UFRJ

Universidade Federal do Rio de Janeiro

MAB117

Computação Concorrente

Sistema Táxi-Passageiro

Aluno:	*DRE:*
Renato Pontes Rodrigues	113131049
Ygor Luís M. P. da Hora	113043644

7 de Julho de 2015

Sumário

1 Documentação da solução

Introdução

A aplicação foi desenvolvida utilizando-se a linguagem **Java**. Por ser um programa concorrente, não somos capazes de descrever o fluxo de execução de forma linear.

O que fazemos a seguir é descrever cada classe `*.java` que criamos e o papel de suas instâncias durante a execução da aplicação.

```
#include <bits/stdc++.h>

using namespace std;

class Package {
public:
    int qt;
    int wt;

    Package(int qt, int wt): qt(qt), wt(wt) {}
};

class State {
public:
    int qt;
    bitset<128> ch;

    State(): qt(0), ch(0) {}
    State(int qt, string ch): qt(qt), ch(ch) {}
};

State maxtoys(int cap, int pac, vector<Package>& p) {
    vector<vector<State> > mt(cap+1, vector<State>(pac+1));
    // int wt = 0;

    for (int i = 0; i <= cap; ++i) {
        for (int j = 0; j <= pac; ++j) {
            if (i == 0 or j == 0) {
                mt[i][j] = State();
            }
            else if (p[j-1].wt <= i) {
                if (mt[i][j-1].qt < p[j-1].qt + mt[i-p[j-1].wt][j-1].qt
                    ) {
                    mt[i][j] = State(p[j-1].qt + mt[i-p[j-1].wt][j-1].
                        qt, mt[i-p[j-1].wt][j-1].ch.to_string());
                    mt[i][j].ch.set(j-1);
```

```
                }
                else {
                    mt[i][j] = mt[i][j-1];
                }

            }
            else {
                mt[i][j] = mt[i][j-1];
            }
        }
    }

    return mt[cap][pac];
}

int main() {
    int tc, pac, qt, wt;

    cin >> tc;
    while(tc--) {
        cin >> pac;
        vector<Package> P;

        for (int i = 0; i < pac; ++i) {
            cin >> qt >> wt;
            P.push_back(Package(qt, wt));
        }

        State sol = maxtoys(50, pac, P);

        int weight = 0;
        int rem = 0;
        for (int i = 0; i < pac; ++i) {
            if (sol.ch.test(i)) {
                weight += P[i].wt;
            } else {
                rem++;
            }
        }

        cout << sol.qt << "␣brinquedos\nPeso:␣" << weight << "␣kg\
           nsobra(m)␣" << rem << "␣pacote(s)\n\n";
    }
}
```

1.1 Main

O objetivo da classe `Main` é fornecer a via de entrada para o início da execução do programa, permitindo a leitura dos dados para a simulação da aplicação distribuída e início do processamento. Esta classe garante ainda que ao fim da simulação seja exibido um relatório com a posição final dos taxistas e os tempos de execução de cada parte da aplicação, na forma descrita na seção *Saída do programa*. O único método desta classe é **`public static void main(String[] args)`**, que inicia a aplicação e é descrito a seguir:

Uma instância da classe `Scanner` permite a leitura dos dados de entrada especificados na descrição do trabalho a partir do dispositivo de entrada padrão. A matriz que representa o mapa da cidade é implícita, e por isso as dimensões N e M fornecidas na entrada são ignoradas. Em seguida, uma vez que o número de clientes é especificado, é criada uma instância para a classe `Central` com o papel de intermediar a comunicação entre clientes e taxistas—que como veremos, terão cada um sua própria linha de execução. Cada cliente e suas especificações ficam associados a uma instância da classe `Client`, com capacidade de tornar-se uma thread. O mesmo encapsulamento serve para os taxistas que por sua vez estarão associados a uma instância da classe `Taxi`. Conforme taxistas e clientes são instanciados, ambos são incluídos como elemento de uma lista de threads.

Após todos os dados de entrada serem obtidos é efetivado o lançamento das instâncias das classes `Client` e `Taxi` vistas na lista de threads formada. Observe ainda que utilizamos um método estático da classe `java.util.Collections` chamado `reverse()` com o objetivo de inverter a ordenação da lista, de modo a fazer com que as threads taxistas tenham uma chance maior de serem iniciadas antes das threads clientes. Este cenário nos pareceu mais interessante porque isto também significa que o primeiro cliente tem mais chances de escolher entre vários táxis, ao invés de ser associado a um dos primeiros táxis a ficarem disponíveis.

