COS470 - Sistemas Distribuídos Trabalho 2 - 21.1

Alunos: Guilherme Goldman e Igor Rocha

Agosto 2021

1 Decisões do projeto

No segundo trabalho prático da disciplina de Sistemas Distribuídos foi utilizado a linguaguem de programação C devido a sua robustez, ampla implementação e documentação que ela oferece. Para implementar programas que utilizam threads para um processamento multithread, a linguagem C oferece a biblioteca pthread que possuem funções prontas para serem reutilizadas e incorporadas em cada parte do projeto.
Houve a preocupação também com relação ao possíveis erros de execução e de entrada dos parâmetros, tentando sempre capturar essas exceções do código e exibindo o erro ao usuário pelo terminal. Além disso foram feitos comentários no código para especificar cada etapa do processamento das funções. O código do projeto pode ser encontrado através do seguinte repositório: `github.com/namdlog/distributed_systems/tree/main/TP2`.
Os testes e validação foram feito em um computador rodando o sistema operacional Windows porém utilizando Cygwin para emular de certa forma as funcionalidades oferecidas nos sistemas Unix, com bibliotecas e ferramentas GNU and Open Source para o compilador gcc da linguagem C utilizar a API POSIX.

2 Somador com Spinlocks

Nessa primeira parte do segundo trabalho implementados um programa que irá somar os valores de um array de maneira paralelizada utilizando o recursos de programação muiltithreads, ou seja, cada thread vai ficar responsável por somar uma parte do array e no final somar em uma variável global. O array que utilizamos foi do tipo char pois a especificação diz que devemos utilizar 1 byte para armazenar o valor numérico e 1 byte em C pode representar o número de -128 até 127 quando considerado o sinal no tipo char. Decidimos por seguir o padrão do último trabalho e pedir para o usuário inserir como argumento à chamada do programa o número de números que devem ser gerados e armazenados no array, seguido do número de threads que vão fazer o trabalho de somar as partes do array.
No começo nós decidimos verificar se a entrada está consistente com os valores

esperados do usuário. Caso esteja tudo bem iremos preencher os N valores solicitados com os número aleatórios entre -100 e 100 armazenados no array de char. Para fazer a geração de números aleatórios utilizamos a função rand() da biblioteca stdlib.h que fornece esse gerador pseudoaleatório de números. Para gerar entre 0 e 100 calculamos o mod 100, ou seja, o resto da divisão do valor gerado por 100 e no caso da decisão se seria positivo ou negativo a decisão é feita por uma moeda também pseudoaleatória, ou seja uma variável booleana caso um número gerado fosse par ou impar. Como já mencionado no parágrafo introdutório, para gestão das threads no programa foi utilizado a biblioteca pthread do C, que oferece de maneira encapsulada funções que criam e organizam as threads durante a execução do código. Em especial, utilizamos as funções e o tipo pthread_t que representa a thread em si. As funções utilizadas foram:

- pthread_create - Função que cria uma thread, passando como parâmetro o endereço de memória onde ela será armazenada, a função ou rotina que ela deve executar e os argumentos passados nessa execução.

- pthread_join - Função que irá fazer com que o processo deverá aguardar a finalização do processamento da thread passada como argumento

Antes de seguir com a criação das threads o lock é criado e instanciado em memória para que seja compartilhado entre as threads. O lock foi definido como uma struct com um atributo booleano chamado de held. Criado o lock e seguindo para a parte de criar as threads, foi utilizado uma estrutura de argumentos na qual o processo irá passar para a função responsável por somar a parte da threads. Ou seja, um argumento irá receber um valor de início de posição para ler no array e um final, assim como um lock para referenciar e um array para iterar. No caso do lock e do array apenas é passado o ponteiro das estruturas que foram declaradas anteriormente, para que todas as thredas utilizem o mesmo lock e iterem sobre o mesmo array de chars que contém os N números aleatórios que serão somados. Para que a divisão de N número para as M threads possa cobrir todo o array independentemente se N é divisível por M foi decidido que o valor que cada thread menos a última irá cobrir será floor(N/M), ou seja, o primeiro menor valor inteiro mais próximo do valor da divisão para cada thread. Ao criar os argumentos que serão passados nas funções por cada thread o controle da sobra do range coberto é passado para a última thread, assim fica garantido que todas as threads irão cobrir algum pedaço e o array é somado por completo. Por fim, o programa passa a cronometrar a execução dos blocos a seguir, utilizando a função clock(). Logo a seguir as thredas são criadas com os argumentos processados e cada thread passa a somar as partes do array. Após todas as threds terem sido criadas iremos iterar sobre o número de threads e chamar a função pthread_join para aguardar o fim de todas as thredas. Após o fim, a função clock é chamada novamente para capturar o tempo final da execução.
Para evitar deadlock foi feito o spinlock através das funções acquire e release que recebem o ponteiro do lock e controlam o lock com uma função atômica. No acquire iremos testar o booleano held através da chamada busy wait

while(__sync_lock_test_and_set(lock.held)), que irá verificar se o lock está livre e setar para true, bloqueando a região crítica. A região crítica foi definida como sendo a parte em que a thread após somar a sua parte do array vai passar o resultado pro contador global, evitando que ela seja retirada da execução preemptivamente antes de outra thread tentar somar também, podendo gerar uma condição de corrida.

