Primeiro Trabalho Prático

Lucas Horta Monteiro de Castro - 16.1.8073
Manoel Farias Paixão Júnior - 17.2.5966
Matheus Nunes Martins de Barros - 17.1.8328
Rosabel Vieira Braga - 17.2.5909

[1]Instituto de Ciências Exatas e Aplicadas (ICEA) – Universidade Federal de Ouro Preto
CEP: 35.931-008 – João Monlevade – MG – Brasil

Abstract. *The objective is to apply the concepts of topics and synchronization of topics / processes seen in the discipline of Operating Systems. Thus, a simulation of a simplified production environment was implemented, in order to explain the correct use of the concepts and maximize the available processing resources.*

Resumo. *O objetivo é aplicar os conceitos de threads e sincronismo de threads /processos vistos na disciplina de Sistemas Operacionais. Dessa forma, foi implementado uma simulação de um ambiente de produção simplificado, afim de explicitar o uso correto dos conceitos e maximizar os recursos de processamento disponíveis.*

1. Introdução

O trabalho consiste na simulação de um ambiente simplificado de produção, aplicando-se threads. Para isso, é exposto a entidade genérica fábrica, que possui máquinas e funcionários. As máquinas são homogêneas, ou seja, com as mesmas especificações. Os funcionários são responsáveis por requisitar as tarefas que as máquinas executam. Nesse cenário, é notório a existência de uma fila para adicionar as tarefas requisitadas. Mediante disso, deparamos com inúmeras situações reais, onde vale ressaltar diferentes abordagens e resoluções para tais.

2. Introdução ao Algoritmo

O algoritmo foi implementado em Java, sendo modulado em duas partes. A primeira atribui as entradas de dados, formada por arquivos de texto distintos de cada funcionário. Já a segunda, refere-se as classes **Fila**, **Funcionário**, **Máquina** e a **Main**, que será destrinchadas e analisadas de forma minuciosa.

2.1. Entrada de Dados

Foi implementado cinco arquivos no formato de texto (.txt) para leitura ao longo do desenvolvimento. Como conteúdo interno, temos um identificador único referente a funcionário e seguido por um conjunto de tarefas para execução. As tarefas são identificadas por um nome e pelo seu tempo de execução (em milissegundos). Observe o arquivo **fun3.txt**, como exemplo:

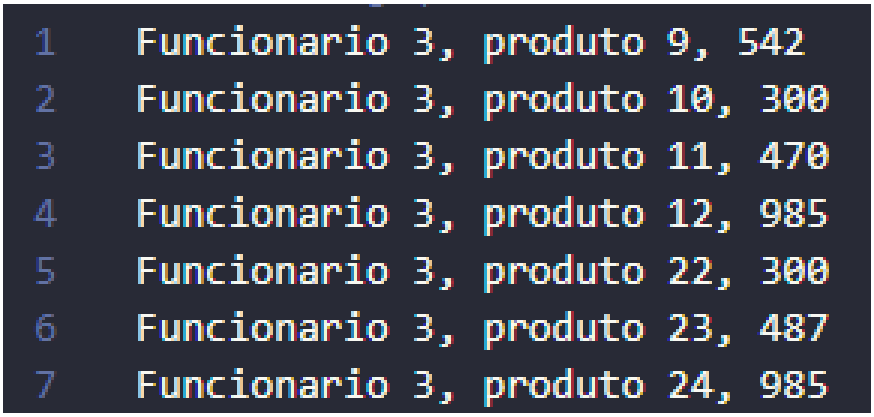


Figura 1. Especificação do código do funcionário, produtos e tempo de execução.

A solicitação das informações, juntamente com a sua leitura também alimentam o algoritmo para melhorar o seu desempenho e sua usabilidade. Portanto, é realizado de forma sucinta, perguntas relacionadas a capacidade da fila de tarefas, o identificador do funcionário e da máquina, como também a quantidade de máquinas. Podemos observar a análise na **figura 2**. Os valores de entrada foram todos **1**, afim de enfatizar apenas as perguntas.

