Semana 4: Coordenação I

Site: moodle23.iscte-iul.pt
Disciplina: Programação Concorrente e Distribuída
Livro: Semana 4: Coordenação I
Impresso por: Diana Andreia Amaro
Data: segunda-feira, 9 de outubro de 2023 às 18:19

Índice

1. Banquete de javalis
2. Lingotes de Ouro
3. Fila Bloqueante
4. Extra: Cadeia de distribuição
5. Extra: Barbeiro

1. Banquete de javalis

Programar o seguinte problema “produtor/consumidor”: num banquete de javalis assados existem vários cozinheiros e glutões concorrentes e uma mesa com capacidade limitada (e.g., só cabem na mesa 10 javalis). Cada cozinheiro repetidamente produz um javali assado e coloca-o na mesa. Caso encontre a mesa cheia fica à espera que a mesa fique com espaço disponível para depositar o seu javali. Cada glutão repetidamente retira da mesa um javali e consome-o. Caso encontre a mesa vazia fica à espera que a mesa contenha de novo algum javali. Sincronizar o recurso partilhado mesa por forma a coordenar devidamente os vários cozinheiros e glutões. Utilizar ainda mensagens apropriadas para observação do comportamento dos vários atores no banquete. Cada Javali deve ser identificado pelo cozinheiro que o produziu e por um número de ordem.

Versão 1: Considerar que os cozinheiros/glutões produzem/consomem um número fixo de javalis.

Versão 2: Considerar que os cozinheiros/glutões ficam a funcionar um determinado tempo (p.ex. 10s), após o que são interrompidos e param.

Versão 3: Teste a solução envolvendo apenas dois cozinheiros que produzem 100 javalis, dois glutões que também consomem 100 javalis, e espaço na mesa para um único javali. Neste caso, use `notify` em vez de a solução recomendada, que usa `notifyAll`. Que problema pode surgir?

2. Lingotes de Ouro

Um lingote típico de ouro é construído com ~12.5 kg de ouro. Para fazer um lingote, são precisos diversos pedaços de ouro extraídos da natureza.
Crie uma aplicação que simule uma escavadora de ouro, uma balança e um ourives. A escavadora recolhe pequenos pedaços de ouro (cada pedaço pesa até 1 kg). Quando a escavadora acha o ouro, coloca-o na balança. Quando estiverem pelo menos 12.5 kg de ouro na balança, o ourives recolhe o ouro para fazer o lingote. A escavadora e o ourives devem ser *threads*. A escavadora pode acrescentar ouro na balança apenas se houver menos do que 12.5 kg de ouro na balança. Por outro lado, o ourives pode apenas recolher o ouro quando ficam 12.5 kg ou mais na balança: retira os necessários 12.5 kg e deixa o resto na balança. Depois de o ourives recolher o ouro da balança, leva-lhe 3 segundos a transformar o ouro num lingote.
Utilize um double para representar a quantidade momentânea de ouro na balança. Deverá criar uma janela com o campo de texto e um botão "Stop". O campo de texto deve mostrar o peso do ouro que está na balança.
Quando o utilizador pressiona o botão “Stop”, a escavadora e o ourives devem ser interrompidos e terminar. Antes de terminar, o ourives deve escrever na consola o número total de lingotes criados.

A sua solução deve enviar mensagens que contenham o id da thread (`Thread.currentThread().toString()`) bem como o tipo de operação (ex: “inicio do run”) para a consola sempre que uma thread faz:

- Inicio e fim do método run
- Antes de tratar uma InteruptedException
- Antes de fazer um join
- Antes e depois de fazer um wait ou um sleep

Exemplo:System.out.println(Thread.currentThread().toString() + “ - inicio do run.”);
Dica: para atualizar a GUI, a balança pode receber no construtor uma instância de `JTextField` que será atualizado (usando `setText`) cada vez que o conteúdo da balança for atualizado.

3. Fila Bloqueante

O objetivo deste exercício é desenvolver uma estrutura de dados genérica para Filas Bloqueantes . Esta estrutura consiste numa Fila (*First In First Out*) com as operações **put** (adiciona um elemento no fim da fila) e **take** (retira e devolve o primeiro elemento da fila). Porém, as chamadas a estas operações são bloqueantes (usando wait) nas seguintes situações :

- **put**: no caso de a fila estar no limite da sua capacidade.
- **take**: no caso de a fila estar vazia.

