Relatório Atividade – Problema do Produtor e Consumidor

Disciplina: Sistemas Operacionais
Professor: José Rodrigues Torres Neto

Aluno:
Cayo Cesar Lopes Mascarenhas Pires Cardoso

- **Introdução**

O problema do produtor e consumidor, também referido como Bounded Buffer Problem (BBP), consiste em um desafio de sincronização de duas ou mais threads (tarefas) concorrentes que têm acesso a um mesmo recurso do programa que está sendo executado. O fato de o recurso estar sendo compartilhado, implica na possibilidade de conflitos entre as threads que disputam pelo controle da CPU para acessar esse mesmo recurso. No caso do BBP, o recurso disputado é um buffer de memória (um vetor), de tamanho limitado/fixo, daí o motivo do nome. O produtor é o responsável por enfileirar novos jobs ou dados nesse buffer, e o consumidor assume a responsabilidade de extrair esses dados do buffer. Porém o seguinte problema poderá emergir: E se o produtor produzir muito rápido, e o consumidor não for capaz de acompanhá-lo? O programa descrito abaixo visa simular esse problema de forma que as threads não entrem em conflitos por recursos.

- **Implementação**

A implementação da atividade foi feita inteiramente em JAVA e foi dividido em 3 classes, sendo elas: produtor.java, consumidor.java e main.java. Cada classe e sua respectiva descrição está detalhada abaixo:

```
import java.util.Scanner;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class main {
   private static final int TAMANHO_BUFFER = 10;
   static final BlockingQueue<Integer> buffer = new
ArrayBlockingQueue<>(TAMANHO_BUFFER);
   static final Object lock = new Object();

   public static void main(String[] args) {
       Scanner scanner = new Scanner(System.in);

       System.out.print("Digite o tempo de produção (ms):
");
       int tempoProducao = scanner.nextInt();

       System.out.print("Digite o tempo de consumo (ms): ");
```

```
        int tempoConsumo = scanner.nextInt();

        Thread produtorThread = new Thread(new
produtor(tempoProducao));
        Thread consumidorThread = new Thread(new
consumidor(tempoConsumo));

        produtorThread.start();
        consumidorThread.start();

        // Interrompe as threads após 10 segundos
        try {
            Thread.sleep(10000); // 10 segundos
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        produtorThread.interrupt();
        consumidorThread.interrupt();

        try {
            produtorThread.join();
            consumidorThread.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        System.out.println("Programa encerrado.");
    }
}
```

O código acima descreve a implementação do código main.java, onde é implementado um buffer compartilhado. O usuário entra com o tempo de consumo e o tempo de produção das threads. As threads do produtor e do consumidor são executadas simultaneamente, sincronizando o acesso ao buffer usando um objeto de bloqueio. Após 10 segundos, as threads são interrompidas, e o programa é encerrado.

```
public class produtor implements Runnable {
    private final int tempoProducao;

    public produtor(int tempoProducao) {
        this.tempoProducao = tempoProducao;
    }

    @Override
    public void run() {
        try {
            while (true) {
```

```
                int item = (int) (Math.random() * 100);

                synchronized (main.lock) {
                    while (main.buffer.remainingCapacity() ==
0) {
                        System.out.println("Buffer cheio,
produtor esperando...");
                        main.lock.wait();
                    }
                    main.buffer.put(item);
                    System.out.println("Produzido: " + item);
                    main.lock.notifyAll();
                }

                Thread.sleep(tempoProducao); // Tempo de
produção fornecido pelo usuário
            }
        } catch (InterruptedException e) {
            System.out.println("Produtor Interrompido");
            Thread.currentThread().interrupt();
        }
    }
}
```

