

**Bacharelado em Ciências da Computação**

*BC1501 – Programação Orientada a Objetos*



**Aula 06**

**Material adaptado do  
Prof. André G. R. Balan**

# Tópicos da aula

- ▶ Programação paralela utilizando **Threads**
  - Introdução
  - Criação e execução
  - Estados de uma Thread
  - Sincronização
  - Modelo produtor/Consumidor

# Introdução

- ▶ Um sistema computacional realiza frequentemente operações em “paralelo”:
  - Navegar na internet
  - Reproduzir músicas, vídeos
  - Download de arquivos
  - Processamento de texto...
- ▶ **Apenas os computadores com mais de um processador podem, de fato, realizar mais de uma tarefa ao mesmo tempo**
  - Cada processador ficará responsável por um ou mais processos

# Introdução

- ▶ Em computadores de um único processador, os sistemas operacionais utilizam de técnicas para simular o processamento paralelo
  - *Time-Sharing*
  - Escalonamento

# Introdução

- ▶ Em programação:
  - Até o momento, estudamos programação de modo linear, ou seja:
    - Definimos operações/comandos um a um
    - Uma operação só é executada depois que a operação anterior foi concluída
  - Na programação não-linear, um programa pode manter múltiplas **linhas de execução** que serão responsáveis por tarefas distintas. Ex: Um browser da internet
    - Uma linha de execução está fazendo o download de um vídeo no YouTube
    - Outra linha de execução é responsável por exibir a parte do vídeo que já foi puxada

# Introdução

- ▶ Outro exemplo:
  - Em um programa de bate papo:
    - Vários usuários estão escrevendo e enviando mensagens ao mesmo tempo
    - O programa pode manter uma linha de execução para interagir com cada usuário
  - Em um gerenciador de download:
    - Temos várias linhas de execução, uma para cada fração do arquivo.

# Introdução

- ▶ Em programação é possível definir várias linhas de execução por meio do mecanismo denominado **Thread** (fio, barbante, linha...)
- ▶ Com as threads, o programa pode se bifurcar. A principais vantagens são:
  - Aumento na velocidade de processamento. Em um computador com vários processadores!
  - Capacidade de desenvolver programas que interagem com diversos dispositivos e/ou pessoas ao mesmo tempo
    - Ex: Sala de bate-papo, Servidores de impressão, Browsers

# Introdução

- ▶ O conceito de threads está disponível em diversas linguagens, porém de maneira diferente:
  - Na maioria das linguagens (incluindo C, C++), existem bibliotecas específicas para cada SO para se criar threads.
    - Desvantagem: **Falta de portabilidade**, pois as bibliotecas são específicas para cada sistema operacional
  - Na linguagem JAVA: a programação com Threads é feita por meio de suas bibliotecas. Existem classes para Threads assim como existem classes para String, Arrays, etc...
    - A **JVM** faz a intermediação do gerenciamento das Threads junto ao sistema operacional em uso
    - Como existe uma JVM para cada SO, é garantida a portabilidade do programa.

# Introdução

- ▶ Programando com várias threads.
  - Escrever programas com várias threads pode ser um pouco mais difícil para a maioria das pessoas.
  - Isto deve-se ao fato de que nosso cérebro consegue com dificuldade executar tarefas em paralelo:
    - Estudar para duas provas ao mesmo tempo
    - Cantar uma música enquanto ouvimos outra
    - Jogar xadrez e ouvir música (jogar xadrez exige concentração, ou seja, exclusividade do cérebro)
  - Porém, ao pegar a prática, torna-se útil, pois podemos criar programas mais complexos.

# Introdução

- ▶ Uma Thread parece e age como um programa individual.
- ▶ Em Java, Threads são criadas por meio de objetos.
- ▶ Cada Thread tem sua própria stack! Aprox 256kb (Java)

# Criando Threads

- ▶ *O programador deve criar uma classe herdeira da classe Thread e sobrescrever o método run()*
- ▶ *O método run() deve conter um loop que irá rodar durante tempo de vida do thread.*
- ▶ *Quando o run(), terminar a execução, a thread “morre”.*

# Herdando da classe Thread

```
public class Trabalhador extends Thread {  
    String nome, produto;  
    int tempo;  
  
    public Trabalhador(String nome, String produto, int tempo) {  
        this.nome = nome;  
        this.produto = produto;  
        this.tempo = tempo;  
    }  
    ...
```

# Herdando da classe Thread

```
...
@Override
public void run() {
    for (int i=0; i<50; i++) {
        try {
            Thread.sleep(tempo);
        } catch (InterruptedException e) {
            break;
        }
        System.out.println(nome + " produziu a " + i + "º " + produto);
    }
}
```

# Iniciando threads herdeiras da classe Thread

```
public static void main(String[] args) {  
  
    Trabalhador mario = new Trabalhador("Mario", "bota", 100);  
    Trabalhador sergio = new Trabalhador("Sergio", "camisa", 200);  
  
    mario.start();  
    sergio.start();  
}
```

