UFRJ

2º TRABALHO DE COMPUTAÇÃO CONCORRENTE

Professora: Silvana Rossetto

Alunos:

- Ingrid Quintanilha Pacheco (DRE: 115149161)
- Júlio Rama Krsna Mandoju (DRE: 115023797)

SUMÁRIO

Ideia para a Implementação do Programa

O programa pedido era um sistema de reserva de assentos. Para isso, pensamos em algumas estratégias que nos auxiliaram na construção da solução: **Leitor/Escritor** e **Produtor/Consumidor**.

- Leitor/Escritor: Ao construirmos o programa, percebemos que alguns métodos da classe t_Assentos só liam o mapa de assentos, enquanto outros o alteravam. Por isso, criamos um leitor/escritor e tiramos a sincronização desses métodos. Dessa forma, o acesso aos métodos se torna mais dinâmico (mais de uma thread acessando), mas o acesso aos assentos continua sendo particular, o que otimiza e torna menos sequencial o programa. - Utilizamos um monitor LE.
- Produtor/Consumidor: A saída do programa (log) é para ser gravada em um arquivo de texto, com todas as ações executadas pelas diversas threads. Por isso, pensamos em criar um *Buffer, no qual cada thread produtora (que executa alguma ação sobre o banco de assentos), ao desempenhar a sua função, escreve-a nele, e outra consumidora, que pega essa String no Buffer e escreve no arquivo de saída, fazendo com que dessa forma, as ações sejam escritas na ordem correta e que não percamos nenhum dado executado. - Utilizamos um monitor Buffer.

Além dessas duas classes importantes (Buffer e LE), foram implementadas t_Assentos e t_Assento, para controlar os assentos.

Era necessário criar uma classe para o controle das ações nos assentos, e para isso, foi criado t_Assentos. Nela, possuímos o *ArrayList de assentos e todas as ações possíveis sobre os assentos, como a visualização deles, alocação direta ou aleatória e desalocação.
OBS: É dentro dessas funções que é colocado em prática o Leitor/Escritor, para os diferentes métodos de acesso ao banco.

Por último, também foi criado t_Assento, que serve mais especificamente para o controle dos atributos de cada assento.
Para uma ação ser executada em t_Assentos, é necessário o id de determinado assento e outros atributos, então esse acesso é feito através de métodos da classe t_Assento, assim como se quisermos alterá-lo, é necessário chamar uma função específica para que o dado seja alterado corretamente.

* Será visto mais minuciosamente na parte de estruturas de dados usadas

Implementação nos Programas

A implementação se deu nas ideias apresentadas acima. Então, de uma forma sucinta, explicaremos os processos utilizados:

- **Auxiliar:** O programa auxiliar serve para verificar a corretude do programa principal, ou seja, verificar se a resposta está correta.

Para isso, criamos uma classe Assento, que tem a mesma função dela no programa principal, e uma main.

Classe Assento

- **getPosition** - Retorna a posição do assento.
- **getThreadId** - Retorna id da thread que a alocou.
- **viewSeat** - Se o assento está reservado retorna o id da thread que o reservou, e se não, retorna 0.
- **setOwner** - Se já está reservado retorna 'false', mas se não, reserva para a thread específica e retorna 'true', que o assento foi reservado com sucesso.
- **freeSeat** - Se está reservado e se a thread que chamar foi a mesma que reservou, libera o assento.
- **equals** - Se a classe do objeto não for Assento retorna falso, caso contrário a comparação é feita na posição do objeto e na thread que o reservou. Se forem os mesmos, retorna true. Esse método, assim como o toString, é um override de funções que já existem, mas que preferimos adptar para serem usadas da nossa forma.
- **toString** - Cria a formatação de cada linha identifica a passada para o arquivo de texto no programa principal.

Classe Principal

- **visualiza -** Imprime na tela que a thread específica visualizou o mapa de assentos.
- **erro -** Função que imprime o erro e dá um exit em seguida.
- **insereRand -** Se o número do assento for menor do que zero, ele não existe, logo retorna o próprio array. Caso contrário, se tentar alocar um que já está reservado, é chamada a função erro, e se for possível alocar, setamos o determinado assento com a threadId que chamou a função, assim, reservando-o para essa thread.

