**RELATÓRIO E DOCUMENTAÇÃO DO SOFTWARE PARA VISUALIZAÇÃO E RESERVA DE ASSENTOS**

George Rappel M. Da Conceição

Gabriel Felipe Vargas Ferreira

Pedro Paulo Kleiz

1. **INTRODUÇÃO**

Um sistema de reserva de assentos consiste em um programa que mantém informações sobre a alocação de assentos em um espaço físico ou em um meio de transporte, permitindo que usuários remotos possam ver o estado das alocações, solicitar nova alocação ou cancelar uma alocação realizada previamente pelo próprio usuário. Normalmente esse tipo de sistema é executado em um ambiente distribuído com duas aplicações distintas: uma que executa o código que gerencia as estruturas de dados usadas para manter o estado de alocação dos assentos e outra que é executada em máquinas remotas pelos usuários em essência, trata-se de uma aplicação concorrente com vários fluxos de execução independentes.

1. **Linguagem de Implementação**

A linguagem de programação escolhida para a realização do sistema tanto quanto o do programa auxiliar de conferência do log foi Java.

1. **ESTRUTURAS USADAS**

**ConcurrentMap**: Usado para armazenar o mapa de assentos do sistema, é um tipo de HashMap com acessos concorrentes gerenciados pela própria estrutura. Usando o método replace, atualizamos um valor no hashmap caso o valor corrente seja o valor especificado como parâmetro. Como a estrutura já tem concorrência nativamente, funciona corretamente mesmo que duas threads tentem trocar o valor para uma dada chave do hashmap ao mesmo tempo.

O método replace recebe como parâmetros a chave (key, ou index) a ser alterada, o valor condicional para ser substituído, e o novo valor. A substituição do valor no mapa só ocorrerá se for igual ao valor passado no parâmetro.

Exemplo: mConcurrentMap.replace(1, 0, 9)

Altera a posição 1 do hashmap para 9 somente se o valor naquela posição for 0. Retorna true se conseguir, retorna false se não conseguir alterar.

**ConcurrentLinkedQueue**: Fila usada para armazenar e organizar os textos que serão enviados para o buffer escrever. Desta forma, usamos os métodos queue.remove() para pegar a próxima linha do buffer, e queue.add(linha) para adicionar uma linha do log no final da fila.

**BufferedWriter**: Buffer de escrita usado para a criação do log de ações do sistema.

**ReentrantLock**: Usado para bloquear o acesso à fila de buffer do log, mantendo ordenada a inserção no log.

**Random**: Variável global randômica necessária para a ação de reservar um assento livre aleatório.

1. **CLASSES E MÉTODOS**

**Main:**   
Recebe os parâmetros referentes ao nome do arquivo que será escrito o log junto da quantidade de assentos e então seta os valores de todos os assentos para 0, indicando que estão livres. Instância todas as Threads do sistema, seta os nomes delas para usar de Id e então dispara todas e espera a finalização de cada uma usando join.

**Thread1**

Gerencia 3 assentos. Repete 3 vezes os passos:

Gera um primeiro assento aleatório.

Aloca três assentos consecutivos a partir do primeiro gerado a partir de um primeiro assento aleatório.

Desaloca os três assentos alocados no passo anterior.

Pausa por 1 segundo e em seguida desaloca os 3 assentos.

**Thread2**

Thread que aloca um assento aleatório livre, e em seguida desaloca o mesmo.

**Thread3**

Thread 3 aloca dois assentos: um livre e um fixo, que é passado como parâmetro no construtor.

**T\_Log**

Thread que inicializa o Buffer, escreve nele os valores da variável global textoBuffer e então ao final da execução de todas as Threads, finaliza o Buffer.

**VisualizaAssentos** ()

Visualiza os assentos ocupados e os assentos não ocupados, listando-os com seus respectivos status.

**alocaAssentosDado** (**int** assento, **int** id)

Passados dois parâmetros, alocam-se o número dado pelo thread id

**int** assento é o número do assento a ser alocado

**int** id é o id do thread que está tentando alocar o assento.

**alocaAssentoLivre** (**int** id)

**int** id é o id do thread que está tentando alocar o assento.

O método aloca um assento aleatório disponível seguindo os seguintes passos:

1 - Faz um loop por no máximo 5 vezes onde ele gera um número aleatório e tenta alocar esse lugar.

2 - Caso não consiga, faz um loop por todos os assentos, do primeiro até o último, buscando um assento vazio.

3 - Caso encontre um assento vazio, o aloca. Se não, retorna.

@param id ID da Thread que chamou a função.

@return 0, se não conseguir reservar um assento, ou o número do assento alocado caso consiga.

**liberaAssento** (**int** assento, **int** id)

Libera um assento dado se o id for igual ao id do thread que o alocou.

@param assento Número do assento a ser desalocado

@param ID do thread que alocou o assento

@return 0 - se o assento não for desalocado. return 1 - se o assento for desalocado.

**fazString** ()

Transforma o mapa de assentos em uma String para ser usado no log.

