UNIVERSIDADE VILA VELHA
ESPÍRITO SANTO

Curso de Ciência da Computação

1o Atividade Computacional - Concorrente

Data da entrega: 31/03/2025

5 componentes por grupo

Objetivo:
Pretende-se com este trabalho aplicar algoritmos e desenvolver soluções com concorrência.

Considerações:

- Observe os requisitos funcionais de cada questão.
- O grupo pode, e deve, sempre que achar necessário incrementar com mais funcionalidades. Aplique os métodos de concorrência que achar necessário.
- O foco aqui **não é** construir *APIs Web* ou algo parecido. O foco é "Computação Raiz", ou seja, no desenvolvimento de algoritmos que busquem as soluções dos problemas computacionais propostos.

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*: a *thread* **AEsperta**, a *thread* **AEconomica** e a *thread* **AGastadora**, concorrentemente.

  - **1. AGastadora:** esta *thread* (de atitude voraz) deverá, a cada 3000 milesegundos verificar se há saldo suficiente, e retirar 10 reais da sua conta. Esta *thread* deve disputar por dinheiro, com as demais *threads*, concorrentemente.

  - **2. AEsperta:** esta *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, esta *thread* irá retirar 50 reais da sua conta. Esta *thread* deve disputar com outras *threads*, concorrentemente.

  - **3. AEconomica:** de todas 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. Esta *thread* deve disputar com outras *threads*, concorrentemente.

- **B)** A classe **Conta** (*recurso compartilhado*) deverá ter pelo menos:
  - Os seguintes atributos: **número** da conta, **titular** da conta e **saldo**.
  - 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 uma *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**, todas as *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**. Esta *thread* deverá depositar 100 reais sempre que a conta estiver zerada. Veja que este thread será "produtora". E as demais serão "consumidoras". (verifique a possibilidade de usar *wait* e *notifyAll* ou outro modelo, *por exemplo*)

- **E)** Fica a critério do grupo se irá 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:

COMPUTACAO.TXT

```
1234567890 BEATRIZ BRITO OLIVEIRA COMPUTAÇÂO CURSANDO
1234567890 DANIEL BATISTA ZOPPI COMPUTAÇÂO CONCLUIDO
1234567890 GUILHERME TOZZI MAFRA COMPUTAÇÂO CURSANDO
1234567890 HEITOR DIAS DALVI CONCLUIDO
1234567890 JOÃO OTÁVIO COELHO MENENGUCI COMPUTAÇÂO CURSANDO
1234567890 KIMBERLY SCALDAFERRO COLODETI COMPUTAÇÂO CONCLUIDO
1234567890 TIAGO COMETTI LOMBARDI COMPUTAÇÂO CONCLUIDO
1234567890 WILLIAN CHEN LIN COMPUTAÇÂO CURSANDO
1234567890 YURI SALIM GUIMARAES COMPUTAÇÂO CURSANDO
```

Dentro deste contexto pede-se: Elabore um código, 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. A base de dados deve ser gerada pelo grupo, de acordo com o modelo apresentado na questão.

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.

Dentro deste contexto pede-se: Elabore um código que permita atender a situação descrita anteriormente.

Dica: *Por exemplo*, 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

Existem várias situações em que o cálculo da soma de números inteiros armazenados em um grande vetor é necessário. Algumas das aplicações práticas incluem:

- *Análise de dados*: Em análise de dados, a soma de valores é um cálculo muito comum e importante. Por exemplo, podemos ter um grande vetor de vendas de uma empresa e precisamos calcular a soma total de vendas em um determinado período.
- *Processamento de imagens*: Em processamento de imagens, o cálculo da soma de valores de pixels em uma imagem pode ser útil para determinar a intensidade média ou total de luz na imagem.
- *Simulação de Monte Carlo*: A simulação de Monte Carlo é uma técnica estatística usada para avaliar a incerteza em modelos matemáticos através da geração de muitas amostras aleatórias. Em algumas aplicações, como finanças, a soma dos valores aleatórios é necessária para calcular uma média ou desvio padrão.
- *Aprendizado de Máquina*: Em muitas aplicações de aprendizado de máquina, os dados são armazenados em grandes vetores e o cálculo da soma de valores é necessário para treinar o modelo ou fazer previsões.
- *Criptografia*: Alguns algoritmos criptográficos usam somas de valores como parte do processo de criptografia ou descriptografia, especialmente em algoritmos baseados em criptografia de blocos.