# Iniciando threads herdeiras da classe Thread

## Saída do programa:

Mario produziu a 1ºbota

Mario produziu a 2ºbota

Sergio produziu a 1ºcamisa

Mario produziu a 3ºbota

Mario produziu a 4ºbota

Sergio produziu a 2ºcamisa

Mario produziu a 5ºbota

Mario produziu a 6ºbota

Sergio produziu a 3ºcamisa

Mario produziu a 7ºbota

Mario produziu a 8ºbota

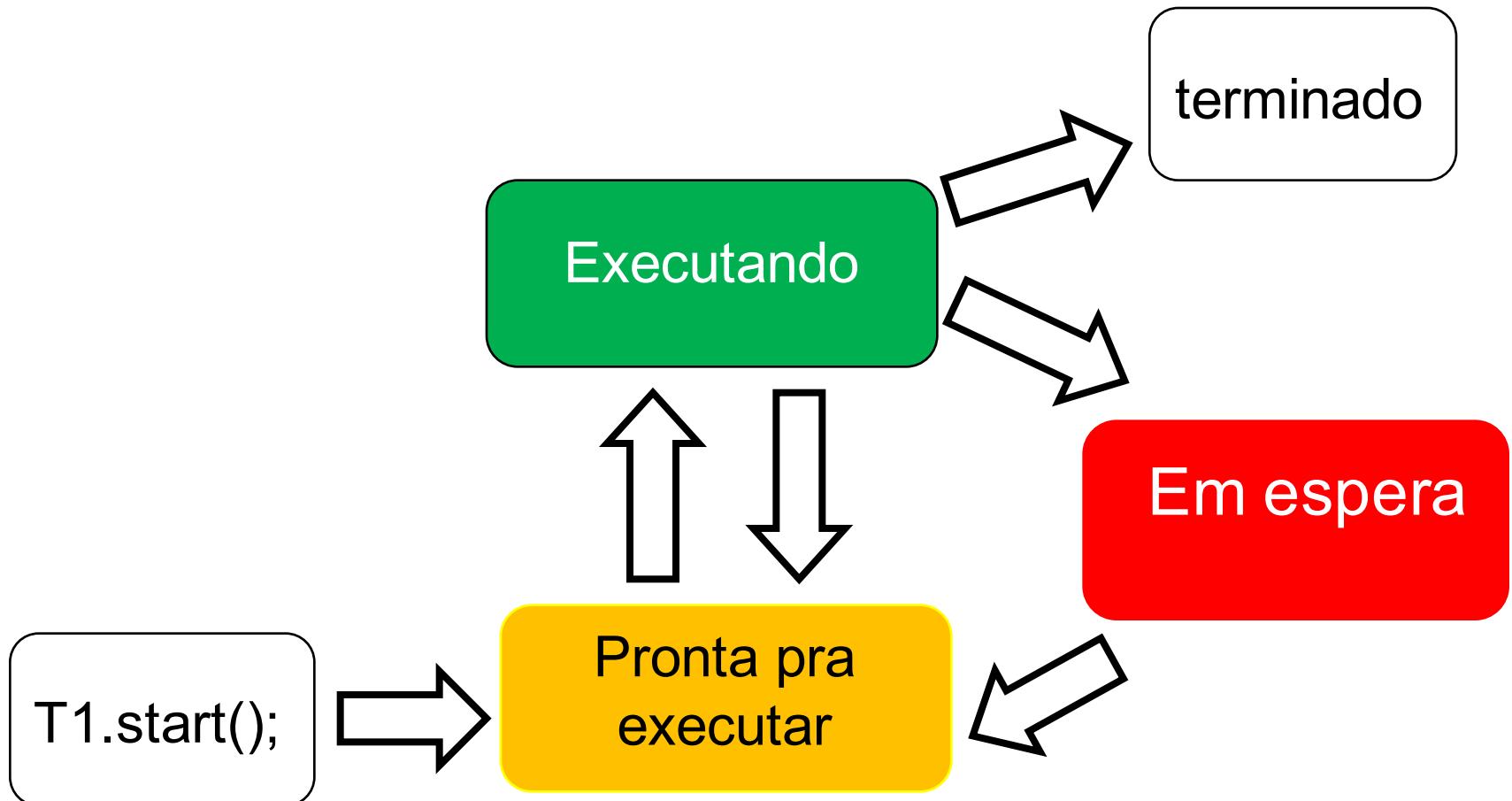
Sergio produziu a 4ºcamisa

Mario produziu a 9ºbota

Mario produziu a 10ºbota