1.2 Coord

Antes de efetivamente entendermos o funcionamento das demais classes do trabalho, e que tem papel estrutural fundamental na realização do mesmo, entenderemos o funcionamento de uma classe com papel organizacional chamada `Coord`. A classe `Coord` tem o objetivo de modularizar os pontos em coordenadas cartesianas a que cada instância de `Client` e `Taxi` estará associada. Seja este ponto de origem ou de destino, todos serão vistos como instâncias da classe `Coord`. Além desta funcionalidade, a classe encapsula alguns métodos úteis relativos ao con-

texto cartesiano.

1.2.1 Atributos

x, y
Coordenadas x e y do ponto (x, y) representado pelo objeto.

1.2.2 Métodos

`public void set(Coord dest)`
Dada uma instância da classe `Coord` é possível mudar seus atributos, ou seja, suas coordenadas cartesianas mesmo após a sua construção, copiando os atributos de uma outra instância da classe `Coord`.

`public int distanceTo(Coord c2)`
A partir de toda instância de `Coord` é possível encontrar sua distância de Manhattan até outro ponto coordenado também representado por uma instância de `Coord`. Este método retorna exatamente esta distância. O cálculo desta distância nada mais é que o valor absoluto da diferença entre abcissas de dois pontos somado ao valor absoluto da diferença entre ordenadas de dois pontos.

`public String toString()`
O objetivo deste método é definir uma `String` representante da classe que deverá ser retornada quando uma instância da classe precisa ser exibida pelo método de saída padrão.

`public int getX()`
`public int getY()`
Os outros métodos desta classe são getters para acessar alguns atributos privados de cada objeto `Coord` a partir da instância de `Central`.

1.3 Client

A primeira classe que abordaremos com papel especialmente estrutural na implementação da simulação do sistema distribuído é a classe `Client`. No momento de sua construção na classe `Main` ela tem o papel de encapsular dados. A partir do seu lançamento e ganho de processamento, ou seja, execução do método `run()` da classe `Client`, esta representa um novo contexto de execução, tornando-se uma thread que concorre com outras lançadas.

1.3.1 Atributos

`idcount`
Uma variável estática responsável por gerar identificadores distintos para cada objeto `Client`.

`id`
Uma variável inteira que guarda o identificador de cada objeto `Client`.
$1 \leq id \leq P$

`origin, dest`
Dois objetos `Coord` que guardam as coordenadas de origem e destino associados a cada instância de um objeto `Client`.

`central`
Uma referência para a única instância de `Central` que existe na aplicação.

1.3.2 Métodos

`public void run()`
Cada thread tem o único objetivo de simular a requisição de um cliente por um táxi ao núcleo do sistema distribuído.

`public int getId()`
`public Coord getDest()`
`public Coord getOrigin()`
Estes métodos auxiliares são getters e setters utilizados por instâncias de `Taxi` e `Central` para acessar atributos privados do objeto `Coord`.

1.4 Taxi

Um objeto `Taxi` encapsula as propriedades e funções dos taxistas. Esta classe implementa a interface `Runnable`, e por isso suas instâncias são capazes de executar o método `run()` concorrentemente.

1.4.1 Atributos

`idcount`
Uma variável estática responsável por gerar identificadores distintos para cada objeto `Taxi`.

id
Uma variável inteira que guarda o identificador de cada objeto `Taxi`.
$1 \leq id \leq T$

coord
Um objeto `Coord` que guarda as coordenadas atuais do objeto `Taxi`.

central
Uma referência para a única instância de `Central` que existe na aplicação.

currentclient
Um objeto `Client`. Se o valor desta variável é `null`, isto quer dizer que o táxi não tem cliente e está disponível para ser escolhido. Quando um objeto `currentclient` é criado, o valor desta variável é `null`.

1.4.2 Métodos

public void run()
Descreve a lógica das threads que representam os táxis. Um táxi deve tentar atender clientes enquanto houver clientes sem táxis associados. Isto é feito com um laço infinito. A cada iteração, o taxi usa o método `central.getRequest()` para informar que está disponível e aguardar que seja escolhido para atender um cliente. Se após a execução deste método o campo `currentclient` for `null`, então todos os clientes foram atendidos e a thread pode interromper o loop, sair deste método e ser finalizada. Se um cliente foi designado para o táxi, ele realiza o atendimento ao cliente utilizando o método privado `travel()`, e em seguida tenta atender outro cliente.

private void travel()
Este método bloqueia a thread por certo tempo, para simular a viagem do táxi até o cliente e do cliente até o seu destino. O tempo de cada bloqueio é proporcional a distância de cada viagem.

public int getId()
public Coord getCoord()
public Client getCurrentclient()
public void setCurrentclient(Client currentclient)
Estes métodos auxiliares são getters e setters para acessar/modificar alguns atributos privados de cada objeto `Taxi` a partir da instância de `Central`.