- **insereDado -** Se o número do assento não estiver dentro do limite do Array, ou seja, se for menor do que zero ou maior do que o tamanho, se não tiver um assento na posição especificada ou se o assento está alocado, ela só retorna o array normal, sem nenhuma modificação. Caso contrário, setamos o determinado assento com a threadId que chamou a função, assim, reservando-o para essa thread.
- **remove -** Se o número do assento não estiver dentro do limite do Array, ou seja, se for menor do que zero ou maior do que o tamanho, se não tiver um assento na posição especificada ou se a thread que o alocou é diferente da que está executando a função, ela só retorna o array normal, sem nenhuma modificação. Porém, se não for esse o caso, setamos o determinado assento com nenhuma ThreadId, pois é como se tivesse sido liberado novamente.
- **escolheFuncao -** Recebe como um dos parâmetros o número da função e faz um switch case, no qual dependendo dele, vai para diversas funções diferentes.

- **Principal:** Para o programa principal criamos 5 threads produtoras e uma consumidora. Além deles, criamos as classes mencionadas acima, uma global e a main.

Threads Produtoras - As threads produtoras mandam executar as ações nos assentos. Existem 5 tipos diferentes delas e cada uma executa o seu conjunto de ações específico.

Thread Consumidora - A thread consumidora retira elementos do Buffer e os grava no arquivo de saída (log).

Classe Buffer

- **insere(Buffer)** - Quando uma thread produtora executa uma função, dentro dela é chamado o método

insere. Nele, a função passa uma String com o id da thread, e todas as informações relevantes. Assim, o método insere tenta inserir no Buffer aquele registro de ação, e se tiver cheio, espera que algum outro seja retirado pelo consumidor, para que possa ser alocado corretamente. Se ela conseguir inserir, envia um "sinal" de que a consumidora pode consumir um dado.

- **retira(Buffer)** - Chamada pela thread consumidora, essa função remove do Buffer o elemento "mais antigo" e o retorna para a thread consumidora. Caso o Buffer esteja vazio, a thread espera que algo seja inserido, para poder continuar. Se remover o último elemento, a thread produtora, caso esteja congelada, recebe um sinal indicando que ela pode inserir um novo elemento.
- **imprime** - Imprime o buffer naquele instante.

Classe LE

- **entraEscritor** - Chamado pelos métodos que alteram os assentos. Ao entrar nessa função, é verificado se já não há alguém alterando ou lendo os assentos, e se não há, então é liberado o acesso para que sejam feitas as devidas mudanças.
- **saiEscritor** - Ao sair do escritor, a permissão é concedida para que outra função possa mexer ou visualizar os assentos.
- **entraLeitor** - Chamado pelos métodos que visualizam os assentos. Ao entrar nessa função, é verificado se já não há alguém alterando, e se não há, é liberado para que os assentos sejam lidos.
- **saiLeitor** - Ao sair, verifica se é o último leitor, para que nenhuma alteração seja feita enquanto um deles lê, e se for, concede a permissão para que outra função possa mexer ou visualizar os assentos.

Classe t_Assento

- **getPosition** - Retorna a posição do assento.
- **getThreadId** - Retorna id da thread que a alocou.
- **viewSeat** - Se o assento está reservado retorna o id da thread que o reservou, e se não, retorna 0.
- **setOwner** - Se já está reservado retorna 'false', mas se não, reserva para a thread específica e retorna 'true', que o assento foi reservado com sucesso.
- **freeSeat** - Se está reservado e se a thread que chamar foi a mesma que reservou, libera o assento.

Classe t_Assentos

- **retornaMapa -** Através de uma StringBuilder, constrói uma String contendo as informações sobre cada assento (a thread que a está alocando, e se não tiver, põe 0 no lugar) e a retorna.
- **visualizaAssentos -** Utilizando uma StringBuilder, cria uma String contendo o registro da ação executada seguindo o modelo proposto no pdf (número da ação, id da thread, o mapa de assentos. Essa é uma função que entra no leitor para chamar a função retornaMapa e visualizar o mapa de assentos, e insere uma compilação de todos esses dados transformado em String (pela StringBuilder) e envia pro Buffer.
- **alocaAssentoLivre -** É uma função que faz alteração nos assentos, então precisa entrar no escritor. Quando criamos o arrayList de assentos, criamos também um de disponíveis e de reservados. Dessa forma, todos os assentos disponíveis estão nesse arrayList. De forma randômica é escolhido um número e pegado o assento disponível naquela posição. São setados os atributos para que o assento seja reservado, ele é adicionado no ArrayList dos reservados e removido dos disponíveis. Durante o processo, através da StringBuilder é criada a String a ser passada para o Buffer, e ao término, a função são do escritor.

