Programação Concorrente:

RELATÓRIO DO PROJETO 1

Victor do Valle Cunha (20104135)

Eduardo Silveira Godinho (20100522)

22 de Agosto de 20

1-OBJETIVOS

Explorar os conteúdos teóricos e práticos da disciplina de programação concorrente por meio de um projeto prático proposto pelos professores, que consiste em manipular e sincronizar threads a partir dos conceitos de mutexes e semáforos tudo o que foi visto durante o decorrer das aulas,.

2-DESENVOLVIMENTO

A primeira implementação realizada pelo grupo foi a sincronização dos vikings para adquirir uma cadeira e dois pratos, respeitando as regras as quais um viking normal e um berseker não podem sentar lado a lado. Para isso, utilizamos um buffer *(buffer_mesa*) do mesmo tamanho ao número de cadeiras e que em cada espaço foram armazenados três valores: -1 representando que o espaço está vazio, 0 um viking normal e 1 um viking berserker. Em seguida, utilizamos mutexes para exercer duas funções. Primeiro, controlar que múltiplos vikings possam alterar o valor do buffer ao mesmo tempo. Segundo, para impedir que no momento da verificação da disponibilidade do lugar em que o viking deseja sentar não haja troca de valores dentro do buffer.

Para implementar tudo isso, dividimos a função *adquire_seats_plates* em duas partes. A primeira resolvendo o caso da mesa possuir menos de um lugar e mais de um. No primeira parte, é feito um lock no mutex[0], verifica-se que a única posição da mesa está vazia e colocando 1 ou 0 de acordo com o tipo de viking

No entanto, se a mesa possuir mais lugares, é feito um laço para tentar bloquear uma cadeira por meio da checagem se *trylock(posicao_mesa) == 0* . Se ele conseguir dar lock naquela posição chama-se uma função auxiliar *posicao_disponivel* que recebe como entrada o buffer, posição da mesa, e se ele é um berserker. O retorno 1 mostra que a posição que foi enviada pode ser ocupada pelo viking e segue as restrições que geram conflitos, ou 0 caso o contrário. Caso a posicão esteja disponível o buffer recebe o número do berserker naquela posição e, em seguida, chama a função *pega_prato* que utiliza dois mutex seguindo a lógica de implementação do jantar dos filósofos: o primeiro que sentar na mesa pega dois pratos invertidos do resto. Após isso, o mutex *controla_busca* recebe um unlock e retorna a posição da cadeira pela função *chieftain_acquire_seat_plates.* Porém, se o retorno for 0 o loop continua até conseguir uma posição vazia no mutex e disponível segundo a regras disponíveis

A próxima implementação foi a espera do término do banquete de todos os vikings para o início das preces. No chieftain é inicializado um semáforo chamado *termina_comer,* um mutex chamado *organ_comida* e um contador *soma_comidos*, na funcao *chieftain_init* ambos são inicializados e o contador comeca em zero, e são destruídos na *chieftain_finalize,* assim

como todos os outros mutexes e semáforos utilizados. Toda vez que um viking entra na função *chieftain_release_seat_plates* para liberar o lugar é feito um lock no mutex, o *soma_comidos* é somado em +1 e um unlock no mutex, para garantir posse exclusiva na variável. Logo após isso, é verificado se o *soma_comidos* é igual o *horde_size,* para identificar se é o ultimo viking a levantar da mesa.Caso isso aconteça é feito *2*horde_size* posts (2*horde size pois o número de vikings totais pode ser até o dobro dos quais já tinham devido aos LATE_VIKINGS) no semaforo *termina_comer,* no qual função *chieftain_get_god* possui um wait nesse mesmo semáforo, fazendo os vikings que já terminaram de comer e os que só chegam para rezar ficarem trancados até que o último viking termine de comer e libere esse semáforo 2*horde_size para que todos consigam pedir um deus para realizar as preces.

Em seguida, o chieftain escolhe um deus para cada viking rezar seguindo todas as limitações esclarecidas no enunciado do problema. Assim, foi utilizado NUM_GODS+6 semáforos (estava dando erro de memória mesmo usando menos posições, mas só é utilizado um semáforo para cada deus), inicializados e finalizados pelo chieftain junto com os outros.

Após a criação dos semáforo inicializados com uma posição livre para cada, e de todos terminarem de comer a função *chieftain_get_god* ira fazer um loop infinito e dentro desse loop será gerado um número aleatório de 0 a 7 para indicar o deus (*num_rand*), com o auxílio da função rand(), após isso é feito um condicional.

Caso sem_trywait (semaforo[num_rand]) == 0, para ver se aquele deus tem espaço para receber preces já que deuses rivais não podem se distanciar muito no número de preces, caso ele consiga dar wait no semáforo do deus fornecido pela função rand(), é feito um post no semáforo do deus rival com o auxílio da função *valhalla_get_rival* , e um post em cada super deus (ODIN E THOR), pois na regra deles eles podem receber até 10% mais do que a soma de todos, entao nao tem problema liberar para receber mais preces, e é retornado o deus que aquele viking pode realizar a prece.

Caso ele não consiga dar wait no semáforo do deus fornecido pela função rand, ele simplesmente recebe outro valor randômico e tenta novamente, até achar um semáforo com uma posição livre.

3. CONSIDERAÇÕES FINAIS

Desde o início, o trabalho apresentou dificuldades elevadas em tentar conciliar dois aspectos da programação concorrente: o aumento do desempenho por meio da concorrência e o controle dessa concorrência. A construção da função *adquire_seats_plates,* por exemplo, foi reformulada muitas vezes e ainda continua possuindo algumas falhas. A busca de um espaço vazio não é tão eficiente quanto esperado, pois tem que varrer todos os espaços. O trylock

poderia ser substituído por uma implementação melhor e menos custosa, pois embora a intenção de seu uso fosse criar um mecanismo para que as threads ficassem insistentemente tentando pegar uma cadeira, tínhamos noção que esse mecanismo causa busy-wait e, por isso, uma queda desempenho e lentidão.