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, o que pode ser muito lento.
- Uma abordagem mais eficiente é dividir o vetor em várias partes, por exemplo, em 10 partes, e calcular a soma de cada parte em uma *thread* separada, para aproveitar o poder de processamento de vários núcleos.

Dentro deste contexto pede-se: Elabore os códigos que permitam atender as duas abordagens propostas.

Dica: Para isso, *por exemplo*, 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. Mais informações sobre essa classe pode ser encontrada aqui. Dica → *addAndGet(int delta)*: *Adiciona atomicamente o valor fornecido ao valor atual.*

Questão 05

O processamento concorrente é essencial em sistemas modernos para melhorar a eficiência e a capacidade de resposta. Muitas aplicações precisam executar várias tarefas simultaneamente, como sistemas de simulação, jogos e serviços em nuvem.

Uma área onde o uso de *threads* se destaca é a simulação de tráfego urbano. Em cidades inteligentes, é fundamental monitorar e otimizar o fluxo de veículos, reduzindo congestionamentos e melhorando a mobilidade. Exemplos de potenciais aplicações que usam recursos de concorrência são:

- *Sistemas de semáforos inteligentes:* Sensores de tráfego enviam dados para servidores que ajustam os tempos dos semáforos dinamicamente, reduzindo congestionamentos.
- *Aplicativos de navegação:* Apps como Google Maps ou Waze processam dados de milhares de usuários simultaneamente para calcular rotas otimizadas.
- *Simulações para planejamento urbano:* Cidades usam simulações para prever o impacto de novas vias, mudanças de sentido ou obras na fluidez do trânsito.

Dentro deste contexto pede-se: Desenvolver uma aplicação que simule um sistema de tráfego urbano utilizando *threads* para representar veículos que transitam por um conjunto de ruas e cruzamentos. A simulação deve permitir a movimentação dos veículos de maneira concorrente, respeitando regras básicas, como a espera em cruzamentos e a liberação por semáforos.

Requisitos

- Criação de veículos: Cada veículo deve ser representado por uma *thread* e deve seguir uma rota específica.
- Sistema de cruzamentos: Os veículos devem parar e aguardar a liberação dos semáforos antes de prosseguir.
- Sincronização: Utilizar mecanismos como, *por exemplo*, *synchronized* ou *locks* para garantir que múltiplos veículos não ocupem o mesmo cruzamento ao mesmo tempo.
- Relatórios: Ao final da simulação, exibir estatísticas, como número total de veículos processados. Sugira outras informações, *por exemplo*, informações sobre os tempos de espera, etc.

Segue um exemplo de possível saída gerada:

```
Veículo 1 entrou no cruzamento.
Veículo 1 saiu do cruzamento.
Veículo 4 entrou no cruzamento.
Veículo 4 saiu do cruzamento.
Veículo 5 entrou no cruzamento.
Veículo 5 saiu do cruzamento.
Veículo 2 entrou no cruzamento.
Veículo 2 saiu do cruzamento.
Veículo 3 entrou no cruzamento.
Veículo 3 saiu do cruzamento.
```

```
== Veiculo == : Veículo 1
tempoEsperandoParaEntrarNoCruzamento: 0 ms
tempoNoCruzamento: 1000 ms
== Veiculo == : Veículo 2
tempoEsperandoParaEntrarNoCruzamento: 4000 ms
tempoNoCruzamento: 1000 ms
== Veiculo == : Veículo 3
tempoEsperandoParaEntrarNoCruzamento: 5500 ms
tempoNoCruzamento: 1000 ms
== Veiculo == : Veículo 4
tempoEsperandoParaEntrarNoCruzamento: 1500 ms
tempoNoCruzamento: 1000 ms
== Veiculo == : Veículo 5
tempoEsperandoParaEntrarNoCruzamento: 2500 ms
tempoNoCruzamento: 1000 ms
Simulação finalizada.
```