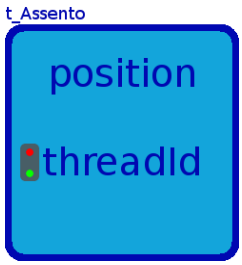
- **alocaAssentoDado -** Outra função que entra no escritor. Desta vez, o assento é passado como parâmetro e é verificado se ele já está reservado, através da função setOwner, e por ela mesma, se não estiver, é reservada para a thread específica. O assento é removido dos disponíveis e adicionado nos reservados. Da mesma forma como no caso acima, a StringBuilder cria a String a ser passada para o Buffer, e ao final da função, sai do escritor.
- **liberaAssento -** A função entra no escritor e verifica, através da função freeSeat, se pode liberar o assento (se ele foi reservado por tal thread), e se puder, retira do ArrayList dos reservados e insere nos disponíveis. Através da StringBuilder, cria a String e manda para o Buffer. Ao final, sai do escritor.
- **pegaAssento -** Pega um assento específico baseado num valor dado.

Estruturas de Dados Usadas

1) t_Assento:

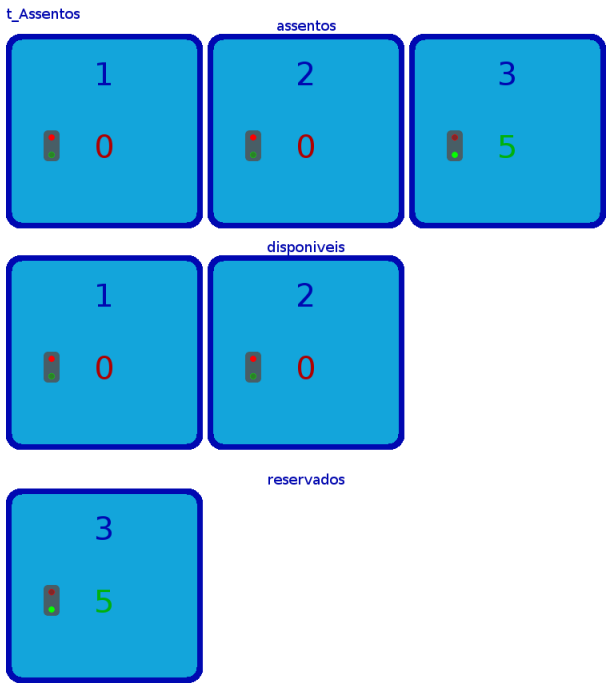
Cada assento é representado como uma instância da classe t_Assento. Esse objeto contém a posição do assento no mapa de

assentos, uma variável de estado que informa se o assento está ou não reservado e o id da thread que o reservou. Vale ressaltar que a função viewSeat retorna o id da thread que reservou o assento apenas quando ele está reservado; quando ele está liberado, esta retorna 0. Isso facilita, em especial, para a liberação desse assento pela thread que o reservou.



No programa auxiliar, uma estrutura muito semelhante foi feita, para que pudéssemos simular a criação do mapa através de ArrayLists.

2) t_Assentos:

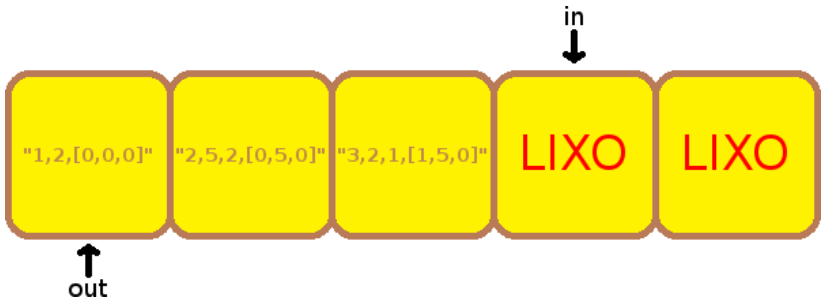


Dentro da classe t_Assentos temos três tipos de ArrayList, os assentos, disponíveis e reservados. Essa separação foi necessária principalmente para a alocação dos assentos. Dessa forma, seria muito mais prático escolher um assento disponível aleatoriamente, visto que não seria necessário checar a disponibilidade de tal, o que otimizaria o código.

Vale notar que, sempre que um assento é modificado em um dos ArrayLists menores (disponíveis ou reservados), o mesmo objeto é modificado no ArrayList que abrange ambos.

