JANTAR DOS FILÓSOFOS

O jantar dos filósofos foi proposto por Edsger W. Dijkstra em 1965 e é considerado um dos problemas clássicos sobre sistemas operacionais. Trata-se de um problema sobre a sincronização na comunicação entre processos e threads em sistemas operacionais.

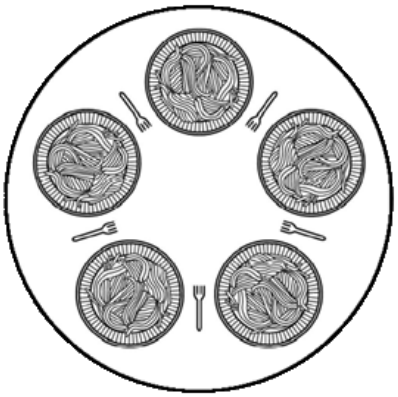
- Explicar processo

Programa que está sendo executado. Possui um único fluxo de execução.

- Explicar thread (tarefa)

Menor unidade de código que pode ser executada, ou seja, dentro de um processo podemos ter várias threads. Possui múltiplos fluxos de execução.

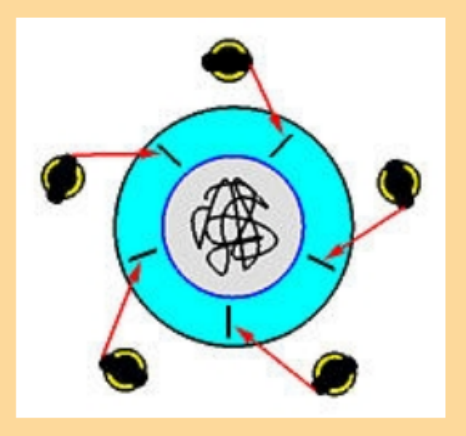
O problema é definido como: Cinco filósofos estão sentados em uma mesa redonda, eles alternam entre duas tarefas: comer ou pensar. Quando um filósofo fica com fome, ele tenta pegar os garfos à sua esquerda e à sua direita; um de cada vez, independente da ordem. Caso ele consiga pegar dois garfos, ele come durante um determinado tempo e depois recoloca os garfos na mesa. Em seguida ele volta a pensar.



Objetivo: criação de um algoritmo que seja implementado de modo que cada filósofo execute suas ações sem ficar travando.

Soluções

1º - Fazer com que todos peguem um garfo.



Problema 1 - Se todos os filósofos pegarem o hashi da esquerda, por exemplo, todos vão ficar aguardando e nenhum pegará o da direita, ocasionando um DEADLOCK.

Deadlock: quando dois ou mais processos são impedidos de continuar suas ações. Ocorre quando um processo precisa de um recurso do sistema para prosseguir sua execução mas já está sendo usado por outro processo, que também fica esperando liberar algum recurso utilizado por outro processo, resultando em todos estes processos bloqueados, esperando uns pelos outros.

2º - Após pegar um hashi, verificar se o outro está livre.

Após pegar o hashi da esquerda, o filósofo verifica se o da direita está livre. Se não estiver, devolve o hashi que pegou, espera um pouco e tenta novamente.

Problema 2 - Se todos os filósofos pegarem o hashi da esquerda ao mesmo tempo:

1. Verão que o da direita não está livre
2. Largarão seu hashi e esperarão
3. Pegarão novamente o hashi da esquerda
4. Verão que o da direita não está livre
5. ... (Looping infinito)

Teremos um caso de starvation (inanição), onde o filósofo poderá morrer de fome. Ele poderá tentar fazer a mesma ação de pegar e depois devolver o hashi para sempre, nunca conseguindo comer.

Precisamos evitar o deadlock e o starvation…

Código:

Em nosso código é criada a classe *Filosofo*, que irá representá-los, com suas ações e características. Ela recebe a implementação da interface Runnable. Um método construtor é definido para preencher os atributos dos filósofos: seu id, o tempo comendo e a identificação do hashi a sua esquerda e o da sua direita. Além do mais, um tempo máximo para o filósofo pensar foi estabelecido, sendo ele de 1 segundo, e um objeto random foi criado e será utilizado em alguns momentos ao decorrer da classe. Em seguida temos a função pensar, que imprime na tela o filósofo que está pensando, através do random estabelece o tempo que o filósofo pensou e executa o método sleep da classe Thread, que coloca a thread para dormir por certo intervalo de tempo. Continuando a classe, temos a função pegarGarfo, que determina qual será a ordem que o filósofo irá pegar os garfos da mesa, baseado em uma estrutura de if, com auxílio do random, imprimindo no console a ordem de execução dos movimentos. Logo depois, seguindo o mesmo raciocínio da anterior, temos a função soltarGarfo, que determina através do random a ordem que o filósofo irá colocar os garfos de volta na mesa, imprimindo essa informação no console. A última função da classe é comer, que informa através de um sysout que qual filósofo está comendo e executa o método sleep, pausando essa thread durante um tempo. Por fim temos o método run com as funções da classe em seu interior, possibilitando a execução e o looping entre elas.