O código acima referencia a parte da implementação do produtor. A classe produtor.java implementa a interface Runnable, o que permite que seja executada como uma thread. No codigo é repassado o valor do tempoProducao que o usuario definiu na execução da classe main.java. A classe representa uma thread que produz itens aleatórios e os coloca no buffer compartilhado. Se o buffer estiver cheio, o produtor espera até que haja espaço disponível através da linha “main.lock.wait()” e ao verificar que há espaço disponível novamente é utilizado o “main.lock.notifyAll()” para notificar todas as threads esperando no objeto de bloqueio que o estado mudou (ou seja, o buffer não está mais cheio) e continuar adicionando objetos no buffer.

```
public class consumidor implements Runnable {
    private final int tempoConsumo;

    public consumidor(int tempoConsumo) {
        this.tempoConsumo = tempoConsumo;
    }

    @Override
    public void run() {
        try {
            while (true) {
                synchronized (main.lock) {
                    while (main.buffer.isEmpty()) {
                        System.out.println("Buffer vazio,
```

```
consumidor esperando...");
                        main.lock.wait();
                    }
                    int item = main.buffer.take();
                    System.out.println("Consumido: " + item);
                    main.lock.notifyAll();
                }

                Thread.sleep(tempoConsumo); // Tempo de
consumo fornecido pelo usuário
            }
        } catch (InterruptedException e) {
            System.out.println("Consumidor Interrompido");
            Thread.currentThread().interrupt();
        }
    }
}
```

O código acima referencia a parte do consumidor, que é similar ao do produtor, onde ele vai consumir os itens do buffer, caso o buffer por algum motivo esteja vazio ele utiliza a mesma função do produtor pra parar a thread e depois é notificado quando tiver itens no buffer pra ela voltar a executar.

- **Resultados**

A execução do programa inicia com a entrada de dados de tempo de consumo e produção que o usuario vai escolher.

```
Digite o tempo de produção (ms): 100
Digite o tempo de consumo (ms): 50
```

Após a escolha dos tempos o programa inicia e roda por exatos 10 segundos até ele encerrar automaticamente. Nesse caso de testes utilizando o tempo de produção maior que o tempo de consumo a thread de consumo vai ter que ser parada algumas vezes ja que ela consome os itens em menos tempo do que eles são produzidos. Segue abaixo um trecho dos resultados pra esse caso.

```
Produzido: 92
Consumido: 92
Buffer vazio, consumidor esperando...
Produzido: 35
Consumido: 35
Buffer vazio, consumidor esperando...
Produzido: 35
Consumido: 35
Buffer vazio, consumidor esperando...
Produzido: 80
Consumido: 80
Buffer vazio, consumidor esperando...
Produzido: 79
Consumido: 79
```

Observe que ao produzir um item ele já é consumido e a thread do consumidor tem que esperar até que outro seja produzido.

Abaixo é mostrado um caso de teste onde acontece o inverso, a thread de produção leva pouco tempo pra encher o buffer e precisa ficar esperando a thread consumidora ir consumindo os itens para produzir mais.

```
Digite o tempo de produção (ms): 25
Digite o tempo de consumo (ms): 260
Produzido: 80
Consumido: 80
Produzido: 47
Produzido: 44
Produzido: 73
Produzido: 46
Produzido: 19
Produzido: 98
Produzido: 97
Produzido: 50
Produzido: 85
Produzido: 25
Consumido: 47
Produzido: 95
Buffer cheio, produtor esperando...
Consumido: 44
Produzido: 23
```

Diante dos casos de teste, é possível observar que o problema do produtor e consumidor é facilmente solucionável se houver um controle correto de recursos, não deixando as threads produtoras e consumidoras acessarem o buffer de forma desordenada ou conflitante. A implementação de um mecanismo de sincronização adequado, como o uso de objetos de bloqueio e métodos wait() e notifyAll(), permite que as threads cooperem entre si e coordenem o acesso ao buffer de forma segura. Ao garantir que o produtor espere quando o buffer estiver cheio e que o consumidor espere quando o buffer estiver vazio, evitamos condições de corrida e garantimos que o sistema funcione de maneira consistente e sem problemas de concorrência. Portanto, com um controle adequado de recursos e uma boa estratégia de sincronização, é possível resolver eficientemente o problema do produtor e consumidor.