Segundo Trabalho Prático de Implementação
Sistema de Reserva de Assentos

Computação Concorrente (MAB-117) - 2016/2
Prof. Silvana Rossetto

Vinícius Garcia Silva da Costa
Rafael Fontella Katopodis

SUMÁRIO

1) Esboços Iniciais

Após a leitura inicial da proposta do trabalho, o que mais nos saltou aos olhos foi como o sistema deveria ser composto por diversos sistemas menores, relativamente independentes entre si. Como decidimos realizar nossa aplicação em Java, nos pusemos a esboçar quais classes seriam necessárias para a concretização do sistema.

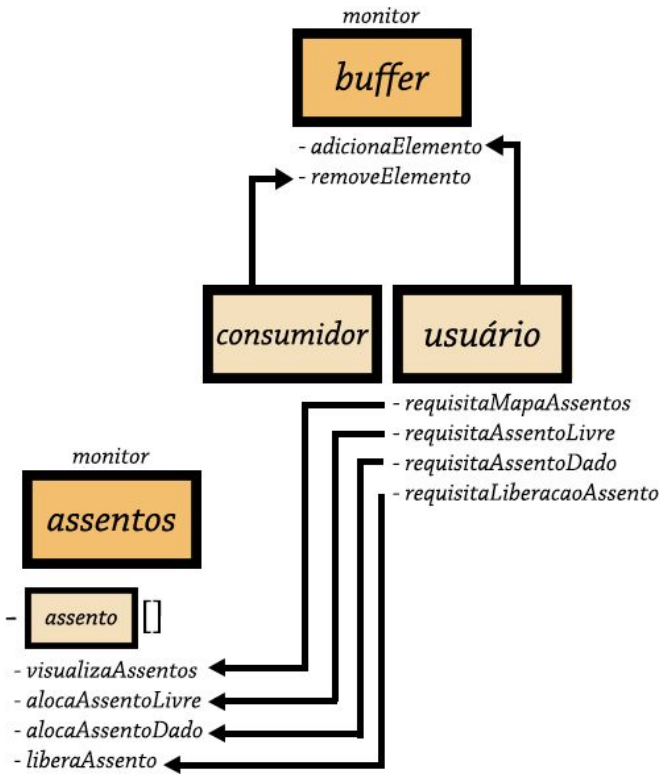
Os *Usuários* são claramente classes que estendem a classe *Thread*, visto que serão eles os responsáveis por realizar requisições simultâneas que a aplicação terá de gerenciar. Porém, como cada *Usuário* deve agir como um *Produtor* e inserir o registro de sua ação em um *Buffer*, concluímos que a classe *Usuário* teria de ter acesso ao *Buffer*, tal qual no padrão 'Produtor/Consumidor' de Java visto em sala. Também concluímos que, já que toda vez que o *Usuário* requisitasse algo ao sistema ele teria de registrar isso no *Buffer*, seria melhor que todas as ações do *Usuário* estivessem encapsuladas em métodos que também geram a *String* apropriada e a adicionam ao *Buffer*.

Cada *Usuário* agiria como *Produtor*, mas o *Consumidor* seria uma classe à parte de mesmo nome cuja única função seria retirar elementos do *Buffer* e adicioná-los a um arquivo de saída. Também percebemos de cara que haveria um problema: como o *Consumidor* saberia que é hora de parar de consumir, isto é, que os *Usuários* pararam de gerar *logs*? Se isto não fosse coberto pelo programa, ele executaria indefinidamente, o que não é algo que queremos. Portanto, eventualmente teríamos de incluir alguma espécie de sinalização para o *Consumidor* parar de esperar. Também nos foi evidente que o *Consumidor*, tal qual o *Usuário*, também teria de ter acesso ao *Buffer*.

O *Buffer* seria apenas uma classe simples do padrão 'Monitor' com um buffer de tamanho *U* (sendo *U* o número de *Usuários*) e inserção/remoção cíclicas em um vetor. Os métodos da classe garantem que os elementos serão retirados na mesma ordem em que forem postos, pois seria catastrófico para a verificação do *log* se ele exibisse linhas desordenadas, como por exemplo um assento que foi desalocado antes de ser alocado pela primeira vez.