# Estados de uma Thread

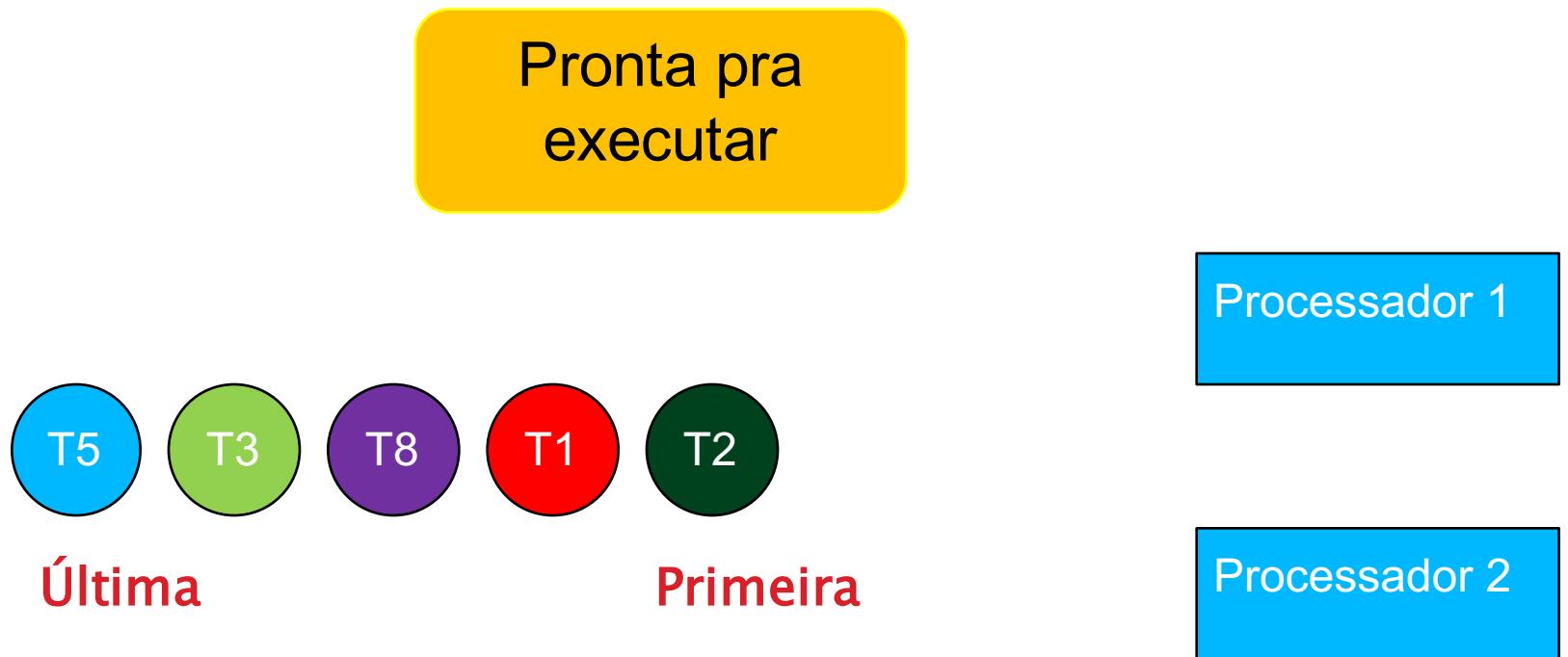
- Uma thread está geralmente em um dos três estados abaixo:



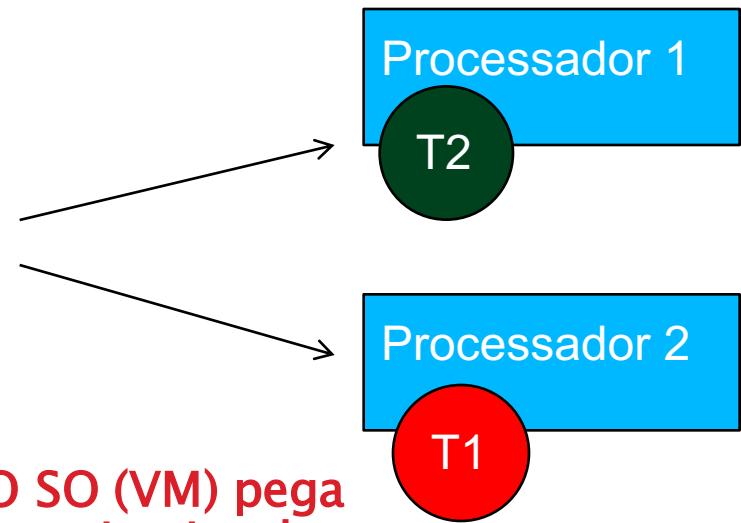
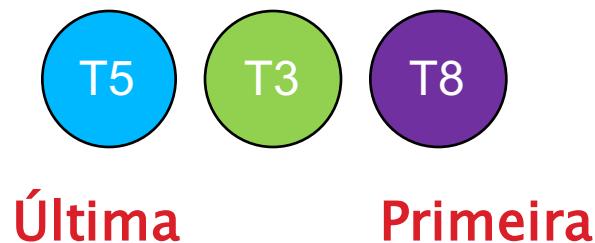
# Estados de uma Thread

- ▶ A thread inicia-se com a chamada do método **start**, que é um método da classe Thread
- ▶ Ao iniciar, a Thread vai direto para o estado “**Pronta para executar**”. Quando a thread está nesse estado, significa que ela foi colocada em uma **fila de execução** de threads:

# Estados de uma Thread

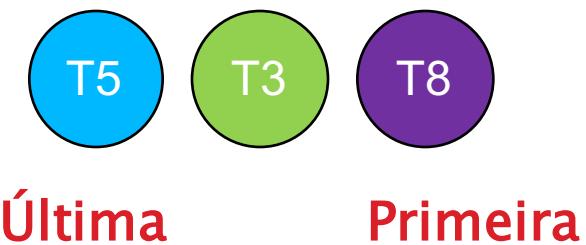


# Estados de uma Thread



O SO (VM) pega  
a primeira da  
Fila e lhe atribui  
um processador,  
para que ela seja  
executada:  
“Despacho”

# Estados de uma Thread

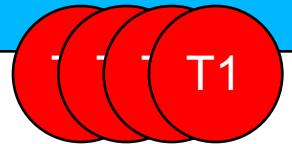


Executando

Processador 1

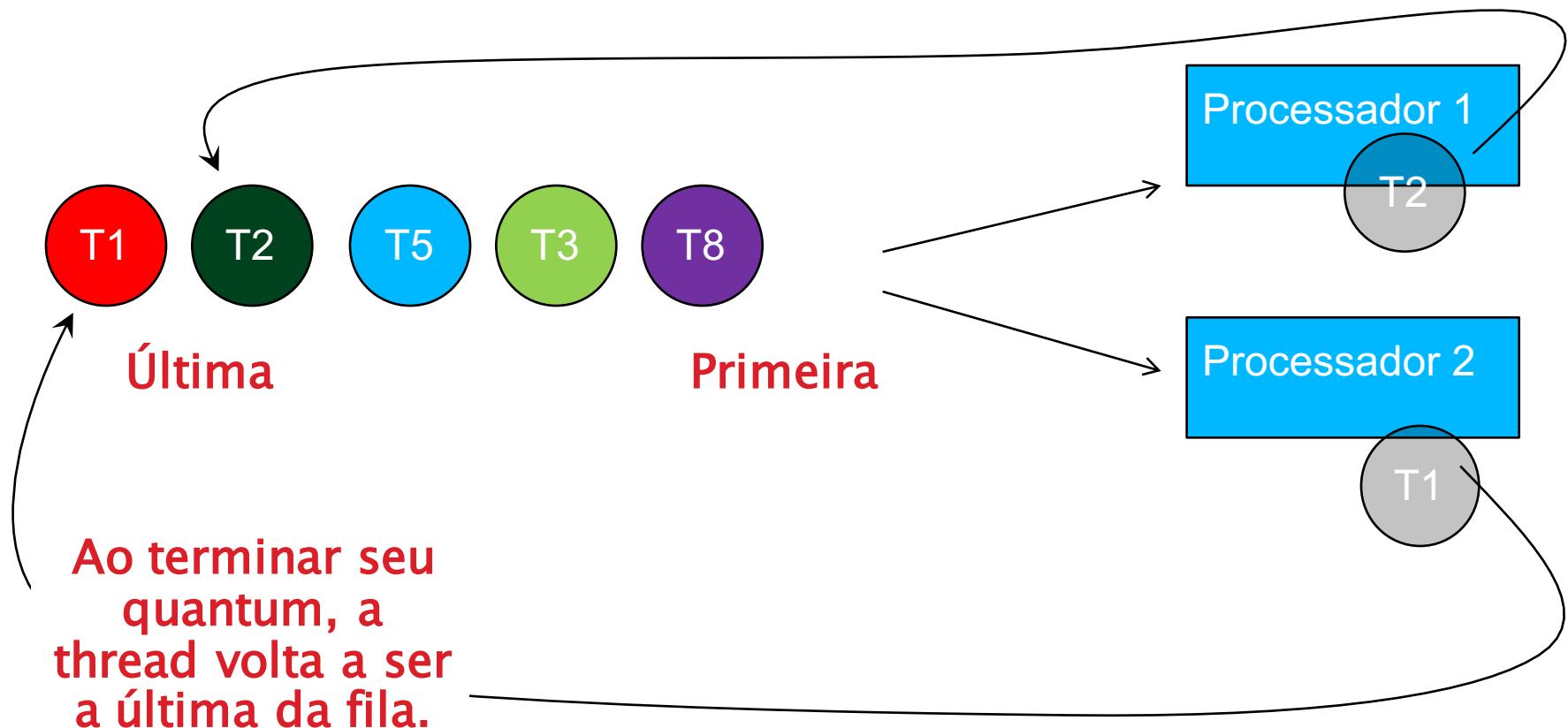


Processador 2



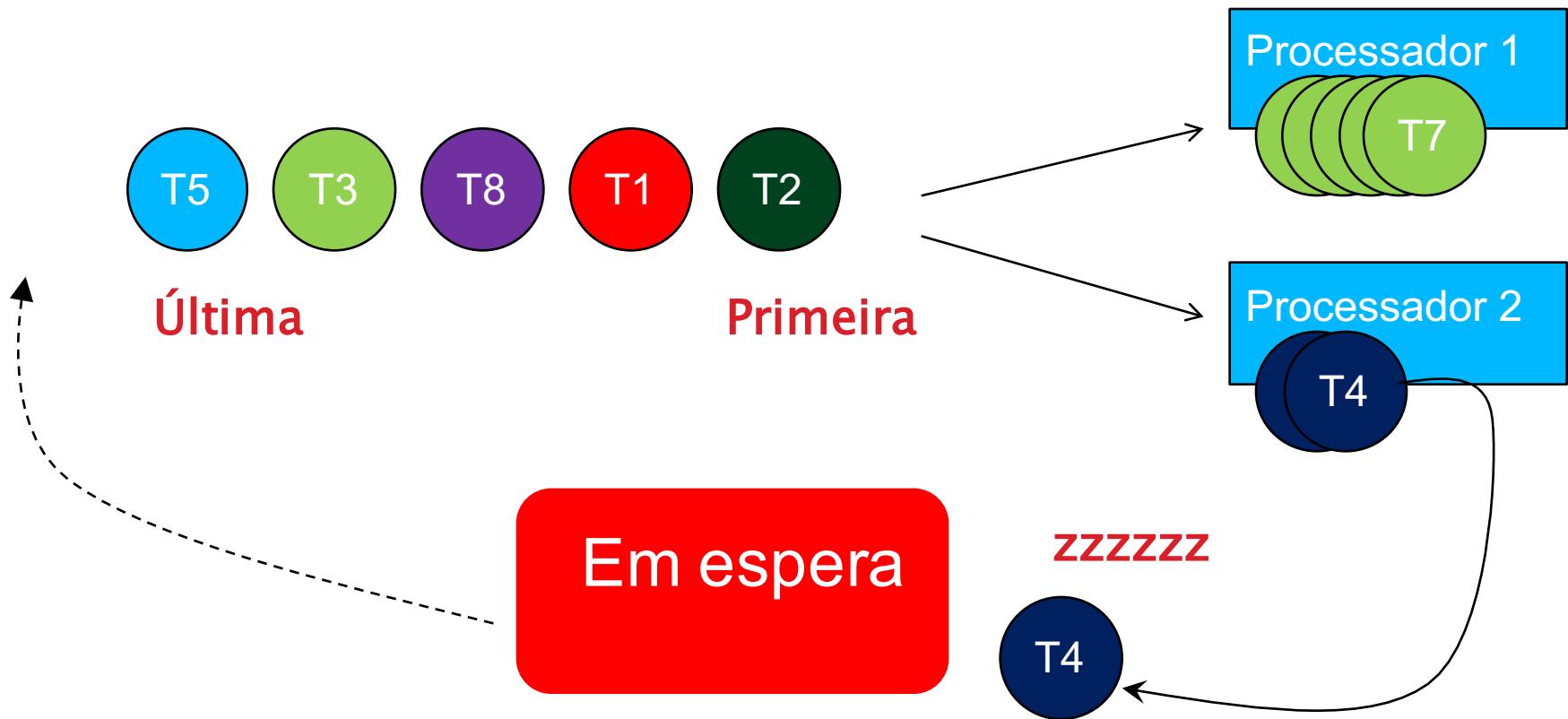
O processador executa a thread por uma pequena quantidade de tempo: quantum

# Estados de uma Thread



# Estados de uma Thread

- ▶ Ocasionalmente, ao estar sendo executada a Thread pode por algum motivo passar para o estado de espera, como acontece quando utilizamos o método Sleep:



# Sincronização de Threads

- ▶ O principal problema na programação multi-thread ocorre quando várias threads acessam um mesmo **objeto**.
- ▶ É possível gerar resultados inesperados quando duas threads tentam acessar propriedades do mesmo objeto, sendo que pelo menos uma das thread altera o valor de uma propriedade.

# Sincronização de Threads

- ▶ Exemplo:

```
public class Conta {  
  
    double saldo;  
  
    public void sacar(double quant) {  
        if (saldo > quant)  
            saldo -= quant;  
    }  
}
```

O método sacar foi implementado para não permitir que o saldo da conta fique negativo!!

# Sincronização de Threads

- ▶ Vamos supor a seguinte ocasião
    - O saldo atual da conta é 1000
- 
1. Uma thread T1 começa a executar o método sacar : **sacar(500)**
  2. T1 verifica que é possível realizar o saque ( $500 < 1000$ ) mas antes de realizar o saque seu tempo de execução termina, e ela volta para o final da fila
  3. Logo em seguida, outra thread T2 também começa a executar o método sacar : **sacar(700)**
  4. T2 verifica que é possível realizar o saque ( $700 < 1000$ ) e realiza o saque
  5. T1, que já tinha verificado ser possível sacar, volta a executar e realiza também o saque. O saldo da conta passa a ser negativo, contrariando o que havia sido previsto no método sacar.

# Sincronização de Threads

- ▶ Os trechos de código onde podem ocorrer este tipo de situação são chamados de **blocos críticos**, ou **operações críticas**.
- ▶ Para que estes problemas não ocorram, estes blocos de código precisam ser **blindados** por meio da palavra reservada **synchronized**.