Também ocorre a criação da classe *Garfo*, que irá representá-los, com suas ações e características. Um método construtor é criado para preencher os atributos dos grafos: seu id e o filósofo que está segurando, a princípio este é definido como null, ou seja, o garfo não está com nenhum dos filósofos. Em seguida é criado o método pegar com o uso de synchronized, garantindo que uma thread por vez execute este método, isso ocorre através de um mecanismo de lock, no qual a thread que inicia o método captura o look e só o libera no fim da execução. Além disso, esse método possui em sua assinatura o throws, ou seja, as exceções não

serão tratadas. Ainda no método pegar, através de um if, verificamos se alguém está segurando o garfo, caso esteja, o filósofo espera, por meio do método wait(), até que o garfo fique disponível. Se ninguém está segurando, o filósofo pega o garfo, onde é usado o método currentThread, responsável por retornar a thread em execução no momento. Para fechar a classe temos a função soltar, que também faz uso do synchronized, esta verifica, por meio de um if, se o filósofo está segurando um garfo, caso esteja coloca o garfo na mesa e finaliza notificando que a thread pausada pode continuar, através de um notify().

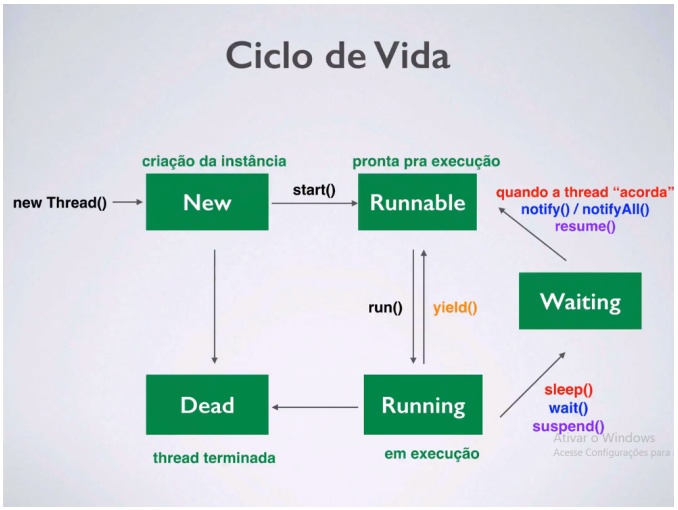
A terceira e última classe é a *Jantar*, seu método mean possui em sua assinatura o throws. Nela são descritos seus atributos: totalFilosofos e tempoComendo. Dois ArrayLists são construídos, um para classe garfo e outro para classe filósofo. Em seguida duas estruturas for são criadas, a primeira executa a criação dos garfos de acordo a quantidade de filósofos inseridos, em nosso exemplo são cinco, e a segunda descreve quem será o garfo próximo do outro. Através do método add do ArrayList, os filósofos são criados e cada um recebe seu id, o tempo comendo, seu garfo à esquerda e a direita. Por fim, um for each dá start em todos os filósofos.

Considerações finais

Esse problema é útil para modelar processos que competem por acesso exclusivo a um número limitado de recursos. Remete a um problema clássico de programação concorrente, inicialmente utilizada na construção de sistemas operacionais e atualmente aplicada no desenvolvimento de aplicações em todas as áreas da computação.

➔ ***Anotações (não é obrigatório explicar, mas é bom saber):***

- Thread no java e comandos



start() - thread pronta para execução, inicia o método run.

run() - método que possibilita a execução da thread.

sleep() - coloca a thread para dormir por determinado tempo.

wait() - estado de espera *(espera indefinida até que outro thread invoque notify())*.

notify() - notifica que a thread pausada pode continuar.

- Runnable

Interface com o método run.

- Tratamento de exceptions

```
try {
   //bloco monitorado (teste)
} catch (tipoException exception) {
   //tratamento do erro
}
```

Se não tiver try e catch, na primeira vez que ocorrer um erro o programa vai parar a execução.

- printStackTrace()

Imprime o StackTrace, a descrição do erro com a linha e classe.

- Throws

As exceções não serão tratadas nesse momento.

- Synchronized

O uso de synchronized em um método garante que a execução deste método seja realizada apenas por uma Thread por vez, utilizando um mecanismo de lock. A Thread que começa a executar o método “pega” o lock, liberando-o ao término da execução do método.

- currentThread

Retorna a thread em execução no momento.

Referências bibliográficas

O que é o problema dos filósofos glutões?. StackOverFlow. Disponível em: <https://pt.stackoverflow.com/questions/283375/o-que-é-o-problema-dos-filósofos-glutões>.

O que é synchronized?. GUJ. Novembro de 2012. Disponível em: <https://www.guj.com.br/t/o-que-e-synchronized/139744/4>.

Wait and notify () methods in java. Baeldung. 04/06/2022. Disponível em: <https://www.baeldung.com/java-wait-notify>.

Java Thread currentThread() method. javaTpoint. Disponível em: <https://www.javatpoint.com/java-thread-currentthread-method>.

FONSECA, Mateus. TRABALHO DA DISCIPLINA DE SISTEMAS OPERACIONAIS, JANTAR DOS FILÓSOFOS - UFC. Disponível em: <https://www.youtube.com/watch?v=PWj6yX4-LY8&t=326s>.

Vídeos do canal Loiane Groner.