Desta forma, a fila pode ser acedida concorrentemente, e é útil para problemas cuja solução encaixa no padrão produtor-consumidor.
Implemente a estrutura numa classe chamada **BlockingQueue**, tendo em conta o seguinte:

- Os objetos são parameterizados com um parâmetro genérico <T> para o tipo de elementos da fila. Internamente, os elementos devem ser guardados numa estrutura apropriada para filas (convencionais), tal como `LinkedList` (lista ligada) ou `ArrayDeque`.
- Deverão ser disponibilizados dois construtores:
  - Sem parâmetros: neste caso, a estrutura não terá uma capacidade máxima, e, assim, a operação **put** nunca bloqueará e a operação **take** não necessitará de notificar outras *threads* (pois não haverá *threads* bloqueadas em poll)
  - Com um parâmetro para definir a capacidade da fila (pré-condição: número positivo).
- Para além das operações **`put`** e **`take`**, deverá existir também a operação **size** (para obter o número de elementos da fila) e **clear** (para esvaziar a fila). As operações bloqueantes devem propagar as exceções de interrupção de *thread* (`InterruptedException`) para os clientes.

```
public class BlockingQueue <T> {
  private Queue <T> queue = ...
...
}
```

4. Extra: Cadeia de distribuição

Crie uma aplicação que simule um *fornecedor,* um *distribuidor* e 5 *retalhistas*.
O fornecedor fornece sucessivamente produtos ao distribuidor. O distribuidor tem capacidade para armazenar todos os produtos recebidos numa única lista (use uma `LinkedList`). Quando há 10 ou mais produtos na lista, o distribuidor pode vender os produtos. A venda é feita aos retalhistas que adquirem todos os produtos do distribuidor. Assim, após uma aquisição, a lista de produtos do distribuidor é esvaziada.
Cada produto gerado pelo fornecedor deve ser representado por um objeto do tipo inteiro, com valores aleatórios de zero a nove (classe `Integer`, de forma a poder ser usada como elementos da lista).
Deverá criar uma janela com um campo de texto e um botão (“Stop”). O campo de texto deve mostrar a lista actualizada de produtos existentes no distribuidor. Para escrever os conteúdos da lista no campo de texto basta aplicar o método `toString` à lista, por exemplo:

```
// assumindo as variáveis previamente declaradas...
JTextField textfield = new JTextField();
LinkedList<Integer> lote = new LinkedList<Integer>; (...)
// escreve-se os conteúdos da lista no campo de texto assim...
textfield.setText(lote.toString());
```

Quando o utilizador pressiona o botão “Stop”, os retalhistas e o fornecedor devem terminar. No entanto, o fornecedor só deve ser terminado após se ter a certeza que os retalhistas terminaram. Quando um retalhista termina deve escrever o número de lotes adquiridos, por exemplo:

```
Retalhista 0: Comprei um total de 7 lotes. Retalhista 1: Comprei um total de 30 lotes.
Retalhista 2: Comprei um total de 50 lotes. Retalhista 3: Comprei um total de 56 lotes.
Retalhista 4: Comprei um total de 30 lotes.
```

A sua solução deve enviar mensagens que contenham o id da thread (`Thread.currentThread().toString()`) bem como o tipo de operação (ex: “inicio do run”) para a consola sempre que uma thread faz:

- Inicio e fim do método `run`
- Antes de tratar uma `InteruptedException`
- Antes de fazer um `join`
- Antes e depois de fazer um `wait` ou um `sleep`

Exemplo:
System.out.println(Thread.currentThread().toString() + " - inicio do run.");

Dica: Este problema deve ser resolvido de acordo com o modelo do produtor-consumidor. Inicialmente, identifique bem quem vai assumir os papéis de consumidor, produtor e recurso partilhado.
Nota: quando uma interrupção é recebida num método bloqueante (p.ex. `wait` ou `sleep`), normalmente vai ser corrido o bloco `catch` envolvente. Neste caso, a *flag* de interrupção é reposta e a *thread* deixa de ter o registo que foi interrompida. Caso seja necessário continuar a detetar a interrupção noutro ponto do código, pode-se voltar a chamar o `interrupt` dentro do bloco `catch`.

5. Extra: Barbeiro

Uma barbearia consiste numa sala de espera com 3 cadeiras e numa sala do barbeiro apenas com a cadeira.
Caso não existam clientes para serem atendidos o barbeiro deve ficar à espera.
Se um cliente entra na barbearia e todas as cadeiras estão ocupadas, então o cliente sai da loja, e regressa após um tempo entre 3 e 10 segundos. Se o barbeiro está ocupado, mas existem cadeiras disponíveis na sala de espera, o cliente senta-se numa das cadeiras livres. Se o barbeiro está inativo, à espera de clientes, deve ser notificado da presença do cliente.
Os clientes devem constantemente cortar o cabelo e depois dormir um tempo aleatório entre 1 e 10 segundos.
Escreva um programa para coordenar o barbeiro e os clientes.
A barbearia deve ainda ter uma janela onde existem um campo de texto e um botão. No campo de texto deve indicar o nº de cadeiras de espera ocupadas. O botão ("Stop"), quando pressionado, deve terminar todas as threads activas. Quando terminados, os clientes devem dizer quantas vezes cortaram o cabelo e o barbeiro quantos cortes efetuou.
Teste o seu programa com 10 clientes e um barbeiro.
A sua solução deve enviar mensagens que contenham o id da thread (`Thread.currentThread().toString()`) bem como o tipo de operação (ex: "inicio do run") para a consola sempre que uma thread faz:

- Inicio e fim do método `run`
- Antes de tratar uma `InteruptedException`
- Antes de fazer um `join`
- Antes e depois de fazer um `wait` ou um `sleep`