Universidade Federal de Santa Catarina
Centro de Florianópolis
Departamento de Informática
e Estatística

Lucas P. Tonussi

T2 Computação Paralela

Florianópolis
4 de dezembro de 2019

Lucas P. Tonussi

T2 Computação Paralela

Trabalho LAB 01: programação paralela (multiprocessador).

Orientador: Prof. Dr. Odorico Machado Mendizabal

Universidade Federal de Santa Catarina
Centro de Florianópolis
Departamento de Informática
e Estatística

Florianópolis
4 de dezembro de 2019

Lucas P. Tonussi

T2 Computação Paralela

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de Engenheiro de Controle e Automação.

Comissão Examinadora

Prof. Dr. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina
Orientador

Prof. Dr. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina

Prof. Dr. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina

Florianópolis, 4 de dezembro de 2019

Dedico este trabalho a todos aqueles que, de alguma forma,
auxiliaram para a concretização desta etapa.

Agradecimentos

Agradeço a meu pai, minha mãe e a você.

"Train yourself to let go of everything you fear to lose."
(Yoda)

Resumo

Resumo em português.

Palavras-Chave: 1. computação paralela. 2. threads. 3. multi núcleos. 4. semáforos.

5. mutexes. 6. java. 7. problema da montanha russa

Abstract

Abstract in english.

Keywords: 1. parallel computing. 2. threads. 3. multi cores. 4. semaphores. 5. mutexes. 6. java. 7. roller coaster problem

Lista de figuras

Lista de tabelas

Lista de Siglas e Abreviaturas

PCS	*Parallel Computing Systems*
UFSC	*Universidade Federal de Santa Catarina*

Sumário

1 Introdução

O problema da montanha russa

Figura 1 – Uma montanha russa, fonte: Stephen Hateley.

Esse problema retirado do livro de programação concorrente de Andrews, mas ele atribui esse problema (quebra-cabeças) à tese de mestrado de J. S. Herman [1].

Suponha que existem N threads passageiros(as) e uma thread carro. Os passageiros, repetidamente, esperam para passear no carro, que pode acomodar C passageiros(as), onde $C < n$ (mais passageiros do que assentos). O carro pode ir pelos trilhos apenas quanto ele está cheio de passageiros(as). No mundo real o número C é fixo, realmente, mas o número N é algo imprevisível. Certa hora podem chegar 100 passageiros, mas também podem chegar apenas 2 passageiros. Medidas tem que ser tomadas para que o carro parta carregado de passageiros.

Aqui estão alguns detalhes adicionais:

1. Passageiros(as) devem invocar board e unboard.
2. O carro deve invocar load, run, unload.
3. Passageiros(as) não podem embarcar até que o carro tenha invocado load.
4. O carro não pode partir até que C passageiros(as) tenham embarcado.
5. Passageiros não pode desembarcar até que o carro tenha invocado unload.

Quebra-cabeças: Escreva código para que os passageiros(as) e o carro contenham essas restrições.

A fonte dessa descrição pode ser encontrada em: "The Little Book of Semaphores", de Allen B. Downey, página 153 (versão 2.2.1).

2 Implementações

A seguir estão as implementações em Java do problema da montanha russa descrito conforme descrito no livro "The Little Book of Semaphores", de Allen B. Downey, página 153-157 (versão 2.2.1).

2.0.1 Main.java

Lança a aplicação.

```
1  package br.ufsc.montanharussa;
2
3  public class Main {
4
5      public static void main(String[] args) throws InterruptedException {
6          RollerCoaster roller = new RollerCoaster();
7          roller.start();
8      }
9
10 }
```

2.0.2 Shared.java

Simula a execução de *tasks* fictícias, cada passageiro irá entrar no carro e passear uma vez. Cada passageiro, como uma *thread*, irá incrementar a variável *count*, na classe *Shared*, uma vez. Foi feito assim para termos um controle de como os passageiros estão se comportando. Ou seja, se existem 200 passageiros, o contador ao final deverá conter o valor 200. Como se fossem 200 *tasks* "completadas".

```
1  package br.ufsc.montanharussa;
2
3  public class Shared {
4
5      volatile int count = 0;
6
7      public synchronized void add() {
8          count = count + 1;
9      }
10 }
```

