Sistemas Distribuídos – COS470

2021/1

Trabalho Prático 2

1 Objetivos

O objetivo deste trabalho é se familiarizar com *threads* e mecanismos de sincronização entre *threads* (e processos) que são instrumentais para coordenar o acesso a memória compartilhada e a execução das *threads*. Para cada parte, você deve desenvolver um programa na linguagem de programação de sua preferência mas que tenha suporte a threads e mecanismos de sincronização, como instruções atômicas e semáforos. A sugestão é utilizar C ou C++, tendo em vista a proximidade das bibliotecas destas linguagens com o sistema operacional, oferecendo ao desenvolvedor (você) um maior controle.

Além da implementação, você deve testar seu programa, rodando os estudos de casos. Você deve preparar um relatório, com no máximo 6 páginas, com as decisões de projeto e implementação das funcionalidades especificadas, assim como a avaliação dos estudos de caso. O relatório deve conter a URL para o código-fonte da sua implementação. O trabalho deve ser realizado em dupla.

2 Somador com *Spinlocks*

Considere o problema calcular a soma de um grande número de números. Você deve projetar e implementar um programa *multithreaded* para resolver este problema, visando reduzir o tempo de execução. Assuma que os números a serem somados estão armazenados em um vetor de tamanho N e seus valores são números aleatórios no intervalo $[-100, 100]$. Seu programa deve ter como parâmetro K, que representa o número de threads que irão trabalhar simultaneamente. Uma forma de resolver o problema é dividir a soma em K parcelas com N/K valores e deixar cada thread realizar a soma de cada parcela, somando o resultado em um acumulador compartilhado. Como várias threads vão escrever no acumulador, o acesso ao mesmo deve ser serializado.

Para serializar o acesso ao acumulador compartilhado, implemente um *spinlock* como vimos em aula. Você deve implementar o *spinlock* utilizando instruções atômicas, como a *test-and-set*, utilizando *busy wait* para bloquear a thread. Em particular, implemente as funções *acquire()* e *release()* para entrar e sair da região crítica. Defina sua região crítica adequadamente.

Para o estudo de caso, considere $N = 10^7$, $N = 10^8$ e $N = 10^9$. Você deve alocar memória utilizando apenas um byte para cada número, e preencher o vetor de forma aleatória (com números no intervalo $[-100, 100]$). Utilize os valores de $K = 1, 2, 4, 8, 16, 32, 64, 128, 256$. Para cada combinação de valores N, K, obtenha o tempo de execução do seu programa, rodando o programa 10 vezes para calcular o tempo médio de execução (para cada combinação de valores, teremos um tempo médio de execução). Não considere o tempo necesário para gerar e preencher o vetor com N posições, mas apenas o tempo para calcular a soma. Apresente um gráfico mostrando o tempo médio de execução em função do número de threads para cada valor de N (cada N deve ser uma curva no gráfico).

3 Produtor-Consumidor com Semáforos

Implemente um programa Produtor-Consumidor *multithreaded* com memória compartilhada limitada. Assuma que a memória compartilhada é um vetor de tamanho N de números inteiros. O valor 0 denota que a posição do vetor está livre. O número de threads do tipo produtor e consumidor são parâmetros do programa dados por N_p e N_c, respectivamente. A thread

produtor deve gerar números inteiros aleatórios entre 1 e 10^7 e colocar o número em uma posição livre da memória compartilhada. A thread consumidor deve retirar um número produzido por um produtor da memória compartilhado, liberar a posição do vetor, e verificar se o mesmo é primo, imprimindo o resultado no terminal.

Repare que a memória compartilhada será escrita e lida por várias threads, então o acesso deve ser serializado, evitando efeitos indesejáveis da condição de corrida. Utilize semáforos para serializar o acesso à memória compartilhada. Repare ainda que quando a memória compartilhada estiver cheia ou vazia (sem posições livres ou sem posições com números) as threads produtor ou consumidor devem aguardar bloqueadas, respectivamente. Ou seja, uma thread produtor aguarda até que haja uma posição de memória livre, e uma thread consumidor aguarda até que haja uma posição de memória ocupada. Utilize semáforos contadores para esta sincronização (que devem ter como parâmetro N), como vimos em aula.

Dica: Você pode implementar um algoritmo bem simples para determinar uma posição livre e uma posição ocupada na memória compartilhada (que pode ser implementada como um vetor). A thread consumidor deve copiar o número a ser processado para sua memória local, liberando o espaço na memória compartilhada.

Para o estudo de caso, considere que o programa termina sua execução após o consumidor processar $M = 10^5$ números. Considere ainda os valores $N = 1, 2, 4, 16, 32$, com os seguintes combinações de número de threads produtor/consumidor:
$(N_p, N_c) = \{(1,1), (1,2), (1,4), (1,8), (1,16), (2,1), (4,1), (8,1), (16,1)\}$.

Para cada combinação de parâmetros, obtenha o tempo de execução do seu programa, rodando o programa 10 vezes para calcular o tempo médio de execução (para cada combinação de valores, teremos um tempo médio de execução). Apresente um gráfico mostrando o tempo médio de execução em função do número de threads produtor/consumidor para cada valor de N (cada N deve ser uma curva no gráfico). Analise o comportamento observado.

Por fim, determine a combinação de (N_p, N_c) que possui o melhor desempenho (ie., menor tempo de execução). O que você pode concluir?