Também seguindo o padrão 'Monitor', todas as requisições dos *Usuários* seriam repassadas a um monitor denominado *Assentos* responsável por gerenciá-las. Ele conteria toda a lógica concorrente de reserva de assentos, ou seja, os 4 métodos fundamentais (alocaAssentoLivre, alocaAssentoDado, visualizaAssentos, liberaAssento). Ele possuiria uma classe interna denominada *Assento*, de modo que cada *Assento* possui um *int* (o ID do *Usuário* que o alocou, ou 0 quando vazio) e um *boolean* (*true* se assento livre e *false* caso contrário). O mapa de assentos seria então um vetor de *Assento* dentro de *Assentos*, apenas acessível dentro da classe. Para facilitar, a classe *Assento* teria dois métodos - *aloca* e *desaloca* -, que nada fazem mais do que definir os valores internos de cada assento e fazer verificações básicas.

Quanto à validação do *log*, decidimos por deixar esta parte para depois, já que ela é por natureza completamente independente do resto da aplicação. Neste estágio, nosso mapa mental do código ficara da seguinte forma:



A partir desta esquemática, construímos uma versão inicial do nosso código, implementando os 4 métodos fundamentais (visualizaAssentos, alocaAssentoLivre, alocaAssentoDado e liberaAssento) utilizando uma lógica básica para cada um deles, sem muitas complicações. Também poupamos esforço na hora de simular usuários 'bobos' (que apenas alocam assentos livres ou requisitam visualização do mapa de assentos), apenas para checarmos se estava tudo nos conformes. Como esperávamos, alguns problemas surgiram.

2) Implementação de Leitores/Escritores

Neste instante, também notamos a semelhança entre a aplicação proposta e o padrão de leitores/escritores que vimos no curso. Apenas uma thread poderia alterar o mapa de assentos por vez, porém um número ilimitado de threads poderiam visualizar o mapa a qualquer momento. De certa forma, a lógica do nosso programa já estava trilhando este caminho sem termos percebido, porém acreditamos que seria mais confiável implementarmos um padrão que já havíamos testado em outras ocasiões e que tínhamos certeza de que funcionaria - especialmente quando a implementação não seria complicada. A transição seria simples: nenhum dos 4 métodos fundamentais seriam synchronized, porém os métodos de entraLeitor e saiLeitor seriam chamados no início e no final do visualizaAssentos e os métodos de entraEscritor e saiEscritor seriam chamados no início e no final das outras 3 funções.

A aplicação permaneceu correta, porém agora estávamos mais confiantes de que ele não estava escondendo erros de nós. Para termos certeza de que não haveria problemas ao fim do caminho, também adicionamos uma prioridade para os escritores para que não houvesse *starvation* nas threads *Usuário* que estivessem à espera de alocar ou desalocar assentos.

3) Lógica principal dos métodos fundamentais

A lógica principal que utilizamos nos quatro métodos fundamentais foi a seguinte:

1) *visualizaAssento*

Retorna o mapa de assentos atual, sem quaisquer complicações. Sendo uma atividade de leitura, está fechada entre entraLeitor e saiLeitor. Logo após de pegar o mapa de assentos, registra a ação no *log*.

2) *alocaAssentoLivre*

Em nossa aplicação, possuímos um ArrayList com os índices de todos os assentos livres. Caso este ArrayList esteja vazio, o método *alocaAssentoLivre* retorna fracasso. Caso contrário, ele aloca o primeiro assento livre e o retira do ArrayList, retornando sucesso e o índice do assento alocado.

3) *alocaAssentoDado*

Verifica se o índice do assento desejado é um índice válido (maior que 0, menor que o tamanho do mapa de assentos). Se for um índice inválido, retorna fracasso. Caso seja um índice válido porém este assento não esteja livre, também retorna fracasso. Caso contrário, este assento é efetivamente alocado e removido do ArrayList de assentos livres.

4) *liberaAssento*

Tal qual alocaAssentoDado, liberaAssento também verifica se o índice do assento desejado é um índice válido, e se não for, retorna fracasso. Caso seja um índice válido, existem duas possibilidades de fracasso: o assento está livre, ou ele foi alocado por outra thread. Nos dois casos, o método retorna fracasso. Caso contrário, o assento torna-se livre novamente e é adicionado ao ArrayList dos assentos livres.