3) Buffer:

Para compartilharmos informações entre as threads produtoras e a thread consumidora, utilizamos um vetor global de Strings chamado “Buffer”. Esse vetor global tem tamanho N (definido como 50) e é acessado por todas as threads. Cada vez que uma ação sobre os assentos é executada, o produtor coloca no Buffer uma String contendo os dados referentes a essa execução na próxima posição vazia. Enquanto isso, a thread consumidora remove uma String do Buffer na mesma ordem em que elas foram inseridas. As Strings não podem ser inseridas enquanto o Buffer está cheio, nem retiradas enquanto estiver vazio.



As posições chamadas de “lixo” seriam as Strings que já foram removidas pelo consumidor e, portanto, não são mais relevantes.

Tentativa e Erro

Ao longo da implementação do algoritmo nos deparamos com alguns erros, e achamos necessário citá-los para que seja entendido o que levou o grupo a chegar ao resultado apresentado.

★ **DeadLock:** Um dos erros mais comuns em computação concorrente e sincronização. Como foram utilizados dois monitores, LE e Buffer, um deadlock é bastante plausível, visto que utilizamos wait dentro dele. Além disso, um notify ao invés de notifyAll poderia causar esse erro também.

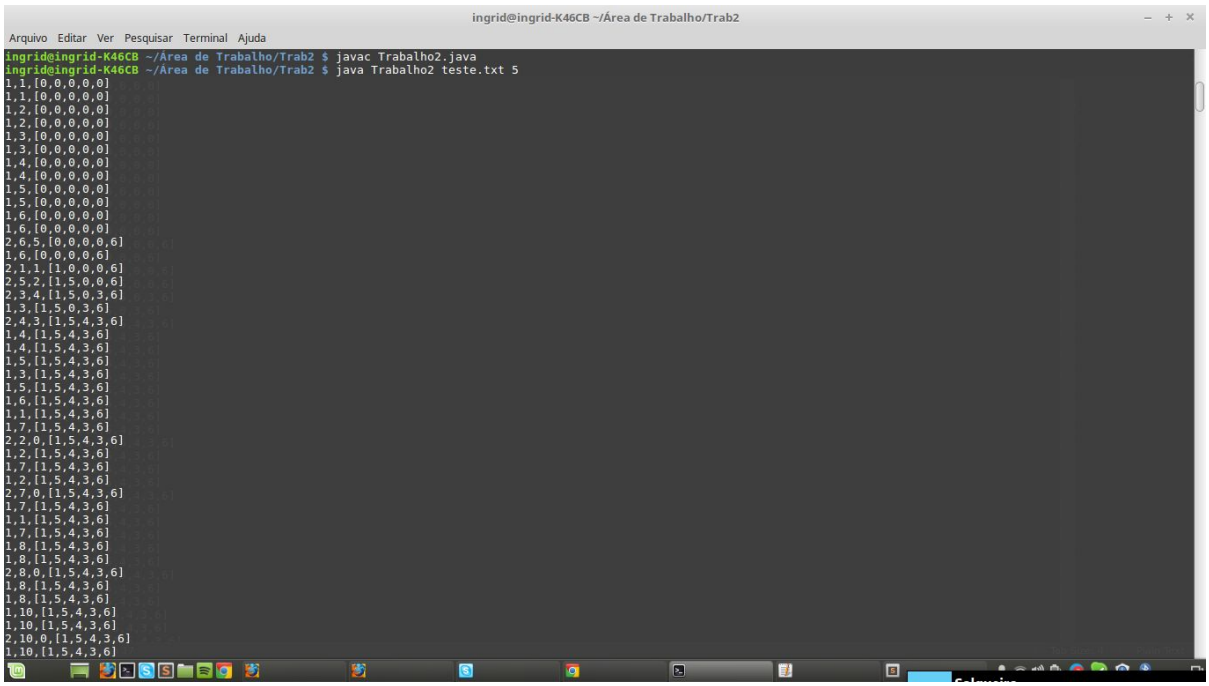
★ **ArrayIndexOutOfBoundsException:** Esse erro foi visto principalmente no programa auxiliar. Como estamos sempre acessando ArrayLists, havia casos em que tentávamos acessar uma posição negativa acidentalmente. Isso se deu graças a falhas na hora de considerar a as listas com indexação 1+ e considerando a posição 0 como uma posição de erro.

★ **Starvation:** Quando pensamos em utilizar a lógica do Leitor/Escritor, percebemos que seria possível o programa entrar em starvation, ou seja, o leitor lendo e o escritor esperando por muito tempo. Para isso, demos prioridade para o escritor, o que resolveu o problema.

