Título
MAC0438 - Programação Concorrente

André Meneghelli Vale - 4898948
andredalton@gmail.com

Marcello Souza de Oliveira - 6432692
mcellor210@gmail.com

1 Problema das abelhas e ursos.

1.1 Introdução:

O problema deste EP deve ser resolvido utilizando monitores para controlar a concorrência. Os monitores que devem ser implementados serão o `abelha` e o `urso`. O método de resolução do problema está descrito nas subsessões a seguir.

1.2 Parâmetros de entrada:

O programa é um script em python que necessita dos seguintes parâmetros de entrada:

- `N`: número de abelhas;
- `B`: número de ursos;
- `H`: capacidade do pote;
- `t`: tempo gasto pelas abelhas no pote;
- `T`: tempo gastto pelos ursos pra se alimentarem;

Os parâmetros acima são obrigatórios e devem seguir a ordem correta e serem números naturais. Também é possível realizar a construção automática dos gráficos necessários ao relatório passando um último argumento `-g`.

1.3 Controle do tempo:

Como o problema descreve uma simulação temporal discreta, é possível fazer grandes saltos no tempo real que está sendo simulado diretamente aos pontos de interesse, onde é necessário imprimir as seguintes informações se ocorrer algum evento especial:

- Estado das abelhas: enchendo o pote, parada porque o urso está comendo mel, voando porque não tem espaço para encher o pote;
- Vezes que cada abelha acordou um urso: esta impressão tomava grande espaço na tela dependendo da quantidade de abelhas, portanto foi tomada a decisão de imprimir a média deste evento, tal decisão não implicou em cálculo adicional ao programa pois ainda é necessário povoar um arquivo com as médias obtidas quando o programa estiver rodando em modo gráfico;
- Quantidade de vezes que cada urso foi acordado: devido as mesmas circunstâncias levas em consideração no ítem anterior, para este evento também foi impresso apenas a média.

Os eventos especiais são:

- *Pote cheio*: implementado no monitor abelha;
- *Pote na metade*: implementado no monitor urso quando este está comendo e no monitor abelha quando esta está trabalhando;
- *Pote vazio*: implementado no monitor urso assim que este finaliza a sua alimentação.

A implementação do fluxo de tempo para o problema se deu através da classe `DeLorean`. Cujo funcionamento é idêntico ao funcionamento de um relógio de competição de xadrez, neste caso as abelhas tem um relógio e os ursos outro fazendo com que o tempo total seja a soma dos tempos das abelhas e dos ursos. Isso permite que saltos no tempo fiquem contidos dentro do escopo de funcionamento de cada um dos monitores.

Um objeto desta classe fica localizado em escopo global para que qualquer um dos objetos monitores possa realizar o salto no tempo quando necessário. O nome do objeto é `time_machine`.

1.4 Término do programa:

O término do programa é informado a todos os monitores por meio de uma variável global `run`. Ela contém 3 estados possíveis:

- *0*: programa sendo inicializado;
- *1*: programa rodando;
- *2*: programa encerrando, ele se considera encerrado uma vez que todos os ursos tenham recebido 10 potes de mel.

1.5 Abelhas:

O monitor abelha controla as `N` threads abelhas.

As abelhas trabalham todas em conjunto dentro das limitações impostas pelo problema (tamanho do pote e número máximo de 100 abelhas por pote). Para tanto existe uma barreira ao final de cada um dos métodos principais da abelha para que todas entrem e saiam gerando um salto no tempo de maneira correta.

O monitor abelha contém dois tipos distintos de semáforo. O primeiro serve para que o número correto de abelhas acesse o pote. Este número é definido da seguinte maneira:

- *H*: quando o tamanho do pote é inferior à `100`;
- *100*: quando o pote tem tamanho superior à `100`.

O segundo serve para tornar a parte de acesso ao recurso `pote` do método `run` do monitor abelha. Este método é o que controla o fluxo da thread em uma aplicação `python` que utilize o pacote `threading`. Para tando a classe implementada deve herdar a classe `threading.Thread` deste pacote. Potanto o segundo semáforo tranca a sequência do método assim que uma das abelhas consiga acesso ao `pote`.

Uma vez que a abelha esteja seguindo pelo método e tenha acesso único ao pote, ela deve seguir uma série de exigências para poder fazer a alteração no conteúdo do pote e talvez imprimir as informações de um evento especial.

Para poder alterar o conteúdo do pote é necessário que esta abelha esteja no mesmo intervalo de tempo das abelhas que estão trabalhando atualmente. É necessário tomar esta precaução já que uma abelha pode ter terminado de encher o pote e neste caso o trabalho da abelha em questão foi totalmente perdido já que o pote foi levado ao urso (como o acesso das abelhas é injusto, não há necessidade de garantir que esta abelha esteja no pote assim que ele volte, então ela vai ter que competir novamente pelo acesso).