4) Problema Inicial - Logs Irregulares

Mesmo com poucos usuários, pudemos notar um problema na geração do log: da forma que havíamos feito, apenas 3 dos métodos fundamentais estavam sendo mutuamente exclusos: alocaAssentoLivre, alocaAssentoDado e liberaAssento (os três tratados como métodos de 'escrita'). Porém, após o fim da execução de um desses 3 métodos, poderia haver livre concorrência entre as threads, tornando possível o cenário em que uma thread *Usuário* aloca um assento e perde sua fatia de tempo do processador antes de visualizar os assentos e armazenar o mapa no buffer. Isto é, a ausência de exclusão mútua entre uma alocação/desalocação e a visualização do mapa para armazenamento no buffer permitia com que uma thread armazenasse no log um mapa que não era necessariamente o mesmo que ela tinha criado assim que terminou. Portanto, teríamos que liberar a exclusão mútua somente depois de visualizarmos o mapa para armazená-lo no buffer.

Não somente isso, mas também teria de haver exclusão mútua na *inclusão do mapa no buffer*. Se não houvesse, poderia haver um cenário em que a thread *Usuário* pega corretamente o mapa de

assentos que ela deixou após provocar sua alteração, mas acaba armazenando ele no buffer depois de uma segunda thread que provocou sua respectiva alteração antes da primeira. Ou seja, a ordem das ações no log não seguiria necessariamente a mesma ordem de execução das ações, futuramente gerando um log válido porém completamente desordenado. Para consertar esse problema, acabamos liberando a exclusão mútua *depois* de termos incluído o registro daquela ação no buffer, gerando um código muito mais sequencial do que gostaríamos. Porém, consideramos que corretude seria mais importante do que desempenho.

5) Tipos diferentes de usuário

Até este momento, estávamos fazendo uso de threads *Usuário* com comportamentos estúpidos, dificilmente problemáticos. Para deixar as coisas mais interessantes (e tentar forçar o surgimento de erros), decidimos por criar comportamentos irregulares para as threads *Usuário*, fazendo uso completo da biblioteca *Random* do Java.

Resumidamente, cada thread *Usuário* realiza entre 1 e N iterações. Em uma iteração, o *Usuário* 'joga uma moeda' para ver se ele requisita a alocação de um assento livre, se ele requisita a alocação de um certo assento específico (determinado aleatoriamente, pode ser qualquer número entre 0 e o tamanho do mapa de assentos) ou se ele não faz nada. Em seguida, o *Usuário* passa por um pequeno delay e 'joga outra moeda' para decidir se vai requisitar visualizações ou não. Caso decida requisitar visualizações, ele 'joga um dado' para ver quantas visualizações ele requisita (de 0 a 2*N). Por fim, o *Usuário* desaloca o assento que ele alocou no início, e se ele não alocou assento algum, ele tenta desalocar um assento aleatório. Optamos por fazer ele com certeza desalocar o seu assento para que não gerássemos logs desinteressantes em que o mapa de assentos ficasse completamente cheio: o *Usuário* surge, aloca um assento e depois vai embora. Após a desalocação, o *Usuário* encerra a iteração e passa para a próxima (se houver). No fim de todas as iterações, ele sinaliza para o *UsuarioCreator*, como veremos posteriormente.

6) Validação do log

Para efetuar a validação do log, optamos por implementar um script sequencial em Python. A lógica seguida foi iterar por cada linha do log, observando se a operação registrada faz o que deveria no mapa e nada mais. O que uma operação deve fazer é determinado por ela própria, seu argumentos e o estado anterior do mapa de assentos. Para cada operação, os comportamentos esperados são os seguintes:

1) *visualizaAssentos*
Deve sempre gerar um mapa idêntico ao anterior.

2) *alocaAssentoLivre*
Se o índice do assento selecionado for -1, significa que não haviam lugares livres. Portanto, o mapa gerado deve ser idêntico ao anterior. Por outro lado, se o índice for um valor entre zero e o número de assentos, é preciso que o assento correspondente no mapa anterior seja zero e o valor no assento do mapa gerado deve conter a id da thread que solicitou a operação.