# Sincronização de Threads

- ▶ Exemplo:

```
public class Conta {  
  
    double saldo;  
  
    public void sacar(double quant) {  
        synchronized(this) {  
            if (saldo > quant)  
                saldo -= quant;  
        }  
    }  
}
```

# Sincronização de Threads

- ▶ Um bloco de código marcado como synchronized pode ser executado por apenas uma thread por vez.
- ▶ Em outras palavras, não é possível duas threads acessarem ao mesmo tempo o mesmo bloco synchronized.
- ▶ Se uma thread T2 tenta entrar em um bloco synchronized que já está sendo executada pela thread T1, T2 entrará em estado de espera, e só sairá deste estado quando T1 sair do bloco synchronized.

# Sincronização de Threads

## ► Vamos novamente supor a ocasião de saque

1. Uma thread T1 começa a executar o método sacar : sacar(500)
2. T1 verifica que é possível realizar o saque ( $500 < 1000$ ) mas antes de realizar o saque seu tempo de execução termina, e ela volta para o final da fila
3. Logo em seguida, outra thread T2 **tenta começar** a executar o método sacar, mas como T1 já está executando o trecho sincronizado, T2 vai para o estado de espera
4. T1, que já tinha verificado ser possível sacar, volta a executar e realiza o saque. Em seguida, T2 sai do estado de espera.
5. T2 volta a executar o método sacar, mas verifica que não é mais possível.

Agora não tivemos problema

# Sincronização de Threads

- Também poderíamos colocar um método inteiro como synchronized. Ex:

```
public class Conta {  
  
    double saldo;  
  
    public synchronized void sacar(double quant) {  
  
        if (saldo > quant)  
            saldo -= quant;  
  
    }  
}
```

# Sincronização de Threads

- ▶ Importante:
  - Devemos marcar com synchronized somente os blocos críticos. Se marcarmos um bloco não crítico como synchronized, deixaremos nosso código menos eficiente porque impediremos que duas threads executem ao mesmo tempo, em processadores diferentes.

**Modelo  
produtor/  
consumidor**

# Modelo produtor/consumidor

- ▶ O modelo produtor/consumidor consiste no relacionamento entre duas partes de um mesmo programa
  - uma parte que **gera dados e os armazena em um container** (esta é a parte **produtora**)
  - E uma parte que **utiliza estes mesmos dados** em algum propósito, buscando-os no container . (esta é a parte **consumidora**)

# Modelo produtor/consumidor

- ▶ Existem inúmeros aplicativos que possuem esse tipo de relacionamento. Exs:
  - Um gerenciador de impressão
    - Vários arquivos são enviados para uma fila de impressão que fica em um servidor de impressão. Assim, o servidor vai despachando (consumindo) estes arquivos para a impressora até que todos sejam impressos
  - Gravação de CD , DVD
  - Reprodução de Vídeos puxados pela internet
  - Eventos gerados em um sistema de janelas

# Modelo produtor/consumidor

- ▶ Esse tipo de programa grava e lêem informações em uma área de memória comum, que denominamos *buffer*
- ▶ Tipicamente, o buffer é uma **fila** que possui uma capacidade de armazenamento limitada
- ▶ Os consumidores e os produtores são linhas de execução independentes: Threads