1.5 Central

A classe `Central` faz o papel de um **monitor**. Ela encapsula toda a lógica de sincronização das threads. Para garantir o comportamento adequado da aplicação, só deve existir uma instância desta classe conhecida pelos objetos `Taxi` e `Client` ao longo de toda a aplicação.

1.5.1 Atributos

`nclient`
Guarda o total de clientes que ainda não tem um táxi associado, mesmo que eles não tenham feito um pedido. Este atributo é utilizado como critério de parada da aplicação. Ele é inicializado com o número de clientes dado pela entrada do algoritmo e decrementa até alcançar o valor zero.

`clientlog`
Variável que guarda quantos clientes estão em contato com a Central, mas ainda não foram associados a um táxi. Esta variável existe apenas para ser usada no log de execução.

`ltaxis`
Uma lista de objetos `Taxi`. Ela contém apenas táxis disponíveis.

`finalcoord`
Um vetor de objetos `Coord` que contém as posições finais de cada taxista. A posição i do vetor guarda a posição final do taxista $i + 1$. Este atributo existe apenas para não intercalar a saída do programa com o log de execução e também para imprimir as posições finais numa determinada ordem.

1.5.2 Métodos

`public synchronized void getRequest(Taxi taxi)`
Este método é usado pelas instâncias de `Taxi`, para informar a `Central` que aquele táxi em particular está disponível para ser escolhido por um cliente. Quando um táxi informa isto, ele é colocado num objeto `ArrayList` administrado pela `Central`, que contém os táxis disponíveis. Se não existem clientes para serem atendidos, a `Central` simplesmente retira o táxi deste método, sem cliente. Se este é o primeiro táxi a ficar disponível, ele tenta notificar os clientes que poderiam ter pedido táxi enquanto nenhum estava disponível. Depois de ser colocado nesta lista, o táxi se bloqueia, aguardando ser escolhido para atender algum cliente.

Se um cliente foi associado a este táxi, o táxi acorda e sai deste método, levando com ele uma referência para o cliente que deve ser atendido. Se todos os clientes foram atendidos, o táxi acorda e sai do método, sem cliente. A exclusão mútua é necessária nesse método, por exemplo, porque objetos `ArrayList` (onde os táxis disponíveis são enfileirados) não são thread-safe.

`public synchronized void makeRequest(Client client)`
Este método é chamado pelas instâncias de `Client`, quando o cliente quer pedir um táxi. A exclusão mútua é justificada pelo uso de variáveis usadas por outros métodos da `Central`, e também porque a `Central` não pode associar o mesmo táxi a mais de um cliente.

O método inicialmente verifica se há táxis na lista de táxis disponíveis que a `Central` mantém. Se não existem táxis disponíveis, o cliente aguarda pela disponibilidade de táxis. Se há táxis disponíveis, a `Central` utiliza o método privado `chooseTaxi()` para determinar o táxi ótimo para este cliente dentre os táxis que estão disponíveis naquele momento. Depois de ter escolhido o táxi, uma referência para este cliente é dada ao táxi designado, e o táxi é retirado da lista de táxis disponíveis. Decrementamos o número de clientes que ainda precisam ser atendidos e usamos um `notifyAll()` para comunicar ao táxi escolhido de que ele já pode atender o cliente. O método então termina.

`private Taxi chooseTaxi(Client client)`
Este método escolhe e retorna, dentre a lista de táxis disponíveis, o táxi que está a menor distância do objeto `Client` passado como argumento, com auxílio do método `distanceTo()` da classe `Coord`.

`public void report(int id, Coord coord)`
Este método é utilizado pelas instâncias de `Taxi` para que estas informem a `Central` suas posições quando não há mais clientes para serem atendidos. Cada táxi escreve suas coordenadas na posição $id - 1$ da lista de coordenadas finais mantida pela `Central`. Como todos os táxis tem identificadores distintos, nenhum deles escreve na mesma posição e por isso este método não precisa de mecanismos de sincronização.

`public void printReport()`
Este método é chamado pelo método `Main.main()` para pedir que a instância de `Central` imprima a saída do programa depois que todas as threads terminam. Ele cria um objeto `StringBuilder` (otimizado para concatenações), e adiciona a esse objeto uma a uma as coordenadas finais dos taxistas, sempre obedecendo ao formato especificado na descrição do trabalho. Este método também garante que uma linha em branco exista entre o fim do log e o início da saída e outra entre o fim da saída e os tempos de execução.