3) *alocaAssentoDado*
O índice do assento selecionado deve sempre ser um valor entre zero e o número de assentos. Se o assento correspondente no mapa anterior tiver valor diferente de zero, significa que já está ocupado. Portanto, o mapa gerado deve ser idêntico ao anterior. Em caso contrário, o valor do assento correspondente no mapa gerado deve ser o mesmo que à id da thread que solicitou a operação.

4) *liberaAssento*
Novamente, o índice do assento selecionado deve sempre ser um valor entre zero e o número de assentos. Se o assento correspondente no mapa anterior tiver valor diferente da id da thread que solicitou a operação, o mapa gerado deve ser idêntico ao anterior. Em contrapartida, se tiver o mesmo valor, o valor do assento correspondente no mapa gerado deve ser zero.

Qualquer comportamento diferente dos listados acima é inválido. Nos usamos então desse fato para identificar diversas possíveis inconsistências no log do programa principal.

7) Geração dinâmica de usuários

Com o validador do log em mãos e com certo grau de irregularidade no comportamento das threads *Usuário*, decidimos realizar testes com grandes volumes de usuários acessando o sistema simultaneamente. Porém, como nossas máquinas apenas possuíam poucos núcleos, não havia muita opção além de criar um número elevado de threads (na casa dos milhares). A primeira ideia que nos surgiu foi simplesmente aumentar o número de threads criadas no laço *for* da função *main*, como sempre o fizemos. Isto funcionou: se fizéssemos a *main* criar 10.000 threads, ela criaria 10.000 threads, embora eventualmente.

No entanto, não tínhamos conhecimento exatamente de como isso se comportaria em outras máquinas, portanto decidimos errar para o lado da cautela. Criamos uma classe denominada *UsuárioCreator* cujo objetivo é exatamente este: criar threads *Usuário*. Ela é inicializada na *main* recebendo como parâmetros para seu construtor, dentre outras coisas, dois ints: um que determina o número total de threads *Usuário* que devem ser criadas durante a vida de *UsuárioCreator* e outro que determina quantas threads *Usuário* devem ser criadas inicialmente pelo *UsuárioCreator*. Isto se deve ao fato de que, da forma como implementamos, o *UsuárioCreator* só cria *Usuários* quando recebe um sinal para o fazê-lo, de modo que ele só recebe este sinal quando a aplicação inicia ou quando uma thread *Usuário* termina. Ou seja, quando um *Usuário* termina, ele pede para que *UsuárioCreator* crie um novo *Usuário*. Quando o número de *Usuários* atinge o limite, o *UsuárioCreator* não aceita novas requisições de criação de *Usuários*, e espera todas as threads remanescentes se encerrarem para sinalizar ao *Consumidor* de que ele pode finalizar seu trabalho e imprimir o *log*. Isto pode ter sido desnecessário, porém acreditamos que um maior grau de controle para o usuário da aplicação não poderia ser ruim - tendo em vista que ele pode controlar quantas threads *Usuário* estão rodando de cada vez alterando o número de threads *Usuário* que são criadas no momento inicial.

8) Corretude final da aplicação

Tendo toda a aplicação construída, modificamos diversas partes do programa para detectar se o validador estava funcionando. Trocamos linhas de código de lugar, replicando os empecilhos que encontramos no Problema 1 para gerar logs irregulares, que claramente foram detectados pelo validador. Testamos também com buffers gigantescos e minúsculos, com muitas threads de uma vez ou com poucas threads de uma vez, com limites baixos e limites altos. O validador por vezes acusou problemas, porém consertamos os erros e tornamos a repetir os testes, até que ele cessasse suas reclamações.

Também alimentamos ele com logs pequenos, feitos à mão, para termos certeza de que ele estava detectando alguns erros perigosos. Porém, quando depois de muito esforço nos pareceu que ele simplesmente não estava mais detectando erros na nossa aplicação, decidimos desistir e supôr que nosso programa estava - pelo menos em grande parte - gerando os logs corretos.