UNIVERSIDADE FEDERAL DO PAMPA – UNIPAMPA
ENGENHARIA DE SOFTWARE

SISTEMAS OPERACIONAIS

Trabalho Prático 1

Problemas Clássicos

Alunos:
Ícaro Machado Crespo – 1801560688
Ketrin Diovana Alves Rodrigues Vargas – 1801570702

Alegrete, 13 de outubro de 2019.

1. INTRODUÇÃO

Este trabalho visa a implementação de problemas clássicos da computação no que diz respeito à *Threads*. Estas serão desenvolvidas na linguagem java e seu código se encontra no link abaixo para acesso. O objetivo é implementar os problemas: (i) produtor-consumidor com buffer limitado; (ii) jantar dos filósofos; (iii) problema da montanha russa, de maneira que seja possível solucionar o problema eficaz e com cobertura de testes para assegurar a integridade do sistema. O desenvolvimento fora realizado por Ícaro Crespo e Ketrin Vargas e vem como parte da aquisição de nota da disciplina de Sistemas Operacionais, ministrada pela Profª. Dra. Aline Vieira de Mello.

O problema do produtor-consumidor tem como prerrogativa a questão de abastecimento de produtos. A "vida" do produtor é baseada em manter populado os produtos para que o consumidor possa os utilizar. Já o problema do jantar dos filósofos consiste em todos os filósofos quererem jantar, porém necessitam de dois garfos para comer a massa. Os garfos sãos dispostos entre todos os filósofos de forma que recebam apenas um e tenham que aguardar que o filósofo ao lado largue o seu. Por fim, o problema da montanha russa é baseado em um carrinho deste brinquedo que carrega passageiros, com uma capacidade máxima, onde eles podem entrar através das diversas portas do carrinho. O carrinho inicia o trajeto assim que atingir a sua capacidade máxima.

2. DESENVOLVIMENTO

O trabalho fora desenvolvido em JAVA, onde ambos os membros desenvolveram em conjunto os três problemas impostos. Fora utilizado como ferramenta de sincronização e controle de versionamento um repositório, denominado "Problemas_Classicos", na plataforma Github.

Para o desenvolvimento do problema clássico do produtor-consumidor, foram feitas classes para o consumidor, produtor, o buffer de dados e uma main para rodar aquela parte do sistema.

Em relação ao problema do produtor-consumidor, foram utilizadas as classes Buffer, Consumidor e Produtor, além de uma classe Main para iniciar aquele escopo do sistema. Já o problema do jantar dos filósofos, implementamos, porém não obtivemos sucesso e resolvemos não entregar. No que diz respeito ao problema da montanha russa, foram elicitadas e montadas as classes Carrinho, Maquinista, MontanhaRussa, Fila e NewFila que possui a função de gerar passageiros a entrarem no brinquedo. Além disso, houve a criação de uma classe Main.

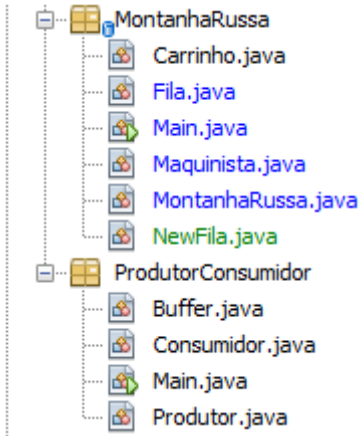

Figura 1: Distribuição das classes aos problemas do trabalho.

A estrutura do projeto consiste na divisão de testes e código fonte da solução. Os códigos estão alocados em */src*, separados por pacotes de acordo com o problema clássico. O mesmo é válido aos testes, no que diz respeito a sua separação. Sua localização é */test*. Abaixo são apresentadas figuras de 2 a 6 que demonstram os métodos e trechos específicos relevantes para desempenhar a função básica de cada problema.

```java
public void put() {
    try {
        valor = (int)(Math.random() * 20 + 1);
        System.out.println("Thread Produtor #" + n_thread + " colocando valor " + valor + " no buffer");
        buffer.add(valor);
        Thread.sleep((long)(Math.random() * 300));
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figura 2: Método de adição à Thread, produtor.

```java
public void get() {
    try {
        valor = buffer.remove();
        System.out.println("Thread Consumidor #" + n_thread + " retirando valor " + valor + " do buffer");
        Thread.sleep((long) (Math.random() * 500));
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figura 3: Método de remoção à Thread, consumidor.

```java
public class Buffer {

    private Integer inicio;
    private Integer fim;
    private Integer limite;
    private Integer buffer[];

    public Buffer(Integer limite) {
        this.inicio = 0;
        this.fim = 0;
        this.limite = limite;
        this.buffer = new Integer[limite];
    }
```

Figura 4: Classe Buffer, construtor e atributos.