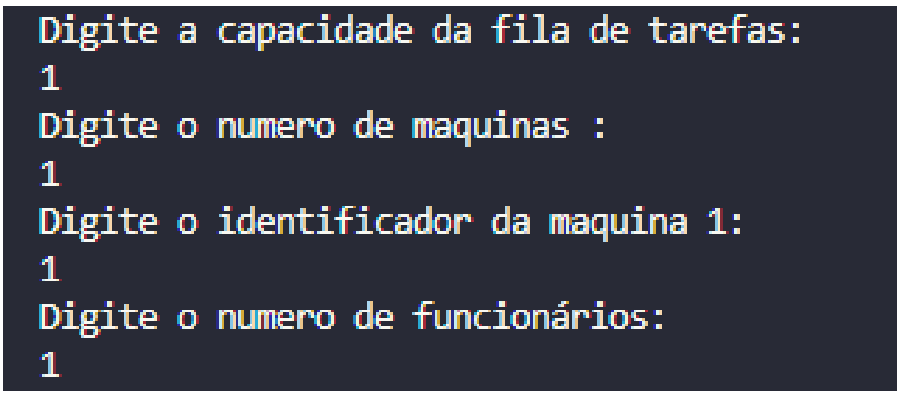


Figura 2. Entrada de dados inseridas pelo usuário.

3. Classes

Devido a natureza da simulação, foram construídas quatro classes bem definidas, sendo elas Fila, Funcionário, Máquina e a Main. Com o nome intuitivo a referência de cada classe, estaremos destrinchadas-as para melhor compreensão. O código foi devidamente comentado para auxiliar o leitor.

3.0.1. Funcionário

A entidade **Funcionario** importa a biblioteca **Java.io**, chamandos os métodos **BufferedReader**, **File**, **FileRead**, e **IOException**. Dessa forma, construímos a **public class Funcionario extends Thread**, usando o conceito de herança para receber os atributos e métodos da superclasse **Thread**. Segue-se instanciando as variáveis necessárias para

a implementação. Observe abaixo a descrição das importações e as variáveis dentro do algoritmo:

- **fila**, referente a classe **Fila**.
- **tarefas**, vetor de tamanho vinte.
- **cont**, contador de inteiros.
- **id**, referente ao identificador do funcionário.
- **end**, variável boolean responsável pelo status da fila.

```
1  import java.io.BufferedReader;
2  import java.io.File;
3  import java.io.FileReader;
4  import java.io.IOException;
5
   You, 14 minutes ago | 1 author (You)
6  public class Funcionario extends Thread {
7      Fila fila;
8      String [] tarefas = new String [20];
9      int cont = 0;
10     int id = 0;
11     boolean end;
```

Figura 3. Importações e criação das variáveis.

Adiante, no método **public Funcionario(Fila buffe, String txt, int x)**, recebemos o texto com as tarefas requeridas. Após receber o texto, é necessário indicar qual arquivo será lido, pois dessa forma, podemos criar um objeto chamado **bufferedReader** para armazenar temporariamente os dados enquanto eles estão sendo movidos.

É realizado um loop para realizar a leitura do arquivo de texto linha a linha. Após fechar a leitura, podemos liberar o fluxo do objeto ou fechar o arquivo. Vale ressaltar o uso da extensão **IOException** para o tratamento de erros de leitura e gravação.

Com a captura e o uso do método **printStackTrace()**, caso ocorra algum erro, temos o retorno direto no console para análise. Observe abaixo a descrição do método no algoritmo:

Seguindo a linha de raciocínio, temos o método **public void run()**, sem retorno e responsável integralmente por iniciar a **Thread**. A implementação é realizada dessa forma, com finalidade de instanciar a classe diretamente na **Main** com o comando **.start**, que será apresentado.