# Modelo produtor/consumidor

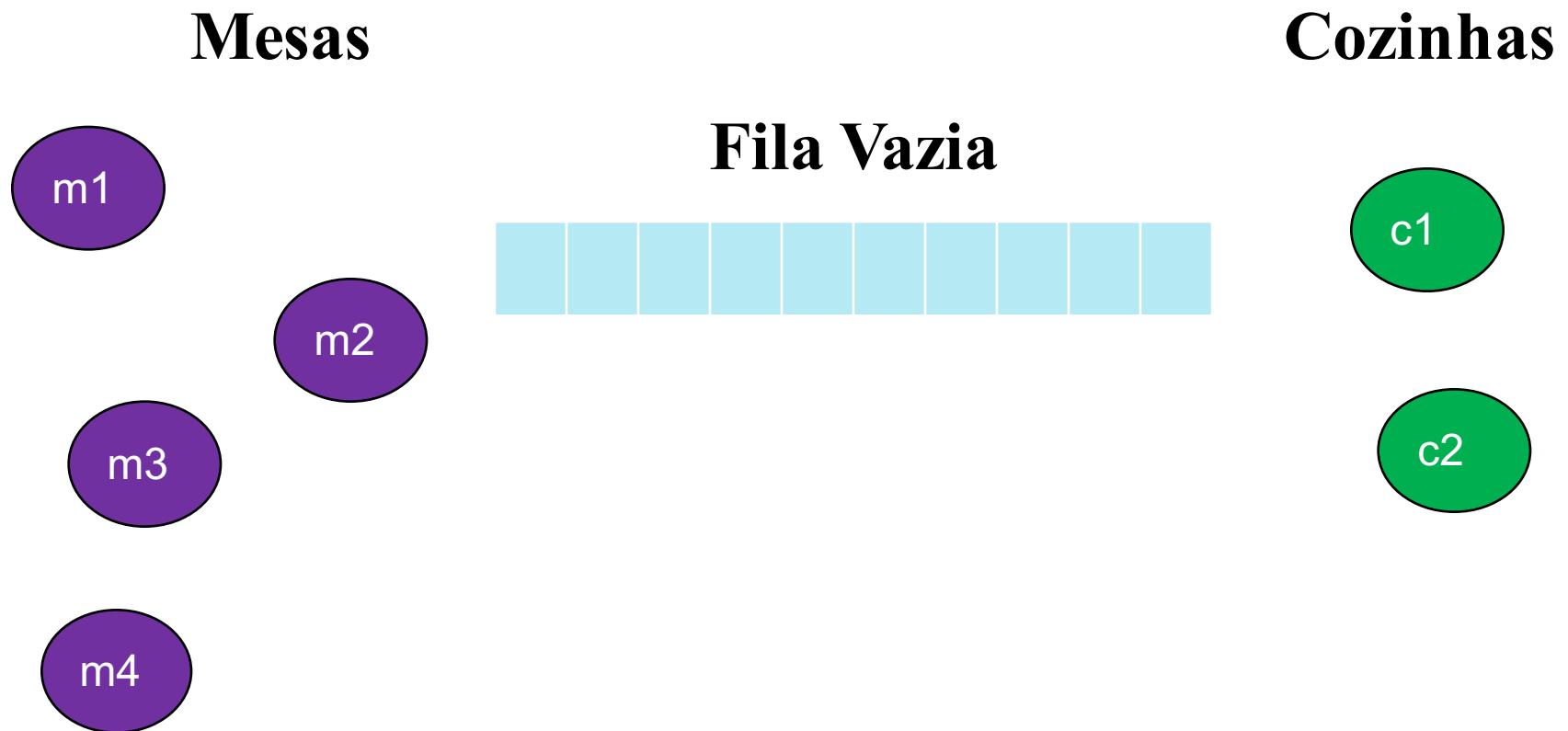
- ▶ Existem alguns comandos importantes, pois
  - Uma thread consumidora pode tentar consumir dados de um buffer vazio
  - Uma thread produtora pode tentar inserir dados em um buffer cheio

# Modelo produtor/consumidor

- ▶ Exemplo: Simulador de pedidos em um restaurante
  - Objetos **Mesa** serão “**Produtores**” de Pedidos
  - Objetos **Cozinha** será “**Consumidores**” de Pedidos