2.0.3 RollerCoaster.java

```java
package br.ufsc.montanharussa;

import java.util.*;
import java.util.concurrent.*;
import java.util.logging.Logger;

// Introduction to Semaphore https://youtu.be/e2ujg5K310s
public class RollerCoaster extends Thread {

    final static Logger logger = Logger.getLogger(RollerCoaster.class.
    getName());

    Shared memory;

    final int MAX_PASSENGERS = 20;
    final int SEATS = 4;

    Car car;

    // mutex protects passengers, which counts the
    // number of passengers that have invoked boardCar.
    Semaphore mutex;
    Semaphore mutex2;

    volatile int boarders;
    volatile int unboarders;

    // passengers wait on boardQueue before boarding
    Semaphore boardQueue;

    // passengers wait on unboardQueue before unboarding.
    Semaphore unboardQueue;

    // allAboard indicates that the car is full
    Semaphore allAboard;

    Semaphore allAshore;

    // Fila de passageiros
    List<Passenger> passengers = new LinkedList<>();

    Random r = new Random(192839182370417291 1L);

    public RollerCoaster() {
        boarders = 0;
        unboarders = 0;

```

```
47          mutex = new Semaphore(1);
48          mutex2 = new Semaphore(1);
49
50          boardQueue = new Semaphore(SEATS);
51          unboardQueue = new Semaphore(SEATS);
52
53          allAboard = new Semaphore(SEATS);
54          allAshore = new Semaphore(SEATS);
55      }
56
57      @Override
58      public void run() {
59          car = new Car("Roller Coaster Vehicle", SEATS, boardQueue,
    unboardQueue,
60                  allAboard, allAshore, boarders, unboarders);
61
62          car.start();
63
64  //          int aux = r.nextInt((MAX_PASSENGERS - SEATS - 1) + 1) + SEATS -
    1;
65          memory = new Shared();
66          for (int i = 0; i < MAX_PASSENGERS; i++) {
67              Passenger passenger = new Passenger("A Person " + i, mutex,
68                      mutex2, boarders, unboarders, boardQueue, unboardQueue,
69                      allAboard, allAshore, SEATS, memory);
70              passengers.add(passenger);
71          }
72
73          for (Passenger passenger : passengers) {
74              passenger.state = PassengerAction.WANTING_A_RIDE;
75              passenger.start();
76          }
77      }
78  }
```

2.0.4 Passenger.java

```
1  package br.ufsc.montanharussa;
2
3  import java.util.concurrent.Semaphore;
4  import java.util.logging.Logger;
5
6  public class Passenger extends Thread {
7
8      final static Logger logger = Logger.getLogger(Passenger.class.getName()
    );
9
```

```
10      Semaphore mutex;
11      Semaphore mutex2;
12      volatile int boarders;
13      volatile int unboarders;
14      Semaphore boardQueue;
15      Semaphore unboardQueue;
16      Semaphore allAboard;
17      Semaphore allAshore;
18      int seats;
19      PassengerAction state;
20
21      Shared memory;
22
23      Passenger(String name, Semaphore mutex, Semaphore mutex2, Integer
        boarders, Integer unboarders,
24              Semaphore boardQueue, Semaphore unboardQueue, Semaphore
        allAboard, Semaphore allAshore, Integer seats,
25              Shared memory) {
26
27          super(name);
28
29          this.mutex = mutex;
30          this.mutex2 = mutex2;
31          this.boarders = boarders;
32          this.unboarders = unboarders;
33          this.boardQueue = boardQueue;
34          this.unboardQueue = unboardQueue;
35          this.allAboard = allAboard;
36          this.allAshore = allAshore;
37          this.seats = seats;
38          this.memory = memory;
39      }
40
41      @Override
42      public void run() {
43          while (state != PassengerAction.SATISFAIED) {
44              try {
45
46                  board();
47
48                  Thread.sleep(100);
49
50                  unboard();
51
52              } catch (InterruptedException ex) {
53                  logger.info(ex.getMessage());
54              }
```

```
55          }
56
57          logger.info("Tasks Completed: " + memory.count);
58      }
59
60      private void board() throws InterruptedException {
61          logger.info(this.getName());
62
63          boardQueue.acquire();
64
65          mutex.acquire();
66
67          boarders = boarders + 1;
68
69          memory.add();
70
71          if (boarders == seats) {
72              allAboard.release();
73              boarders = 0;
74          }
75
76          mutex.release();
77
78          state = PassengerAction.INSIDE_CAR;
79      }
80
81      private void unboard() throws InterruptedException {
82          logger.info(this.getName());
83
84          unboardQueue.acquire();
85
86          mutex2.acquire();
87
88          unboarders = unboarders + 1;
89
90          if (unboarders == seats) {
91              allAshore.release();
92              unboarders = 0;
93          }
94
95          mutex2.release();
96
97          state = PassengerAction.SATISFAIED;
98      }
99  }
```