**inicializaBuffer** ()

abre o arquivo de texto para escrita

**buffer** (**int** codigo, **int** id\_thread, **int** assento)

Envia para a variável global textoBuffer a operação realizada com a lista nova de assentos para ser escrito pela Thread do log.

**finalizaBuffer** ()

fecha o arquivo no final da execução.

**AvaliaLog.avaliar** (**String** nome\_arquivo)

Método que faz a conferência do nome\_arquivo de log, garantindo que as operações foram feitas corretamente, na ordem, e mantiveram a consistência dos dados anteriores através dos múltiplos threads.

**AvaliaLog.Linha**

Classe que representa uma linha única do log do programa, possuindo como atributos int operacao, int id\_thread, int assento e HashMap mapa.

operacao - id da operação realizada.

id\_thread - id da thread que realizou a operação.

assento - assento modificado pela operação. É 0 caso a operação seja de visualizar os assentos.

mapa - mapa de assentos no momento.

**AvaliaLog.Linha.comparaMudancasRetornaCorreta (Linha** proxLinha**)**

Compara a linha atual (this) com a próxima, simulando a operação com o id do thread envolvido.

Retorna null caso as linhas sejam incompatíveis, retorna proxLinha caso tudo esteja correto.

1. **PROCESSO DE DESENVOLVIMENTO DO SOFTWARE**
   1. **PRÉ-DESENVOLVIMENTO**

Como solicitado pela Professora Flávia Delicato, na primeira semana projetamos a ideia inicial do projeto, incluindo alguns métodos importantes, propostas de formas de sincronização das variáveis, variáveis globais, entre outros detalhes.

Nossa proposta foi modificada completamente à partir da nossa primeira ideia. Em nossa pesquisa, encontramos classes nativas mais simples, e seguras em java para manter a consistẽncia dos dados, como o ConcurrentHashMap, em detrimento do array de inteiros que havíamos proposto em primeira instância, e não poderia ser sincronizado.

Similarmente, o buffer do log sofreu grandes modificações, principalmente após a finalização da etapa de alocação e desalocação dos assentos (ver parte 6.3).

* 1. **ESCOLHAS DAS ESTRUTURAS**

Para armazenar o mapa de assentos, optamos pelo ConcurrentHashMap. Ele apresenta todas as características comuns do HashMap: utiliza um objeto chave/índice e um objeto de valor. Além disso, ele tem técnicas implementadas nativas de sincronização e consistência de dados através de múltiplos threads.

O método mais utilizado foi o replace, que permite substituir um valor apenas caso o valor atual seja igual ao valor especificado como parâmetro. Assim, não é necessário fazer um teste de valor antes (Ex: if(map.get(“chave”) == 0) map.put(“chave”, 1)), mas sim tentar fazer a alteração direto e garantir que o valor não foi trocado em outro thread entre o teste e a atualização.

Para armazenar o log, utilizamos uma ConcurrentLinkedQueue, ou seja, Fila Encadeada Concorrente. Similar ao concurrent hash map, essa classe apresenta formas de sincronização nativas. O principal é poder incluir elementos que devem ser escritos no log sempre no final da fila, e escrever no log sempre removendo o próximo elemento.

* 1. **CONFERÊNCIA DO LOG**

O processo de finalização do projeto apresentou novos erros e complexidades que não foram notados durante o desenvolvimento da maior parte do programa.

Após a implementação da rotina de conferência do Log, pudemos notar que alguns passos estavam sendo sobrepostos e o log apresentava uma ordem incorreta das operações, apesar de ainda exibir no final todos os assentos novamente desalocados, como previsto.

Notamos que, apesar de não haver problemas funcionais nas operações, alocações e desalocações, o acesso de leitura do mapa de assentos retornava, por vezes, valores anteriores. Também era possível ter valores alterados enquanto os valores do mapa eram lidos.

Para resolver este problema, foi utilizado um semáforo binário (que atua como um Mutex). Este semáforo bloqueia os acessos ao mapa e permite que apenas uma operação seja realizada por vez. A implementação deste recurso, de certa forma, inutiliza as vantagens de usar um ConcurrentHashMap, mas foi a solução encontrada mais confiável de tanto ler quanto escrever dados fielmente.

A implementação do semáforo também permitiu a remoção de todos os **synchronized** em todos os métodos de acesso às variáveis, deixando o software totalmente dependente do ConcurrentHashMap e do Semáforo implementado.

1. **EXEMPLOS DE TESTES REALIZADOS**

Fizemos o teste de execução do programa usando a entrada com nome do arquivo como “teste.txt” e sendo 10 a quantidade de assentos escolhida. Para melhor visualização e entendimento do funcionamento dele, setamos uma print dizendo onde começa e inicia o programa, além de informar as ações das threads usuários, dizendo se está alocando, liberando ou visualizando os assentos, além de mostrar qual o assento e o mapa de assentos quando necessário. Após isso, temos 2 prints, informando a inicialização do programa auxiliar que irá avaliar o log feito pela execução do programa.  