Para disparar o evento de pote na metade enquanto as abelhas trabalham, basta que esta abelha tenha atingido a metade do pote quando teve acesso ao conteúdo deste.

Para disparar o evento de pote cheio, basta que esta abelha tenha completado o pote quando teve acesso ao conteúdo deste. Neste caso, como o pote estava fora de alcance das abelhas, acontece um salto no tempo, consequentemente fazendo com que todas as abelhas que ainda pudessem trabalhar neste período tenham que competir novamente pelo acesso ao pote (como explicado anteriormente).

Se todas as abelhas possíveis de trabalhar neste tic de tempo trabalharam e esta for a última abelha, então esta deve fazer o salto no tempo necessário. Observe que desta vez não ocorrerá o problema de impedir que uma abelha que teve o acesso ao pote garantido seja cancelado, já que todas as outras abelhas já o acessaram.

O objeto `time_machine` já é inicializado com o tempo gasto por uma abelha no pote, desta forma fazendo com que todas as abelhas tratem o evento necessário no tempo correto e evitem que um salto no tempo tenha que ser repassado ou impedido de maneira complexa entre os monitores.

1.5.1 wait()

O monitor abelha implementa o método `wait()` que controla o fluxo de acesso das abelhas. O `wait()` irá segurar as abelhas caso um urso esteja acordado, ou existam 100 abelhas no pote ou então se o pote está cheio. As abelhas serão liberadas caso o programa indique que chegou ao fim por meio da variável global `run`.

1.5.2 signal_all()

O monitor abelha implementa o método `signal_all()` que atualiza a variável `sleep` para `True`. Esta variável indica se existe ou não um urso acordado. Este `signal_all()` acaba servindo tanto para acordar as abelhas como para fazer os ursos dormirem (uma vez que este método é a ultima coisa que o urso chama antes de dormir).

1.6 Ursos:

O monitor urso controla as `B` threads ursos.

Como todos os ursos precisam receber alimentos e foi definido inicialmente que cada urso receberia 10 potes de urso cada vez que o programa rodar então existe uma variável global `roleta` cujo conteúdo é o urso atual que pode estar acordado. A modificação deste valor acontece pelo método `roletaUrsa()` implementado no monitor urso. Este método incrementa em 1 o valor de `roleta` e atribui então à esta variável o conteúdo do resto da divisão dela pelo número de ursos `B`.

1.6.1 wait()

O monitor urso implementa o método `wait()` que controla o fluxo de acesso das abelhas. Este método irá segurar os ursos uma vez que seu `id` seja diferente que `roleta` ou que todos os ursos estejam dormindo ou que o pote não esteja cheio. O a restrição do pote estar cheio impede que outro urso tenha acesso ao pote depois que o urso atual rode `roletaUrsa()` e antes de ter terminado de rodar o método `signal_all()` nas abelhas, chamado por este urso, fazendo com que as abelhas voltem ao trabalho e todos os ursos durmam.

1.6.2 signal()

O monitor urso implementa o método `signal()` que atribui `False` para a variável global `sleep`. Fazendo com que todas as abelhas parem de trabalhar e permitindo que um dos ursos (`id=roleta`) possa ter acesso ao `pote`.

2 Gráficos:

Para este programa foram implementados 3 tipos de gráficos para este relatório. Eles estão agrupados por tipo estão inseridos à seguir:

2.1 Ursos:

Este gráfico contém a média do número de vezes que cada urso comeu. É interessante observar o formato do gráfico estritamente crescente, porém em forma de escada. Isso se deve ao fato de que as abelhas só podem produzir quando não existe nenhum urso comendo (ou seja, todos estão dormindo).

2.2 Abelhas:

Este gráfico contém a média do número de vezes que cada abelha acordou um urso. Neste gráfico também é possível notar um formato de escada, já que as abelhas não produzem enquanto um urso se alimenta.

2.3 Histograma:

Também achamos interessante criar um gráfico que contabilize o acesso das abelhas ao pote. Portanto criamos um histograma com o número de acessos ao pote pelo id da abelha. As abelhas tiveram acesso muito bem distribuído ao pote mas não justo, fazendo com que algumas abelhas acessem mais o pote que outras.

3 Conclusão

Foi possível observar que o desenvolvimento de monitores para controle de acesso a recursos é algo bastante útil quando o problema tenha uma complexidade concorrete bastante elevada e a sessão crítica acabe sendo acessada em sequência. Neste caso é interessante criar uma classe que formalize de maneira mais adequada o problema. No entanto quando apenas poucas áreas de sessão crítica existem e elas tomam muito pouco tempo de processamento, então esta é uma solução pouco eficiente.