Após essa ação na **Main**, conseguimos inserir as **n** tarefas dentro da fila. Subsequente, é imprimido o código do funcionário e o status da tarefa. Por fim, é criado uma função **public boolen** com o nome **Verificar**, a fim de retornamos a variável **end**, que foi instanciada no início, informando se o status é verdadeiro ou falso. Observe abaixo a descrição do método do algoritmo:

```java
public Funcionario(Fila buffe, String txt, int x) { // recebendo o txt com as tarefas
    fila = buffe;
    id = x ;

    File dir = new File("src/txt");
    File arq = new File(dir, txt);

    try {
        FileReader fileReader = new FileReader(arq);//O arquivo que será lido
        BufferedReader bufferedReader = new BufferedReader(fileReader); //Criação o objeto bufferReader
        String linha = " ";

        while ( ( linha = bufferedReader.readLine() ) != null) {//Loop linha a linha no arquivo.
            tarefas [cont] = linha;
            ++cont;
        }

        fileReader.close(); //Liberação o fluxo dos objetos ou fechamento do arquivo
        bufferedReader.close();

        }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figura 4. Método public Funcionario (Fila buffe, String txt, int x)

```java
    public void run() {
        for (int i = 0 ; i < cont; i++) {
            fila.inserir(tarefas[i]);
        }

        System.out.println("FUNCIONARIO "+id+" FINALIZADO !");
        end = true;
    }

    public boolean Verificar() {
        if (end == true) {
            return true;
        }
        return false;
    }
}
```

Figura 5. Método public void run() e public boolean Verificar()

3.0.2. Máquina

Construímos a **public class Maquina extends Thread**, usando o conceito de herança para receber os atributos e métodos da superclasse **Thread**, assim como **Funcionario**. É instanciado as variáveis necessárias da classe **Maquina** para a implementação. Observe abaixo a descrição das variáveis do algoritmo:

- **id**, string referente ao identificador da máquina.
- **item**, objeto referente a classe Fila.

```java
public class Maquina extends Thread {
    String id;
    Fila item;
```

Figura 6. Variáveis da classe Máquina.

Após instanciar as variáveis, estaremos criando o seu construtor para receber a **fila**

e o identificador da máquina, sendo a variável **idMaquina**. Acompanhando o raciocínio do método **public void run()** na classe **Maquina**, temos a mesma ideia de implementação do método **public void run()** na classe **Funcionario**.

A partir do momento em que a ação será executada na **Main**, conseguimos inserir uma variável **aux** do tipo **Objeto**, que foi criado na classe **Fila** para percorrer o vetor na posição desejada da máquina, sendo assim, conseguimos remover o item da fila. Observe abaixo a descrição do construtor e do método do algoritmo:

```
6       public Maquina(String idMaquina,Fila fila){
7           id = idMaquina;
8           item = fila;
9
10      }
11
12      public void run() {
13          Object aux;
14          while ((aux = item.remover()) != null ) {
15              System.out.println(id +","+ aux);
16          }
17      }
18
19  }
```

Figura 7. Criação do construtor e método public void run().

3.0.3. Fila

A entidade **Fila** remete a fila de tarefas, sendo implementada dentro da **public class Fila**. Dessa forma, instanciamos as variáveis necessárias, sendo todas privadas, diferente das variáveis encontradas nas outras classes. Subsequente, temos o construtor com o nome **public Fila(int tamCap)**, recebendo como parâmetro a variável **tamCap**, responsável pela capacidade da fila de tarefas. Observe abaixo a descrição das variáveis e do construtor do algoritmo:

- **cont**, variável global referente ao contador de inteiros da classe.
- **entrada**, referente ao valor inteiro de entrada.
- **saida**, referente ao valor inteiro final.
- **buffer**, variável do tipo Objeto.
- **capacidade**, variável inteira relacionada a capacidade da fila.

Agora precisamos inserir as tarefas na fila para contabilizar e realizar futuramente as Threads. Dessa forma, usamos o método **public synchronized void inserir(Object item)** para a inserção. Constituido por um laço de repetição **while**, será verificado se a tarefa será inserida ou não, dependendo do status da fila, pois a mesma pode encontrar-se vazia ou cheia. Enquanto a variável **cont** for igual a capacidade da fila, então o **while** estará executando.