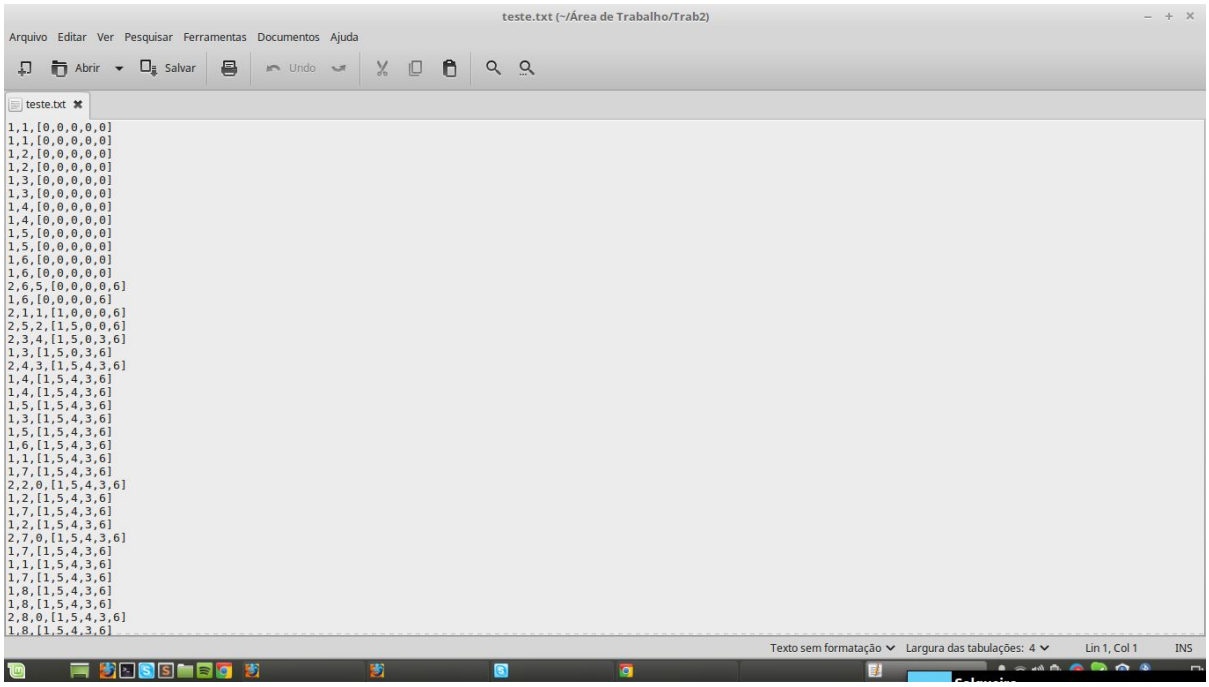
★ **LiveLock:** Ao implementarmos o programa, não sabíamos quais condições utilizar para fazer com que as threads parassem de funcionar, então deixamos um while(1) nelas, o que fazia com que não parasse e entrasse em estado de LiveLock.

Saídas do Programa

No programa principal temos a saída requisitada (impressão num arquivo de texto) e em tela:

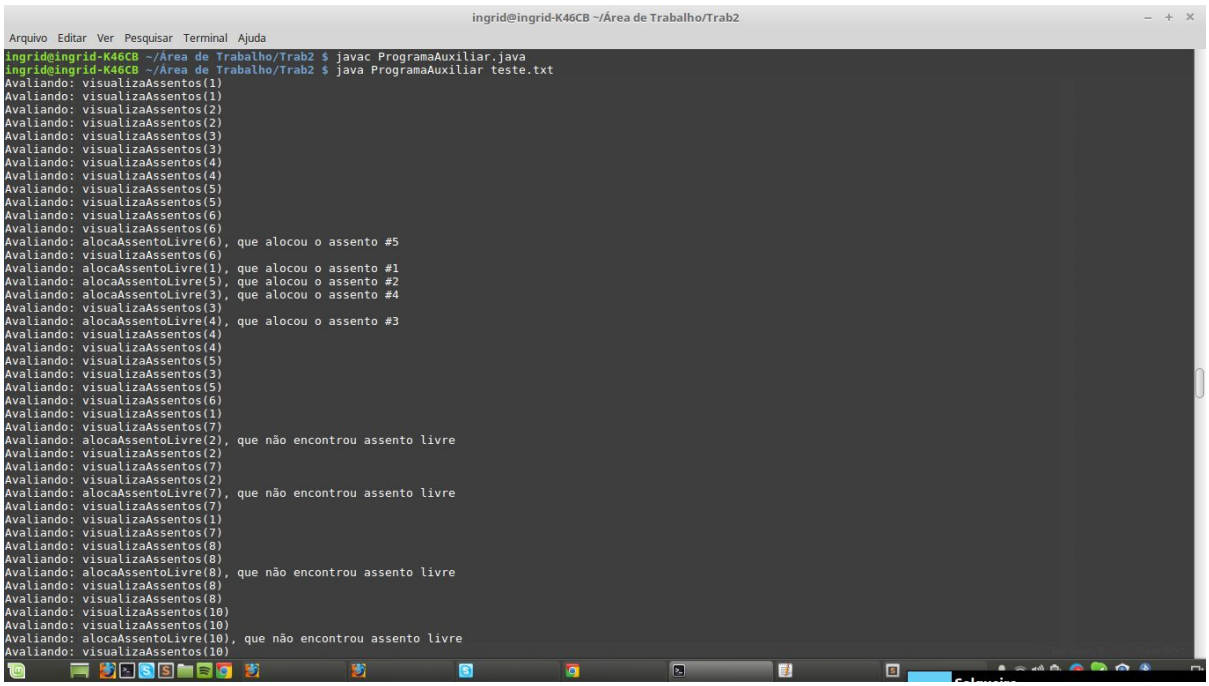


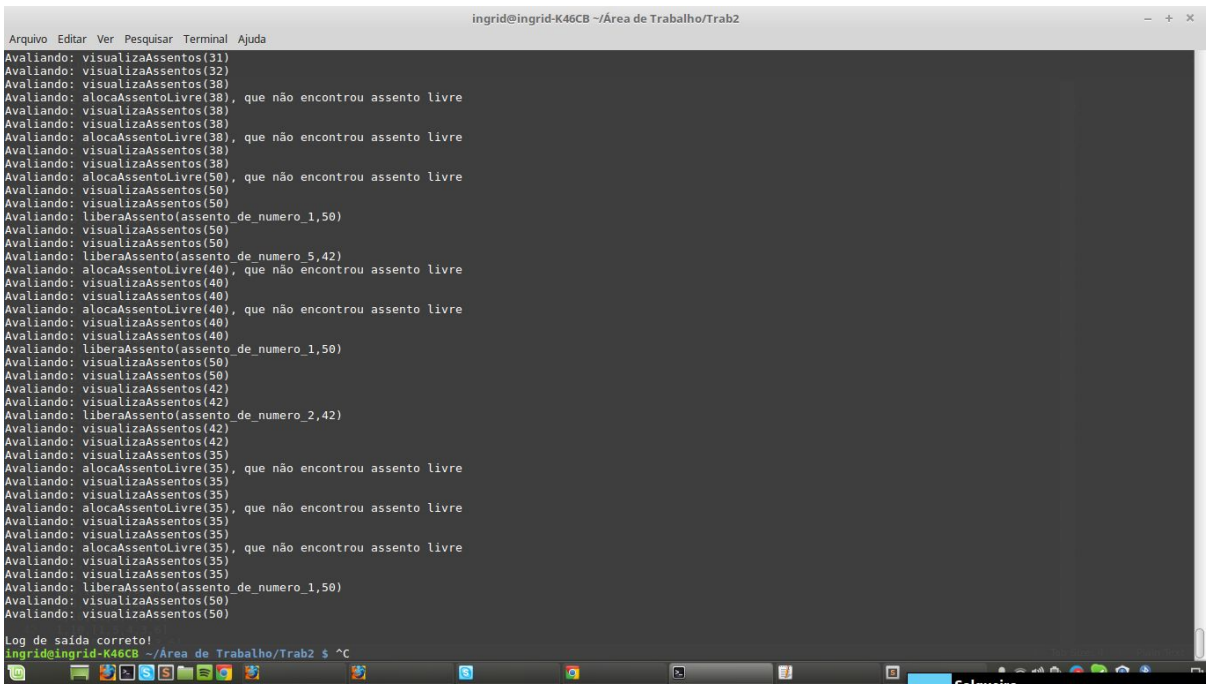
Saída do programa principal impressa em tela (recebe arquivo de log e número de assentos como argumentos).



Saída do programa principal impressa em arquivo de texto.

Já no programa auxiliar, temos que a saída será no próprio terminal, acusando cada ação que foi executada sobre os assentos, e no final, se o resultado está correto.





Bibliografia

Baseado nas aulas, exercícios propostos e material didático fornecido no site.