2 Utilização da aplicação

2.1 Compilação

ATENÇÃO: Para compilar é necessário ter o compilador **JAVAC ≥ 1.7.x** instalado. Versões mais antigas podem funcionar, mas não foram testadas.

Linux

Para compilar a aplicação no Linux, basta abrir o terminal na raiz do diretório `TaxiDriver` e utilizar o `makefile` incluso, digitando

```
$ make
```

ou, para compilar manualmente,

```
$ mkdir -p bin
$ javac -sourcepath src -cp bin -d bin -encoding utf8 src/Main.java
```

Os arquivos `*.class` serão gerados no diretório `bin` (será criado se não existir).

Windows

Alguns pacotes incluem um porte do `make` para Windows (*Ruby DevKit*, *MinGW*). Nessas condições, basta utilizar o `makefile` incluso de forma análoga ao processo no Linux.

Para compilar a aplicação no Windows manualmente, basta abrir a linha de comando na raiz da pasta `TaxiDriver` e utilizar os seguintes comandos:

```
$ mkdir -p bin
$ javac -sourcepath src -cp bin -d bin -encoding utf8 src/Main.java
```

Os arquivos `*.class` serão gerados na pasta `bin` (será criada se não existir).

2.2 Gerador de casos de teste

Na pasta `tc` está incluído um script `gen_tc.py` (deve ser executado com Python 3) que gera casos de teste para a aplicação.

No Linux pode ser necessário dar permissão para o script ser executado:

```
$ chmod +x gen_tc.py
```

É recomendável que o script esteja no HD e não em um disco removível. Para executá-lo basta estar na pasta `tc` e escrever:

```
$ ./gen_tc.py
```

É necessário fornecer ao gerador os parâmetros N, M, P e T, conforme instruções exibidas na tela pelo programa. Note que estes parâmetros devem estar em conformidade com os limites estabelecidos na descrição do trabalho:

$$4 \leq N, M \leq 1000$$
$$1 \leq P \leq 200$$
$$1 \leq T \leq 100$$

Se qualquer uma destas condições for violada, o script se recusará a gerar o caso de teste.

2.3 Entrada do programa

A entrada tem o formato especificado na descrição do trabalho e é lida do dispositivo de entrada padrão.

Existem casos de teste inclusos junto ao código-fonte. Com o terminal/cmd aberto na pasta `bin`, podemos utilizar estes casos de teste fazendo

```
$ java Main < ../tc/sample2.txt
```

Consulte o diretório `TaxiDriver/tc` para consultar/gerar outros casos de teste. Segue um exemplo de caso de teste válido:

```
4 5
3
1 2 4 1
2 3 2 2
3 0 0 0
3
0 3
3 1
4 0
```

2.4 Saída do programa

Durante a execução, o programa imprime na tela mensagens que informam sobre o estado dos taxistas e passageiros. Quando todas as threads terminam, uma linha em branco é impressa e seguem T linhas, onde a i-ésima linha indica a posição final do i-ésimo táxi fornecido na entrada. Depois temos uma linha em branco. Seguem, por fim, exatamente quatro linhas que imprimem detalhes do tempo de execução. Mostramos a seguir uma possível saída para o caso de teste `tc/sample2.txt` mostrado na sessão anterior:

```
Táxi #3 está disponível. Total de 1 táxi disponível.
Táxi #2 está disponível. Total de 2 táxis disponíveis.
Táxi #1 está disponível. Total de 3 táxis disponíveis.
Cliente #1 pediu um táxi. Total de 1 cliente aguardando atendimento
O cliente #1 será atendido pelo táxi #1
2 táxis disponíveis e 0 clientes aguardando atendimento
Cliente #3 pediu um táxi. Total de 1 cliente aguardando atendimento
O cliente #3 será atendido pelo táxi #3
1 táxi disponível e 0 clientes aguardando atendimento
Cliente #2 pediu um táxi. Total de 1 cliente aguardando atendimento
O cliente #2 será atendido pelo táxi #2
0 táxis disponíveis e 0 clientes aguardando atendimento
Táxi #3 alcançou o cliente #3
Táxi #1 alcançou o cliente #1
Táxi #2 alcançou o cliente #2
Taxi #3 chegou ao destino (0, 0) do cliente #3
Taxi #2 chegou ao destino (2, 2) do cliente #2
Táxi #3 terminou o expediente na posição (0, 0)
Táxi #2 terminou o expediente na posição (2, 2)
Taxi #1 chegou ao destino (4, 1) do cliente #1
Táxi #1 terminou o expediente na posição (4, 1)

4 1
2 2
0 0

Tempo de execução da leitura de dados da entrada: 0 ms
Tempo de execução do processamento concorrente: 16 ms
Tempo de execução da impressão da saída: 0 ms
Tempo de execução total: 16 ms
```