```java
public class Fila{
    private int cont , entrada, saida;
    private Object[] buffer;
    private int capacidade = 0;

    public Fila (int tamCap) {
        cont = 0;
        entrada = 0;
        saida = 0;
        buffer = new Object[tamCap];
        capacidade  = tamCap;
    }
```

Figura 8. Variáveis e construtor public Fila (int tamCap).

Consequentemente, temos a thread entrando no laço de repetição e seu status em espera, sinalizado pelo comando **wait()**, travando a thread até que a variável **cont** seja igual a capacidade da fila. Quando a variável **cont** torna-se diferente da capacidade da fila, a thread deixa o laço de repetição e insere a tarefa na fila, ocorrendo o mesmo processo sucessivamente nas entradas futuras.

Quando uma thread invoca **wait( )**, ela libera o lock do objeto, seu estado é definido como bloqueado e a thread é colocada no conjunto de espera do objeto, conhecido como **wait set**. Já o **notify()**, quando é invocado pela **thread**, temos a seleção de um **thread** qualquer dentro do conjunto X, onde ela é movida da espera para o conjunto de entrada, e breve, o estado da **thread** é definido como executável.

É de suma importância entender que caso a fila chegue no final do buffer, ela voltará a zero, como explicitado no comentário. Observe abaixo a descrição do método e o uso das invocações **wait()** e **notify()**:

```java
public synchronized void inserir(Object item) {
    while (cont == capacidade) {
        try {
            wait();
        }
        catch(Exception e) {}
    }

    ++cont;
    buffer[entrada] = item;
    entrada = (entrada + 1) % capacidade; // caso a fila chegue no final do buffer ele volta a 0

    notify();

}
```

Figura 9. Método public synchronized void inserir(Object item)

Após adicionar, precisamos remover as tarefas da fila. Mediante isso, a necessidade da criação do método **public synchronized Object remover()**. O funcionário requisita a fila e a máquina à remove. A máquina consegue remover os itens da fila quando o laço de repetição compara o valor do **cont == 0**, sinalizando que a fila encontra-se vazia.

Assim que inserido algo, o **cont** sai de zero e a tarefa é removida da fila, remetendo

a mesma ideia evidenciada com o uso do **wait()** e **nofity()** no método **public synchronized void inserir(Object item)**. Observe abaixo a descrição do método do algoritmo:

```
30      public synchronized Object remover() {
31          Object item;
32
33          while(cont == 0) {
34              try{
35                  wait();
36              }
37              catch(Exception e) {}
38          }
39
40          --cont;
41          item = buffer[saida];
42          saida = (saida + 1) % capacidade;
43
44          notify();
45
46          return item;
47
48      }
```

Figura 10. Método public synchronized Object remover()

Por fim, é criado um método **public boolean Verificar()**, a fim de retornarmos a variável **cont**, que foi instanciada no início. Seu status pode ser verdadeiro ou falso. Observe abaixo a descrição do método do algoritmo:

```
50      public boolean Verificar() {
51          if(cont == 0) {
52              return  true;
53          }
54
55          return false;
56
57      }
```

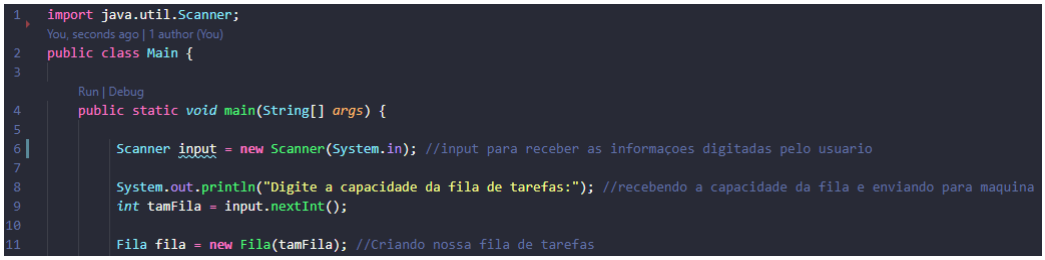
Figura 11. Método public boolean Verificar()

3.0.4. Main

A classe **Main** inicia instanciando a variável **input**, utilizando o método **Scanner** da biblioteca **Java.util.Scanner**. Ele é útil para receber as informações vindas do usuário, ou seja, a entrada de dados fornecidas pelo usuário.

Conforme o enunciado do trabalho solicita, é elucidado a capacidade da fila de tarefas para o usuário. Após salvar a informação no **tamFila**, instanciamos a Fila, enviando

o **tamFila**, variável de argumento do construtor na classe **Fila**, já explicidado. Observe abaixo a descrição no algoritmo:



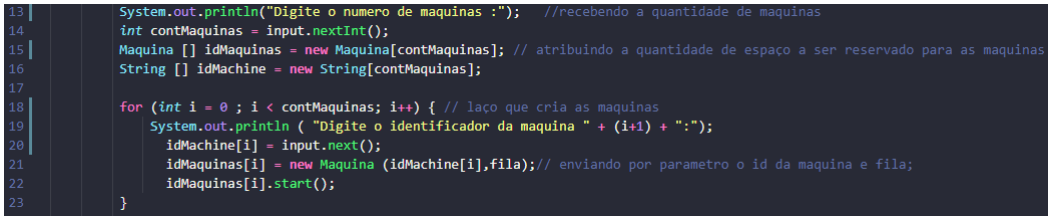
```java
import java.util.Scanner;
You, seconds ago | 1 author (You)
public class Main {

    Run | Debug
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in); //input para receber as informaçoes digitadas pelo usuario

        System.out.println("Digite a capacidade da fila de tarefas:"); //recebendo a capacidade da fila e enviando para maquina
        int tamFila = input.nextInt();

        Fila fila = new Fila(tamFila); //Criando nossa fila de tarefas
```

Figura 12. Instância do Input e entrada do tamanho da fila de tarefas)

Com a fila pronta, iniciamos com o usuário informando a quantidade de máquinas disponíveis. É adicionado o valor na variável **contMaquinas**. Sabendo a quantidade de máquinas, é criado dois vetores, sendo eles **idMaquinas[]** e **idMachine[]**, tendo o tamanho igual a quantidade de máquinas. Eles encontram-se vazios, porém, com o tamanho já definido.

Cada máquina possui um identificado, que será informado pelo usuário. Tratamos a quantidade de máquinas dentro da condição **for**. É inserido a informação do identificador da máquina dentro do vetor, pois com a inicialização deles, é adicionado o identificador ao vetor **idMachine[]**.

Logo, é enviado como parâmetro o identificador junto com a fila, ao método **public void run()**. Imediatamente, é executado o método **public void run()** dentro da classe **Maquina**, através do método presente na **Main**, chamado **idMaquinas[i].start()**. Observe abaixo a descrição do algoritmo:



```java
        System.out.println("Digite o numero de maquinas :");   //recebendo a quantidade de maquinas
        int contMaquinas = input.nextInt();
        Maquina [] idMaquinas = new Maquina[contMaquinas]; // atribuindo a quantidade de espaço a ser reservado para as maquinas
        String [] idMachine = new String[contMaquinas];

        for (int i = 0 ; i < contMaquinas; i++) { // laço que cria as maquinas
            System.out.println ( "Digite o identificador da maquina " + (i+1) + ":");
              idMachine[i] = input.next();
              idMaquinas[i] = new Maquina (idMachine[i],fila);// enviando por parametro o id da maquina e fila;
              idMaquinas[i].start();
        }
```

Figura 13. Entrada do identificador da máquina e método .start()

Podemos observar que a quantidade de threads é a quantidade de máquinas inserida pelo usuário. O método **.start()** realiza um jump para a classe **Maquina** , exatamente no método **public void run()**. Em sequência, é realizado outro jump para a classe **Fila**, exatamente no **wait()**.

O usuário é requisitado novamente, para informar a quantidade de funcionários que estarão inserindo as tarefas na fila, salvo pela variável **contFuncionario**. O vetor **qtdFunc[]** é criado com a quantidade de funcionários informada.Observe abaixo a descrição do algoritmo:

```
25          System.out.println("Digite o numero de funcionários:");
26          int contFunc = input.nextInt();
27          Funcionario [] qtdFunc = new Funcionario[contFunc];
28
29          for( int x = 0 ; x < contFunc; x++) { //laço que cria os funcionarios
30              qtdFunc[x] = new Funcionario (fila,"func"+(x+1), x+1);// enviando por paramentro a fila, o arquivo txt e o id do funcionario
31              qtdFunc[x].start();
32          }
```

Figura 14. Entrada do identificador do funcionário e método .start()

Podemos observar que a quantidade de threads é a quantidade de funcionários inserida pelo usuário. O método **.start()** realiza um jump para a classe **Funcionario**, exatamente no método **public void run()**. Em sequência, é realizado outro jump para a classe **Fila**, exatamente no **wait()** e posteriormente para a classe **Maquina**.

Por fim, o funcionário após inserir as tarefas na fila, finaliza suas atividades. Dessa forma, temos um retorno da verificação igual a true. A máquina, por sua vez, não consegue finalizar como o funcionário, pois ela sempre estará presa no **while(cont == 0)** do método **public synchronized Object remover()** da Fila.

A máquina não finaliza a atividade pelo fato do valor referente a exclusão não chegar em zero. Dessa forma, precisamos de duas condições para finaliza-la, sendo elas, a fila encontrar-se vazia e o funcionário ter finalizado a atividade. A partir dessas duas condições, conseguimos finalizar a atividade da máquina. Observe abaixo a descrição do método do algoritmo:

```
34          boolean aux = false;
35
36          while (aux == false) {
37              if(fila.Verificar() == true && qtdFunc[contFunc-1].Verificar() == true ) {
38                  for(int y=0 ; y < contMaquinas; y++) {
39                      System.out.println(idMachine[y]+" FINALIZADA !");
40                  }
41                  aux = true;
42              }
43          }
```

Figura 15. Condição tratando a finalização da atividade da máquina.

4. Resultados de Entrada

Estaremos mantendo os valores de entrada a fim de analisar a diferença entre os resultados, para realizar o mapeamento das threads de usuário e observar principalmente a questão do sincronismo, ou seja, o processamento simultâneo de várias threads dado que possuimos mais de um processador. Observe no dois casos, a diferença entre as saídas para a mesma entradas:

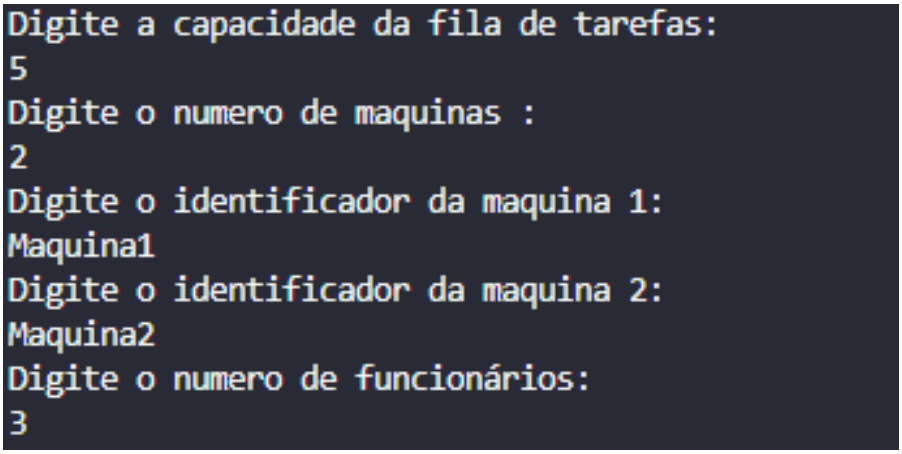


Figura 16. Entrada de Dados.

No exemplo acima, foi informado pelo usuario que a capacidade da fila de tarefas seria de 5 unidades, seguido da quantidade de maquinas disponiveis para fazer a retirada das tarefas da fila, que neste caso foram 2 unidades, em seguida é necessario informa o ID de cada maquina, para só então informa a quantidade de funcionarios do programa.

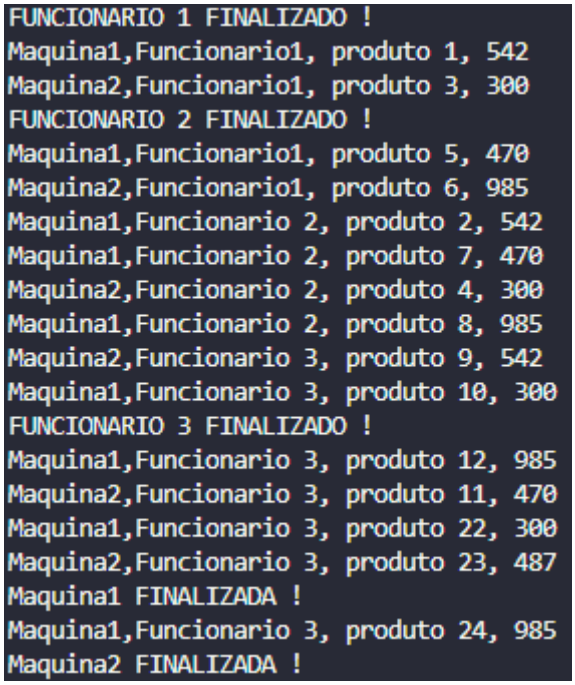


Figura 17. Primeira saída de dados.

Resultado de impressão na primeira vez que o programa foi executado, com a finalização dos funcionarios e das maquinhas depois de finalizar todas as tarefas.

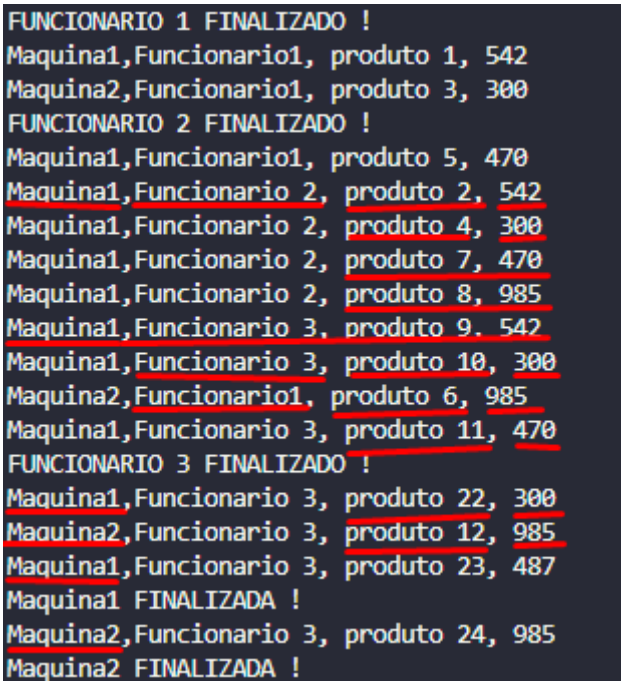


Figura 18. Segunda saída de dados.

5. Conclusão

As threads se comportam de forma independente.Elas rodam concorrentemente num mesmo processo, e os processos executam concorrentemente num sistema operacional.O uso de threads é interessante quando quer executar pelo menos duas coisas ao mesmo tempo em um programa para tirar vantagem da múltiplas CPUs ou ainda para evitar que o programa inteiro fique travado ao executar uma operação demorada.

O caso de uso mais comum de threads no Java é para atender requisições em aplicações web. Se você está de alguma forma familiarizado com Servlets, Spring MVC, JSF, Struts ou algum outro framework web Java, deve saber que todos eles atendem cada requisição HTTP em uma thread diferente. Isso permite atender vários usuários simultaneamente e ao mesmo tempo ter um certo isolamento das informações, pois o servidor de aplicação (como o Tomcat ou JBoss) associa os dados de cada requisição com a respectiva thread, então isso faz com que o mesmo código seja executado por todos os usuários, mas cada com com informações isoladas umas das outras.