2.0.5 Car.java

```
1  package br.ufsc.montanharussa;
2
3  import java.util.concurrent.Semaphore;
4  import java.util.logging.Logger;
5
6  public class Car extends Thread {
7
8      final static Logger logger = Logger.getLogger(Car.class.getName());
9
10     int seats;
11     volatile int boarders;
12     volatile int unboarders;
13     Semaphore boardQueue;
14     Semaphore unboardQueue;
15     Semaphore allAboard;
16     Semaphore allAshore;
17     CarAction state;
18
19     Car(String name, Integer seats, Semaphore boardQueue,
20             Semaphore unboardQueue, Semaphore allAboard,
21             Semaphore allAshore, Integer boarders, Integer unboarders) {
22         super(name);
23         this.boarders = boarders;
24         this.unboarders = unboarders;
25         this.boardQueue = boardQueue;
26         this.unboardQueue = unboardQueue;
27         this.allAshore = allAshore;
28         this.allAboard = allAboard;
29         this.seats = seats;
30     }
31
32     @Override
33     public void run() {
34         while (state != CarAction.STOPPED) {
35             try {
36
37                 load();
38
39                 Thread.sleep(5000);
40
41                 unload();
42
43             } catch (InterruptedException ex) {
44                 logger.info(ex.getMessage());
45             }
```

```
46          }
47      }
48
49      private void load() throws InterruptedException {
50          logger.info(this.getName());
51          boardQueue.release(seats);
52          allAboard.acquire();
53          state = CarAction.HIDE_ON;
54      }
55
56      private void unload() throws InterruptedException {
57          logger.info(this.getName());
58          unboardQueue.release(seats);
59          allAshore.acquire();
60          state = CarAction.WAITING;
61      }
62
63  }
```

2.0.6 CarAction.java

```
1  package br.ufsc.montanharussa;
2
3  public enum CarAction {
4      HIDE_ON, STOPPED, WAITING
5  }
```

2.0.7 PassengerAction.java

```
1  package br.ufsc.montanharussa;
2
3  public enum PassengerAction {
4      WANTING_A_RIDE, INSIDE_CAR, SATISFAIED
5  }
```

3 Questões

O primeiro problema que encontramos nessa implementação é: Como asseguramos que o carro só parte com exatamente C passageiros a bordo?[1]

No caso foi usado 2 contadores do tipo inteiro, 4 e semáforos com o tamanho C (C é igual ao número de assentos disponíveis). A solução é usar um esquema de *Count-down-and-lock* implementado com Semáforos e Inteiros em Java.

```
C <- NRO DE ASSENTOS NO CARRO

boarders = 0
unboarders = 0

boardQueue = Semaphore (C)
unboardQueue = Semaphore (C)

allAboard = Semaphore (C)
allAshore = Semaphore (C)
```

Os passageiros vão um-a-um decrementando *boardQueue* (que é um semáforo contador do número de assentos disponíveis). Toda vez que um passageiro aciona BOARDQUEUE.ACQUIRE() BOARDERS vai sendo incrementado. E no caso o carro só parte se acontecer o seguinte caso:

O último passageiro "fecha a porta"com o IF BOARDERS == SEATS. As variáveis boarders e unboarders são tipo volátil para que todas as threads sempre tenham o valor delas igual e atualizado.

```
1 (lado dos passageiros)
2 if (boarders == seats)
3     allAboard.release();
4     boarders = 0;
5
6 (lado do carro)
7 synchronized (allAboard)
8     allAboard.acquire()
```

Se essas 2 condições acontecerem, então o carro partirá com C passageiros.

[1] As questões foram retiradas do site <https://pages.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-roller.html>.

Quem é responsável por emitir o sinal para tratar esse *lock* para que outros passageiros possam se assentar no carro?

Obviamente, o carro deve fazer isso. Quando os assentos se tornam todos desocupados e prontos para correr nos trilhos novamente, o carro emite um sinal para o semáforo RELEASE(NRO DE ASSENTOS) para soltar todos os passageiros. Isso é o primeiro par de *acquire*/*release* mostrado no diagrama abaixo. O diagrama abaixo foi, refeito, seguindo a fonte <https://pages.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-roller.html>.

