3 componentes por grupo

Considerações:
Observe os requisitos funcionais abaixo. **O grupo pode e deve sempre que achar necessário incrementar com mais funcionalidades.** Aplique os métodos de concorrência que achar necessário:

Questão 01

- **A)** Deve-se criar uma aplicação que modele o funcionamento de sua **conta bancária**. Esta conta será o *recurso compartilhado* e deverá ser acessada/modificada por três threads: o thread **AEsperta**, o thread **AEconomica** e o thread **AGastadora**, concorrentemente.

    - **1. AGastadora:** este thread (de atitude voraz) deverá, a cada 3000 milesegundos verificar se há saldo suficiente, e retirar 10 reais da sua conta. Este thread deve disputar por dinheiro, com os demais threads, concorrentemente.
    - **2. AEsperta:** este thread será mais comedido que a anterior: somente a cada 6000 milesegundos, irá verificar o seu saldo. Mas não se engane: se houver saldo suficiente, este thread irá retirar 50 reais da sua conta. Este thread deve disputar com outros threads, concorrentemente.
    - **3. AEconomica:** de todos as threads, esta será a que mais prezará por você e suas finanças. Ela irá verificar o saldo de sua conta apenas a cada 12000 milesegundos. Se houver fundos, a thread econômica irá tentar retirar apenas 5 reais da sua conta. Este thread deve disputar com outros threads, concorrentemente.

- **B)** A classe **Conta** (*recurso compartilhado*) deverá ter pelo menos:
    - Os seguintes atributos: **número** da conta, **titular** da conta e **saldo**.
    - Defina, **construtores**, métodos **get/set** para cada um dos atributos e um método **toString**.
    - Quando necessário, implemente também algumas regras básicas de integridade (número de conta negativos, etc.).
    - Finalmente, implemente os principais métodos que vão manipular o saldo da conta: **deposito** e **saque**. Inicie o saldo (recurso compartilhado) da conta depositando uma quantia de **R$ 1.000,00**.

    - DICA: Faça um esboço de um diagrama inicial de classes para lhe ajudar a pensar melhor antes de ir diretamente para o código. O diagrama ajuda você a visualizar quantas classes precisará e como elas se relacionam.

    - IMPORTANTE: Sempre que um thread movimentar fundos da sua conta, o sistema deve informar não apenas qual thread efetuou a operação (saque ou depósito), mas, principalmente, qual é o seu saldo atual final (após o saque). Isso permitirá que você acompanhe a situação financeira em tempo real.

    - LEMBRE-SE: Nesta aplicação, não defina prioridades (todos devem ter as mesmas chances). Além disso, não permita que haja corrupção de dados (ou seja, cada thread deve conseguir retirar sua quantia sem ser interrompido por outro thread materialista). Use *synchronized* ou outro modelo, por exemplo.

- **C)** Quando a conta estiver com **saldo zero**, todos os threads deverão ser colocados em estado de espera. Ao ser colocado em espera, cada thread deverá imprimir a quantidade de saques efetuados e o valor total retirado da conta. Execute e veja o resultado

- **D)** Acrescente agora mais uma thread de nome **APatrocinadora**. Este thread deverá depositar 100 reais sempre que a conta estiver zerada. Veja que este thread será “produtora”. E as demais serão “consumidoras”. (necessário usar wait e notifyAll ou outro modelo, por exemplo)

- **E)** Fica a critério do grupo se vão ou não usar GUI.

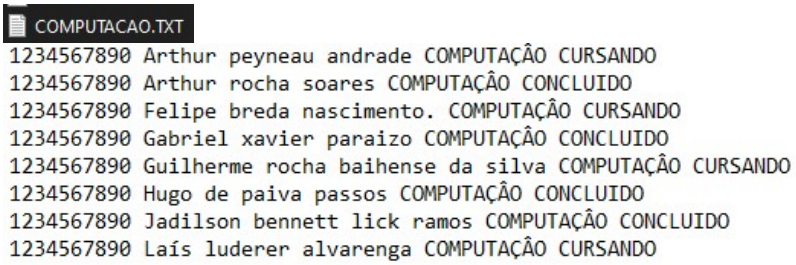
Questão 02

Uma Universidade precisa saber quais são os alunos que estão se formando no semestre para enviar estas informações para a empresa de eventos e poder organizar a festa de formatura e a impressão dos diplomas.

Para isso o setor de Tecnologia da Informação da Universidade preparou um arquivo (no formato "*.txt*") para cada curso de graduação contendo os dados de todos os alunos do curso. A Universidade tem 15 cursos de graduação. No arquivo existe uma *flag* indicando a conclusão de curso com o valor "CONCLUIDO".

Cada arquivo base para processamento possui as informações: {matrícula, nome do aluno, curso, *flag*}. A *flag* pode ser definida como CURSANDO ou CONCLUÍDO.

Segue um exemplo do arquivo gerado:



Faça um programa com o uso de programação concorrente para listar todos os alunos formandos (*status* da *flag* como CONCLUÍDO) a partir de uma busca em todos os arquivos dos cursos de graduação.

Questão 03

O cinema do Shopping ABC está contratando profissionais de TI para desenvolver uma "solução concorrente" no processo de pedidos de lanches no cinema. Basicamente o serviço oferece pipoca e refrigerante que funciona da seguinte forma:

1. No balcão, você pede uma pipoca e um refrigerante;
2. A atendente pega o pedido e solicita a execução para a sua equipe. Um membro da equipe prepara a pipoca e o outro membro prepara o refrigerante. Basicamente, ambas as tarefas acontecem simultaneamente. O seu pedido só é entregue, e naturalmente é considerado completamente realizado, somente quando ambas as tarefas são finalizadas;
3. Ambos os membros da equipe entre os pedidos para o atendente;
4. Agora você pode receber e se deliciar com o seu lanche.

Dica: Utilizando a API "*CompletableFuture*" podemos implementar um método "*getPipoca()*" que retorna um "future" com a *string* "Pipoca Pronta", um método semelhante, "*getRefrigerante()*", que retorna um "future" com a *string* "Refrigerante Pronto" e um método simples "*lanchePronto()*" que retorna uma *string* informando que o lanche já está pronto e que só deve ser chamado depois que a pipoca e o refrigerante já estiverem disponíveis.

Importante reforçar: tanto "*getPipoca()*" quanto "*getRefrigerante()*" são executados de forma concorrente. Uma vez que ambos tenham completado, as informações são retornadas o que permite a execução do "*lanchePronto()*"

Questão 04

Um problema simples que pode ser resolvido com threads é o cálculo da soma de um vetor grande de números inteiros.

Suponha que temos um vetor de 1 milhão de números inteiros e queremos calcular a soma de todos eles. Podemos resolver esse problema de forma sequencial, percorrendo o vetor e somando cada número, mas isso pode ser muito lento. Uma abordagem mais eficiente é dividir o vetor em várias partes e calcular a soma de cada parte em uma thread separada, para aproveitar o poder de processamento de máquinas com vários núcleos.

Para isso, podemos criar uma classe SomaThread que implementa a interface Runnable e recebe como parâmetros o vetor de números inteiros, o índice do início e do fim da parte do vetor que será somada, e um objeto AtomicInteger para armazenar o resultado parcial da soma.

Importante:

- **Valor da Atividade Computacional: 3.0 pontos**