3 Produtores e Consumidores

Na segunda parte do trabalho prático foi implementado um processo multi-thread com uma memória compartilha implementada no nosso caso em C, a partir de um array de int. Foi implementado um algoritmo que irá criar np threads produtoras e nc threads consumidoras, que irão preencher campos vazios do array com números primos e as threads leitoras irão recuperar esse valor e printar na tela caso o número lido em uma posição ocupada da memória compartilhada é primo ou não. Para isso seguimos o padrão de primeiro verificar se os 3 argumento do programa estão de acordo com o esperado, ou seja, se n, o número de posições do array, np, número de threads produtoras e nc, número de threads consumidoras são positivos e inteiros. Caso não seja iremos solicitar que o usuário entre com os dados corretos e sair da execução do programa. Caso contrário iremos seguir para a criação dos semáforos, que irão fazer a sincronização entre as threads consumidoras e produtoras. Foram utilizados 3 semáforos, um mutex, ou seja, um semáforo de início com o valor 1 para dar acesso à região crítica de escrita e leitura, um semáforo chamado full que irá controlar se existe alguma posição preenchida e um semáforo empty que irá controlar se existe alguma posição no array vazia. O semáforo full começa com 0 e o empty com o valor n, que é o número de posições iniciais do array. Para utilizar os semáforos da linguagem C foi utilizado a biblioteca semaphore.h, que oferece a criação de semáforos do tipo sem_t. Além disso foram utilizados para inicializar o semáforo a função sem_init que recebe o ponteiro para o semáforo, um booleano de controle e o valor inicial do semáforo. Para gerenciar a sincronização entre threads foram utilizadas as funções sem_wait e sem_post, que respectivamente fazem a thread ficar em espera caso o semáforo esteja fechado (valor em 0) ou ande com a thread caso esteja aberto e a função post que irá liberar o semáforo para que outra thread ande.
Nesse trabalho 2 a função de gerar números aleatórios e verificação de primo foi incluída em uma biblitoeca separada própria chamada 'common.h' para evitar repetição de código. Além disso tivemos 2 funções que auxiliaram na descoberta das posições do array que estão preenchidas com valores e as que não estão. Por fim, utilizamos uma função chamada free_threads que libera o fim das outras threads pois o trabalho limita a leitura de 10^5 valores e após a leitura dos valores necessários algumas threads ficaram presas no semáforo, portanto toda vez que uma thread sai de execução ela irá liberar o semáforo para que a outra thread saia de execução também.

4 Testes e estudo de caso

4.1 Spinlocks

No primeiro estudo de caso foi a execução do programa que irá somar os números do array dividindo um pedaço para cada thread e foi utilizado como entrada 3 valores distintos de N, 10^7, 10^8 e 10^9. Para cada um foram executados 10 vezes a com valores de números de threads distintos, seguindo a ordem das potências de 2: 1, 2, 4, 8, 16, 32, 64, 128 e 256. Para cada execução foi medido o tempo de execução apenas para cada thread calcular a sua parte e tirado a média desses valores obtidos de tempo no final.
Os resultados dos tempos medidos podem ser visualizados no gráfico 1 a seguir. Na coordenada x temos os números de threads e em y o tempo médio de cada ponto de cada curva. Cada curva representa um tamanho de array, que varia entre 10^7, 10^8 e 10^9. Para executar esse processamento automático foi utilizado um script bash, disponível no repositório que automatizou a execução e a coleta dos resultados.
Para nossa surpresa, esperávamos que conforme o núemro de threads aummentasse na execução do processo o tempo iria diminuir, porém não foi o observado. Pelo que foi possível refletir, acredita-se que o resultado de um processamento com mais threads não diminuiu o tempo pois ao aumentar o número de threads ficando superios ao número de núcleos da CPU pode acontecer um overhead na troca de contexto e o processamento não ser tão paralelo assim e não valer tanto a pena tantas threads para obter muito alto desempenho em um computador comum.
A partir do gráfico podemos visualizar que conforme os números de threads aumentam