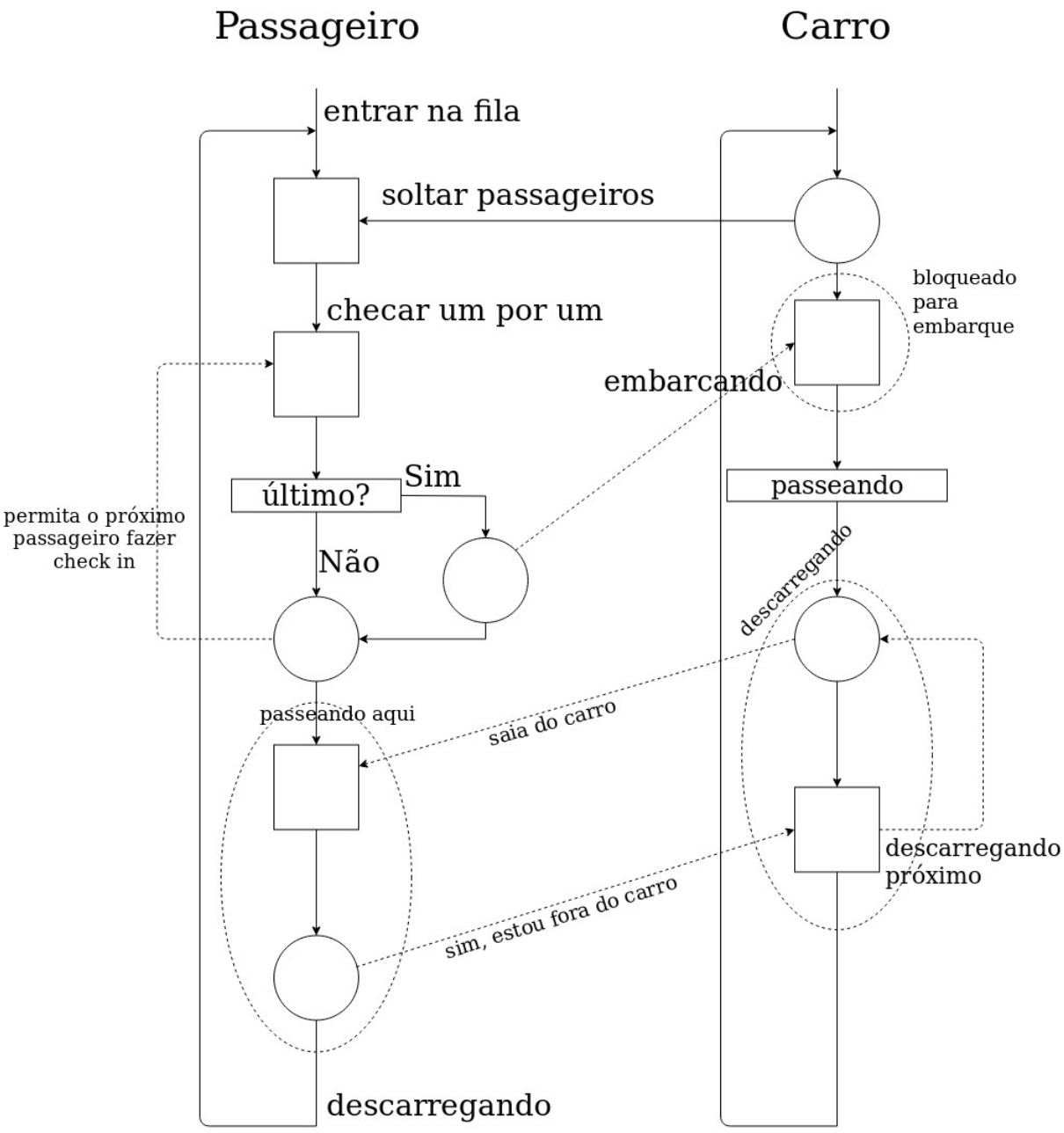

Figura 2 – Esquemático do Fluxo de Recursos sendo Disputados.

Como podemos impedir que um passageiro salte do carro antes que o carro pare e esteja seguro para o passageiro sair?

Para isso foi implementado um semáforo para o passageiro saltar quanto o carro libera o semáforo allAShore executando: ALLASHORE.RELEASE(C). Teoricamente existem problemas em liberar todos os passageiros (em código) de um vez só podendo gerar conflito entre novos passageiros entrando e passageiros saindo.