Questão 06

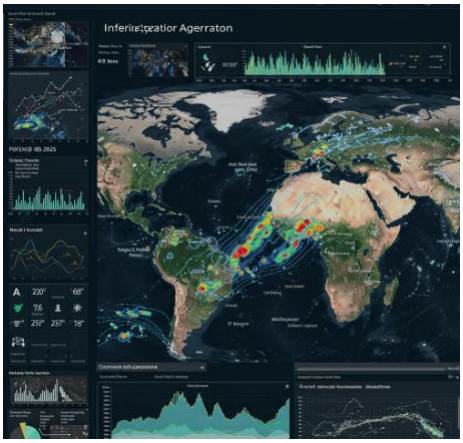
Nos sistemas modernos, a programação concorrente é essencial para garantir desempenho e escalabilidade. Muitas aplicações precisam integrar dados de diferentes fontes e processá-los em tempo real.

Em serviços de previsão do tempo, por exemplo, as informações meteorológicas são obtidas de múltiplas bases de dados, como estações climáticas, sensores e serviços de terceiros. Para consolidar esses dados rapidamente, utiliza-se, *por exemplo* API como "*CompletableFuture*". Exemplos de potenciais aplicações que usam recursos como estes podem ser:

- *Serviços de previsão do tempo*: Aplicações como o Weather.com ou o Climatempo agregam dados de diversas fontes para fornecer previsões mais precisas.
- *Integração de sensores em cidades inteligentes*: Estações meteorológicas distribuem dados em tempo real, permitindo monitoramento climático mais eficiente.
- *Análises climáticas para agricultura*: Fazendas utilizam múltiplas fontes de dados para prever secas e planejar a irrigação.

Dentro deste contexto pede-se: Desenvolver uma aplicação que simule um agregador de informações meteorológicas. A aplicação deverá coletar dados de diferentes bases de dados simuladas (utilizando coleções de dados como banco de dados, por exemplo), processá-los concorrentemente e consolidá-los para exibir um relatório final.

Requisitos



- Bases de dados simuladas: Os dados meteorológicos serão armazenados em listas ou mapas, simulando bancos de dados distintos.
- Coleta assíncrona: Buscar os dados das diferentes fontes concorrentemente, *por exemplo*, utilizando "*CompletableFuture*".
- Processamento de dados: Calcular a média da temperatura coletada. Sugira outras informações, *por exemplo*, informações sobre as localizações de cada base de dados, etc.

Segue um exemplo de possível saída, bem simples, gerada:

```
Iniciando coleta de dados...
Base1 respondeu com média: 22.35°C
Base3 respondeu com média: 21.55°C
Base2 respondeu com média: 24.375°C
Média geral da temperatura: 22.758333333333336°C
```

O que deve ser entregue:

- Arquivos contendo o projeto com os algoritmos e a documentação.
    1. Esse material deve ser submetido para o Blog, na tarefa de entrega específica para essa atividade, **até a data e horário configurados no sistema.**
    2. **O envio deve ser individual. A nota está condicionada a esse envio.**
    3. Algoritmos podem ser elaborados na linguagem de programação de sua escolha, desde que se encaixem nos quesitos propostos no trabalho.

Avaliação:

A avaliação será composta por duas partes:

- Avaliação do material pedido no item "**O que deve ser entregue**:"
- **Arguição INDIVIDUAL** dos componentes do grupo, no dia da entrega da atividade. Serão realizados vários questionamentos sobre o código apresentado.

Importante:

- Não serão pontuados os componentes que deixarem de entregar algum dos itens especificados.
- Procure ter um cuidado especial com a formatação da interação com o usuário (*entrada e saída de dados*).
- **Valor da Atividade Computacional: 2.0 pontos**