Vale ressaltar que ambas as classes, Produtor e Consumidor, estendem de Thread, para poderem trabalhar sobre o mesmo conjunto de dados e em threads. A figura 4 mostra a classe Buffer, um TAD (Tipo Abstrato de Dado) criado para manipularmos um conjunto de atributos como limite, seu início e fim.

Em relação ao problema da montanha russa, decidimos adotar uma outra abordagem, como visto em uma vídeo aula, implementamos à classe o Runnable, que força a implementação do método run() que dita seu comportamento, figura 5. Utilizamos isto, pois uma thread utiliza deste recurso e ele dita sua execução. Além disso, utilizamos constantes para checar o funcionamento do sistema, novamente, sem massa de teste. Outa classe importante de ser comentada é a Maquinista que possui um funcionamento de checagem do carrinho a todo o momento para dar partida e iniciar o trajeto na montanha russa, figura 6.

```java
public MontanhaRussa(Carrinho carrinho) {
    this.carrinho = carrinho;
}
public MontanhaRussa() {
}

@Override
public void run() {
    synchronized (this) {
        try {
            Thread.sleep(tempoVoltaMontanha * 1000);
            Thread.sleep(Fila.tempoEmbarqueDesembarque * 1000 * Carrinho.numeroLugaresCarrinho);
            // utilizamos 1000 como visto em uma vídeo aula e fora um bom número
            // aqui é possível alterar o valor a ser multiplicado para ter diferentes saídas
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

Figura 5: Classe MontanhaRussa e seu método run().

```java
public Maquinista(Fila fila, Carrinho carrinho, MontanhaRussa montanha) {
    this.fila = fila;
    this.carrinho = carrinho;
    this.montanha = montanha;
}

public Maquinista(){

}

public void run() {
    System.out.println("Maquinista");
    while (true) {
        try {
            while (!fila.estaVazia() || carrinho.estaCheio()) {
                boolean entrouNoCarrinho = carrinho.add();
                if (entrouNoCarrinho) {
                    System.out.println("NOVO PASSAGEIRO");
                    System.out.println("Pessoas no carrinho: " + carrinho.getPassageiros());
                    fila.sair();
                } else {
```

Figura 6: Método construtor e trecho do run() da classe Maquinista.

3. RESULTADOS OBTIDOS

O sistema teve com princípio de testes a inserção de grandes valores para checar a amplitude dos dados e se gerariam problemas ao sistema. O sistema se comportou de maneira esperada. A garantia dada ao usuário da integridade das funcionalidades é assegurada a partir do princípio de que, no problema produtor-consumidor haverá produtos a serem retirados pelo consumidor apenas se houverem. Caso isso não ocorra, ele fica em *standby*, aguardando um produto para executar sua ação no tempo estipulado, formando assim uma fila de ações. Em relação à montanha russa,

Como dificuldades, vale ressaltar o entendimento não alcançado em sala de aula e a necessidade de sua complementação por fora, além de provas que tivemos ao longo da semana, houvera a indisponibilidade durante quase uma semana de Ícaro, isto fez com que a qualidade do trabalho caísse. Ademais, vale destacar que o jantar dos filósofos não conseguimos implementar a tempo por estas questões.

4. CONCLUSÃO

Pudemos notar no desenvolvimento dos algoritmos aspectos que complementaram as aulas ministradas. É percebido a importância da programação paralela, principalmente ao se trabalhar com Threads, já que o sistema se torna mais dinâmico e rápido, podendo alocar recursos de maneira distinta às suas semelhantes. É notório destacar que a programação paralela necessita de um cuidado maior dos programadores, ´pois podem gerar erros de dados inconsistentes com valores sendo manipulados e alterados por mais de uma Thread.

Os problemas clássicos de sincronização são, independente da tecnologia e linguagem, um desafio aos programadores que não são acostumados com a programação paralela, já que é necessário assegurar a integridade do dado a todo o momento. Este é um ponto que, ao nosso ver, não perderá seu espaço no mundo da programação, principalmente por causa do avanço tecnológico e da necessidade da sociedade em adquirir informação mais rapidamente.

Por fim, destacamos que o trabalho realizado apenas soma ao conhecimento traçado e desejado à disciplina, já que força os alunos a lidar com os diferentes problemas frente à programação paralela. Vale acrescentar que para futuros engenheiros de software é imprescindível aperfeiçoar os aspectos deste tipo de programação e ser adaptável às novas tecnologias que poderão existir.

ANEXOS

Esta seção mostra os anexos utilizados na elaboração deste documento. Serão apresentados abaixo as referências bibliográficas, lista de figuras e de links que auxiliaram o desenvolvimento do trabalho e deste relatório.

Link 1: Utilizando Threads parte 1 – Devmedia. <https://www.devmedia.com.br/utilizando-threads-parte-1/4459> Acessado em 5 de outubro de 2019.

Tanenbaum, A. S., & Machado Filho, N. (1995). Sistemas operacionais modernos (Vol. 3). Prentice-Hall.