Suponha que o semáforo *allAboard* fosse removido, junto com a chamada dos passageiros *allAboard.release()* e a chamada do carro allAboard.acquire(). Porque o código remanescente falha?

Executando o código com essas mudanças fizerm com que os passageiros saíssem do carro com ele em movimento.

4 Conclusões

Discussão sobre alguns problemas envolvendo a implementação, que podem aparecer ou não.

Race condition

As variáveis contadoras **boarders** e **unboarders** estão protegidas por Semáforos binários. Assim os passageiros apenas mudam essas variáveis com proteção à região crítica. O último passageiro a embarcar irá zerar a variável de **boarders**, e o último passageiro a desembarcar irá zerar a variável **unboarders**.

Morte de passageiros

Os passageiros não "saltam do carro em movimento"por causa que o semáforo unboardQueue é apenas liberado quando o carro liberar esse semáforo (UNBOARDQUEUE.RELEASE(SEATS)). Assim, será possível que os passageiros saiam do carro. E apenas o carro executa esse comando. Passageiros apenas "competem"por executar **boardQueue/unboardQueue.wait()**. O que seria a morte de passageiros, em uma aplicação real poderia ser a simples não execução de uma tarefa qualquer, perda de pacotes, alguma falta no sistema, que pode levar à algum estado inconsistente.

```
1 (lado do carro)
2 private void unload() throws InterruptedException {
3     unboardQueue.release(seats);
4     allAshore.acquire();
5 }
```

Embarque/Desembarque

O semáforo binário mutex1 faz o papel de organizar a fila de embarque, fazendo com que n passageiros entrem primeiro (1-a-1 passageiros embarcam), e depois do passeio, outro semáforo binário mutex2 é utilizado para fazer o mesmo papel de desembarque (1-a-1 passageiros desembarcam).

Deadlock

Existe a possibilidade de haver deadlock? É possível haver deadlock se os passageiros forem implementados com um *while(true)* no método *run()* de *Thread*, gerando complicações e travando recursos sem libera-los. Algumas mudanças foram feitas, nessa implementação os passageiros podem embarcar uma vez apenas, depois disso eles ficam satisfeitos e terminam sua execução como Thread. Os objetos deles teriam que ser deletados e outros objetos passageiros teriam que ser criados para podem embarcar novamente.

Livelock

É possível haver livelock? Para haver essa situação teríamos que observar uma situação onde a JVM fica trocando o contexto de execução das threads, e elas nunca progredem (por

exemplo: nunca liberando algum recurso). Essa situação não ocorre na implementação.

Starvation

Preempção

Na verdade, a preempção aqui ajuda o progresso da implementação. Todavia, a implementação não usa espera ocupada, mas semáforos. Por usar semáforos terá que haver recursos do sistema operacional em uso para fornecer acesso à regiões críticas (i.e. variáveis contadoras "embarcados", "desembarcados", conseguir um assento).

Exclusão Mutua

Garantida pelos semáforos e semáforos binários (mutex).

Paralelismo, Speedup, etc

O aumento do número de passageiros esperando para tomar um assento altera a eficiência da execução a partir de 200 passageiros threads, não por causa de falta de memória mas por causa dos semáforos e como eles estão organizados na implementação é preciso fazer intervenções e monitoramentos na forma de código para forçar a progressão da execução com um número muito grande de passageiros. Por exemplo para aumentar para 200 passageiros é preciso aumentar o número de assentos para 50 e assim sucessivamente, isso não é um grande problema pois o número de assentos nos semáforos é um contador (não gera gastos exacerbados). Para saber mesmo se haveria diferença com o aumento do número de threads passageiro precisaríamos implementar alguma execução completa quando as threads passageiro entram no carro e vão dar uma volta na montanha russa.

Mas no caso do aumento do número de carros, aí sim há um aumento da vazão de passageiros passeando na montanha russa, ou seja, o aumento do número de carros aumenta a vazão de passageiros. Todavia o número de carros deve seguir o número de núcleos do processador, senão começa a perder eficiência com trocas de contexto e para terminar o passeio dos passageiros vai demorar mais e mais.

Referências Bibliográficas

1 DOWNEY, A. *The Little Book of Semaphores*. Green Tea Press, 2005. (Open Textbook Library). Disponível em: <https://books.google.com.br/books?id=pojKZwEACAAJ>. 12