconds(200), [] {return !isTotallyFull(sharedMemory);})){

//Escreve produto na memória compartilhada de tamanho N

semaphoreFull.notify\_all();

}

Quando a *thread* produtora em execução lê na função *wait\_for* ela checa se o vetor tem posições livres, fazendo uso da função auxiliar isTotallyFull. Se ele não tiver, a thread é bloqueada e colocada na fila de *threads* em espera. Ela é colocada em estado *ready* para continuar sua execução caso o semáforo contador *empty* seja incrementado/notificado pela função *notify\_all* ou então quando passar o tempo definido como argumento da função *wait\_for*. Quando a thread voltar a ser executada e a condição for satisfeita, o semáforo mutex será trancado, garantindo que a região crítica não será interrompida, e destrancado depois que o produto for escrito na memória compartilhada. Por fim, os semáforos *full* são incrementados/notificados.

A função que implementa a *thread* consumidora funciona de forma semelhante, com a diferença que usa o semáforo *full* (usando a função auxiliar isTotallyEmpty como condicional) e incrementa/notifica o semáforo *empty*:

unique\_lock<mutex> lock(semaphoreMutex);

if(semaphoreFull.wait\_for(lock, chrono::milliseconds(200), [] {return !isTotallyFree(sharedMemory);})){

//Lê produto da memória compartilhada, libera a posição e escreve produto na memória local

semaphoreEmpty.notify\_all();

}

int isPrimeNumber = isPrime(consumerProduct);

A verificação se o número é primo ou não acontece fora da região crítica, fazendo uso da função auxiliar isPrime.

Na medição do tempo para diferentes números de threads consumidoras (Nc) e produtoras (Np) e também diferentes tamanhos de memória compartilhada (N), foram tomados os seguintes cuidados: Foi utilizada a biblioteca *chrono* cujas funções permitiram a obtenção do “tempo de relógio” entre o início e fim da execução de todas as threads; Não foram incluídas na medição do tempo a criação de threads, vetores, ou quaisquer outras estruturas de dados, restringindo a medição de tempo a atividade principal do programa. Os resultados podem ser observados no gráfico a seguir:

De maneira geral, em todas as variações de memória, pode ser observado que um pequeno aumento do número de *threads* produtoras ou consumidoras provocou diminuição no tempo de execução, porém o grande aumento acarretou seu aumento. Isso acontece porque o aumento do número de threads é somente para umas das duas entidades (produtor ou consumidor). Pegando como exemplo 1 thread consumidora e 16 produtoras, muito provavelmente a thread consumidora não dará conta de processar os produtos na velocidade em que são gerados e muito provavelmente a memória ficará com todas as suas posições cheias, fazendo com que as threads produtoras tenham que aguardar uma posição vazia para escrever. Desta forma, o aumento do número de threads consome recursos e não ajuda na melhoria da performance do processo.

Vale ressaltar, também, que durante a medição havia outros programas em execução na máquina que podem ter afetado o tempo de execução deste programa.