Note que a mensagem `Total de 1 cliente aguardando atendimento` conta apenas clientes que já contactaram a Central. Da mesma forma, `0 clientes aguardando atendimento` considera clientes que estão em contato com a Central mas que ainda não foram associados a nenhum táxi. Em outras palavras, um cliente ser atendido pela Central é diferente do cliente ter chegado a seu destino (este evento é notificado em mensagens do tipo `Taxi #3 chegou ao destino (0, 0) do cliente #3`).

3 Relatório de execução

3.1 Descrição dos testes realizados e resultados obtidos

A aplicação foi testada numa máquina com 4 processadores. Testamos basicamente três tipos de caso de teste que julgamos suficientes para avaliar a corretude do programa. Utilizamos sempre o pior caso em que o mapa tem dimensões $1000x1000$, o maior tamanho possível. Para cada caso foram feitas 10 execuções e os tempos são todos dados em **milissegundos**. Descrevemos abaixo cada um desses casos:

1. **Um passageiro e vários taxistas**
   Utilizamos o arquivo `tc/1000_1000_1_100.txt` para avaliar este caso. Encontramos as seguintes médias de tempo de execução:

Tempo médio de entrada	16.7
Tempo médio do processamento concorrente	466.2
Tempo médio da impressão da saída	3.6
Tempo total médio	486.5

   Este foi o caso mais rápido, como esperávamos. Como o número de clientes que chegaram ao seu destino é condição de parada da aplicação, é natural que um caso de teste que só contém um passageiro termine rapidamente.

2. **Vários passageiros e um taxista**
   Utilizamos o arquivo `tc/1000_1000_30_1.txt` para avaliar este caso. Encontramos as seguintes médias de tempo de execução:

Tempo médio de entrada	10.4
Tempo médio do processamento concorrente	41204.5
Tempo médio da impressão da saída	0.1
Tempo total médio	41215

Este foi o caso mais demorado. Tão demorado que consideramos não usar o número máximo de passageiros, diminuir o número de execuções ou diminuir o tamanho do mapa. Decidimos manter o número de execuções e tamanho do mapa e diminuir o número de passageiros para 30. Este resultado era esperado já que este caso sequencializa o atendimento dos taxistas aos clientes. Como só existe um táxi, ele tem que atender um cliente por vez.

3. **Vários passageiros e vários taxistas**
   Utilizamos o arquivo `tc/1000_1000_200_100.txt` para avaliar este caso. Encontramos as seguintes médias de tempo de execução:

Tempo médio de entrada	46.5
Tempo médio do processamento concorrente	3817.7
Tempo médio da impressão da saída	3.2
Tempo total médio	3867.4

Utilizamos o número máximo de clientes e taxistas possíveis na entrada, considerando que a razão 1/2 entre o número de taxistas e o número de passageiros seria suficiente para um processamento mais real da aplicação. Este caso foi mais rápido que o anterior, pois os táxis atendem vários clientes ao mesmo tempo, mas foi mais lento que o primeiro, que era o caso trivial em que só era necessário atender um cliente.

O **tempo médio de entrada** é o tempo necessário para ler os arquivos de entrada e criar as threads clientes e taxistas.

O **tempo médio de processamento concorrente** inclui o lançamento de todas as threads e o tempo da simulação do sistema táxi-passageiro (incluindo impressão do log de execução).

O **tempo médio de impressão da saída** inclui apenas a impressão da saída no formato especificado na descrição do trabalho.

O **tempo total médio** é a soma dos outros três tempos médios.

3.2 Dificuldades encontradas e estratégias adotadas

A lógica de sincronização entre threads taxistas e clientes foi a principal dificuldade encontrada. Em particular, foi difícil pensar em como fazer a central informar a um taxista que ele havia sido escolhido. Apesar de ser intuitivo que um táxi possui um cliente, tentamos primeiro algumas estratégias que falharam ou pareciam ineficientes, como utilizar um `HashMap` e vetores que mantinham o

estado de cada cliente e taxista. No fim, nos aproveitamos da orientação a objetos fornecida pela linguagem para passar o objeto `Cliente` diretamente ao objeto `Taxi` por meio da instância de `Central`, implementando a ideia intuitiva de que um táxi possui um cliente.