Programação Concorrente:

Relatório do Projeto II

Eduardo Silveira Godinho (20100522)

Victor do valle Cunha (20104135)

1. INTRODUÇÃO

Este relatório consiste na explicação para a proposta de solução desenvolvida pelos estudantes Victor do Valle Cunha e Eduardo Silveira Godinho em um problema para a disciplina de programação concorrente. Em suma, o problema descrevia um cenário de uma academia de natação, a qual nadadores e funcionários eram vistos como threads e precisavam ser sincronizados para se obter o comportamento desejado. Para isso, os alunos procuraram desenvolver a solução do problema por meio mecanismos de controle de concorrência, sempre tendo em vista as novas particularidades que obteve ao desenvolver um programa concorrente em Python.

2. DESENVOLVIMENTO

2.1 Estratégias adotadas para controlar cada recurso

Para controlar os acessos aos armários, foram criados dois semáforos, *sem_arm_masc e sem_arm_fem,* um para cada gênero. Inicializados no arquivo init com um contador inicial equivalente a variável global *quant_armarios_por_vestiario,* eles seguem a lógica comum do uso de semáforos a qual permite **n** threads de acessarem uma região crítica, sendo **n** o número definido em seu contador. De maneira análoga, o monitoramento das pranchas foi feito com o semáforo *sem_prancha* e a variável global *quant_pranchas*.

O acesso aos vestiários pelos nadadores e nadadoras, por sua vez, foi simplesmente implementado através da inclusão do *id* nas lista*s vestiario_mas* e *vestiario_fem,* uma vez que não foi imposto um limite de pessoas para cada vestiário. As duchas, no entanto, necessitam o uso dos semáforos *sem_ducha_mas* e *sem_ducha_fem* com um contador com o mesmo número de duchas definidas no exercício a fim de controlar que o número de pessoas tomando banho seja equivalente ao número de duchas, assim como, no caso do vestiário masculino, impedir que nenhum nadador tome banho enquanto o funcionário não termine de limpar todas as duchas.

Por fim, para controlar a entrada de nadadores em suas raias se utiliza um semáforo *sem_raia* que é inicializado com um contador com duas vezes a quantidade de raias. Isso se deve ao fato que se todas as raias só conterem crianças é possível ter o dobro de nadadores ao mesmo tempo. Nesse sentido, a lógica de implementação foi criar uma fila de espera para todos os nadadores, crianças e adultos, por meio de um *lock escolha_raia,* seguindo o

raciocínio descrito no enunciado do trabalho o qual propõe que a medida que uma criança saia da piscina e se obtenha um cenário o qual duas delas estão sozinhas, deve-se supor que uma delas migre para a raia da outra. Assim, toda vez que uma thread consegue acessar a região desse *lock,* é verificado se a mesma é uma criança ou um adulto. No caso de ser uma criança, olha-se para o valor salvo na variável global *raias_ocupadas,* se este estiver com valor superior à 7.5 essa thread tenta adquirir o semáforo *sem_raia* e fica suspensa até que uma outra thread de um *release.* Se não estiver com o valor superior, inclui-se o id desse objeto na lista *piscina* e incrementa a variável *raias_ocupadas* em meio. Já no caso de ser um adulto, segue-se a mesma lógica, com a diferença que ao em vez de analisar *raias_ocupadas* com um valor superior à 7.5, analisa-se com o valor 7 para impedir que adultos entrem em raias totalmente ocupadas ou com apenas uma delas com uma única criança e o resto completamente ocupado. Além disso, também é feito dois *adquire* e incrementa-se 1 em *raias_ocupadas.*

2.2 Controle do funcionário no vestiário feminino

Para garantir que o funcionário espere todas as nadadoras saírem do banheiro para realizar a limpeza, utiliza-se um *condition*, uma lista *vestiario_fem*, e uma flag *vest_fem_sendo_limpado.* A implementação é feita da seguinte forma: sempre que uma nadadora entra no vestiário é visto se a flag *vest_fem_sendo_limpado* está com valor 1. Caso esteja, ela é suspensa no *condition* a fim de evitar *busy-wait*. No entanto, caso contrário, ela é adicionada na lista. Na saída do vestiário, ela é retirada da lista e logo em seguida é visto se a lista está vazia para então a thread da nadadora realizar um *notify* no funcionário. Este, por sua vez, sai do estado suspenso, e o mesmo configura a *flag* para 1, impedindo que novas nadadoras entrem no vestiário. A *flag* é configurada para 0 no fim da limpeza, e o funcionário realiza um *notify all* o qual avisa todas as threads nadadoras que o banheiro está pronto para uso

2.3 Starvation e DeadLock

Inicialmente, no mecanismo de controle para o acesso às raias ocorria um deadlock. Quando a piscina com apenas um lugar disponível e uma thread nadadora adulta que já houvesse passado da verificação de raias ocupadas e uma segunda thread adulta fosse disparada antes que a primeira desse seu segundo *adquire*, pois a primeira ficaria esperando

pela segunda e a segunda pela primeira. Nesse sentido, foi adicionado o *lock escolhe_raia* com a finalidade de garantir exclusividade da thread para realizar os dois incrementos no semáforo sem que outra thread o interrompa.

Com relação à possibilidade de *starvation*, houve o cuidado de impedir que uma thread morresse abruptamente com algum recurso e outra bloqueada pela falta deste ficasse esperando infinitamente até que ficasse disponível o recurso. Para isso, bastou dar um *join* no final do arquivo *init* para impedir que thread morresse repentinamente. Além disso, como os outros recursos utilizam semáforos, no caso dos armários e das raias, ou *locks* associados a *condition,* como as duchas e as raias, garante-se as threads não continuem em execução após a negativa de acesso a esses recursos. Dessa forma, inviabilizou-se a possibilidade de inúmeras requisições negadas a uma thread para o acesso a um recurso - *starvation*- com essa em contínua execução e exigindo tempo de processamento desnecessário.

2.4 Processos vs Threads: uma análise de desempenho

Para resolução do problema foi requisitado que nadadores e funcionários fossem vistos como threads e os estudantes acreditam que essa é a melhor forma de se obter o maior desempenho do programa. Isso porque o uso de processos, embora possibilite paralelismo real, demanda muito mais recursos computacionais e é mais lenta quando é preciso compartilhar os dados, pois necessita de um meio de comunicação como pipes, array ou filas. Dessa maneira, tendo em vista a forma como foi implementada o nosso programa, com bastante proveito de variáveis globais, o uso de processos poderia deteriorar o desempenho da solução proposta.

3. CONCLUSÃO

Por fim, acredita-se que o trabalho desenvolvido cumpriu objetivos propostos, pois embora exista a dificuldade inerente de programas concorrentes de visualizar se de fato o programa está tirando proveito da concorrência e ao mesmo tempo entregando o resultado esperado, os estudantes procuraram discutir incessantemente em vídeo-chamadas semanais a melhor abordagem para o melhor gerenciamento de cada recurso. Ademais, também debateram a respeito das particularidades de se programar concorrentemente em Python e C, como por exemplo a percepção que o aumento no controle do gerenciamento de memória diminui a produtividade.