4.2 Produtores e Consumidores

No segundo estudo de caso foi utilizado um script que irá executar 10 vezes e medir o tempo para cada combinação de número de valores no array, número de threads produtoras e threads consumidoras. Após a obtenção desse número foi utilizado um notebook em python para plotar os valores coletados. Segue a imagem do gráfico de tempo de execução médio de 10 rodadas para cada combinação em 5 Ns diferentes, 1, 2, 4, 16 e 32. Foi utilizado um script em python que leu as exportações do tempo de execução de cada rodada e tirou-se a média das 10 execuções para cada combinação.
Pela análise do gráfico 2 de cada curva parece que conforme o número de threads consumidoras aumentam o tempo de execução aumenta pois elas terão mais threads aguardando o processamento de poucas threads produtoras, ao passo que conforme o número de produtoras aumentam o consumo é imediato e elas passam a ter tempos mais sincronizados de leitura e escrita, alcançando a meta mais rápido.

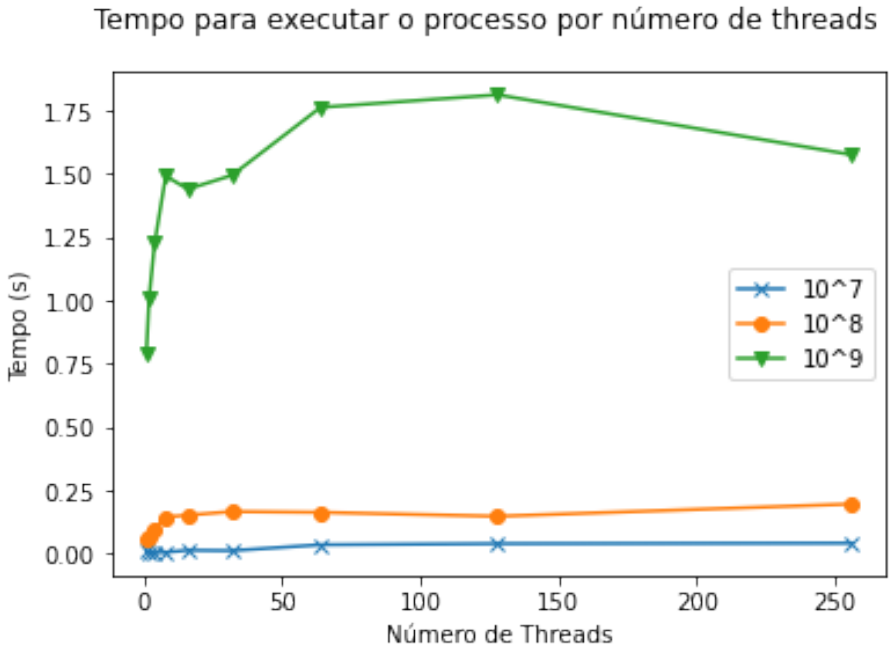

Figure 1: Gráfico de estudo de caso da parte 1 - Spin Locks

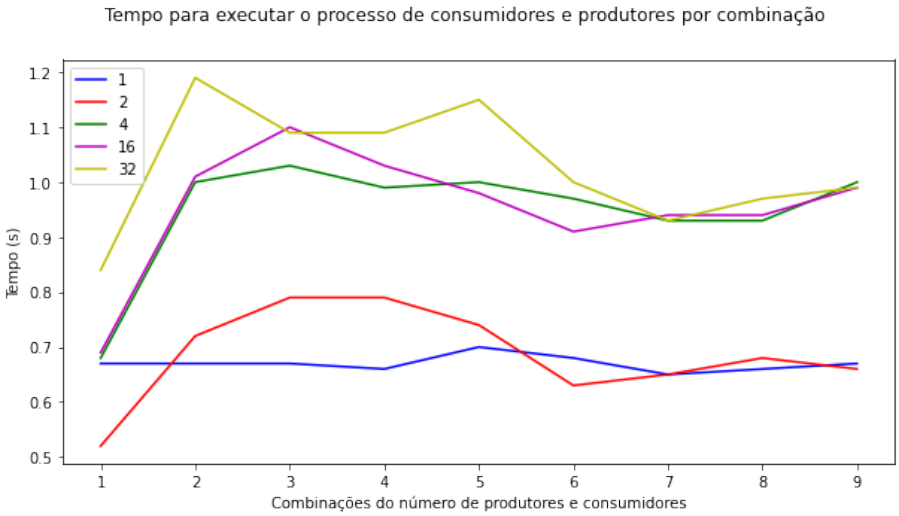

Figure 2: Gráfico do estudo de caso da parte 2 - Produtores e Consumidores