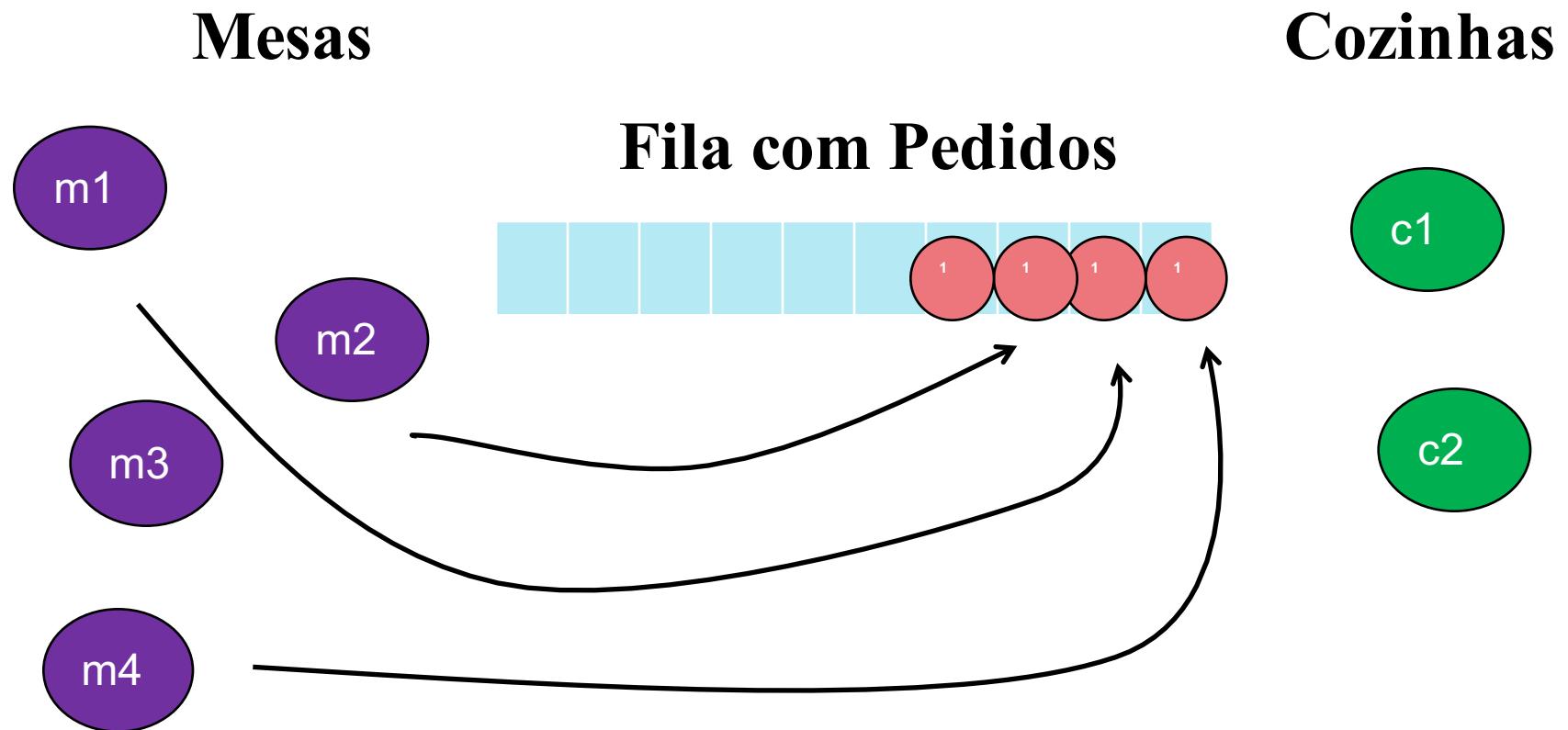
# Modelo produtor/consumidor

- Exemplo: Simulador de pedidos em um restaurante



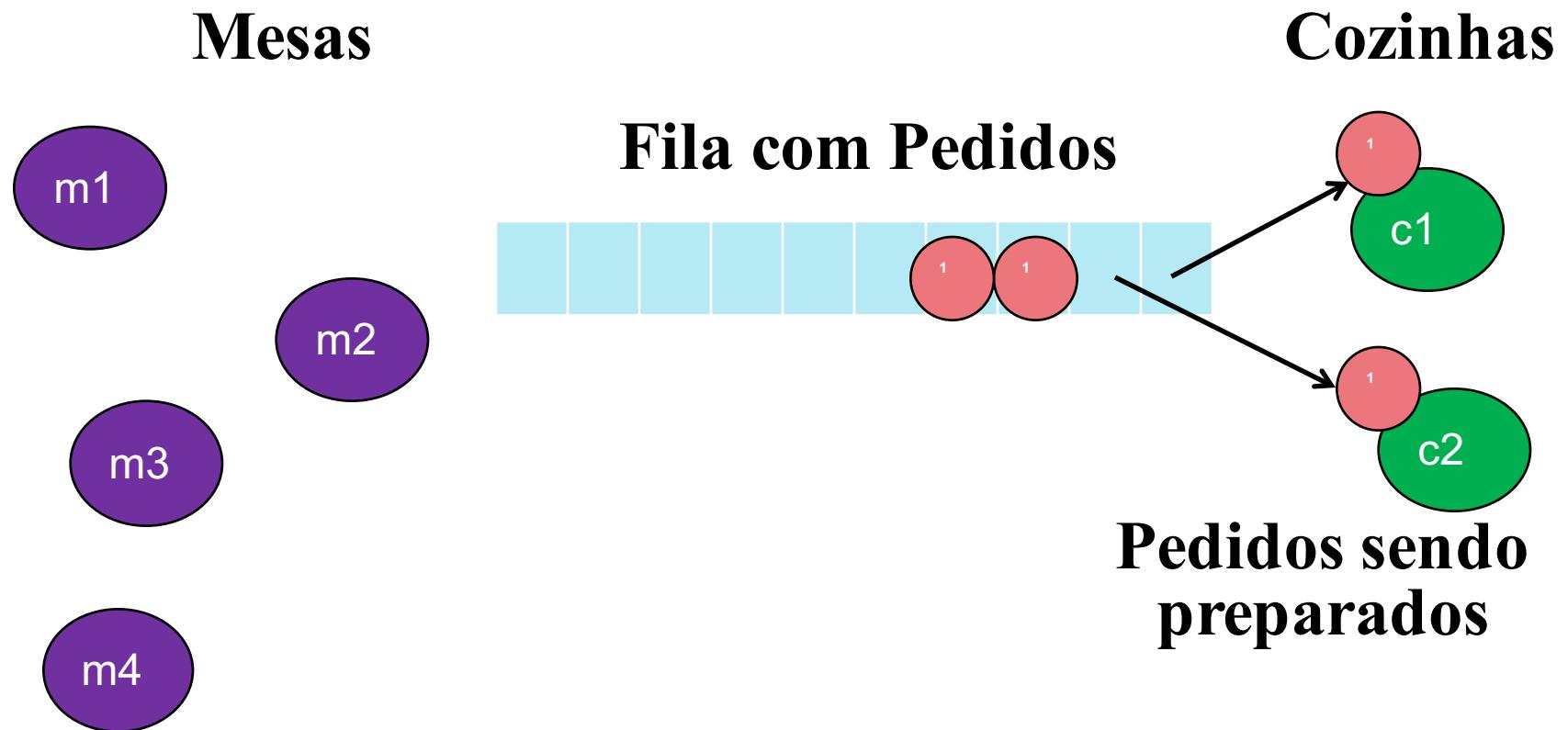
# Modelo produtor/consumidor

- Exemplo: Simulador de pedidos em um restaurante



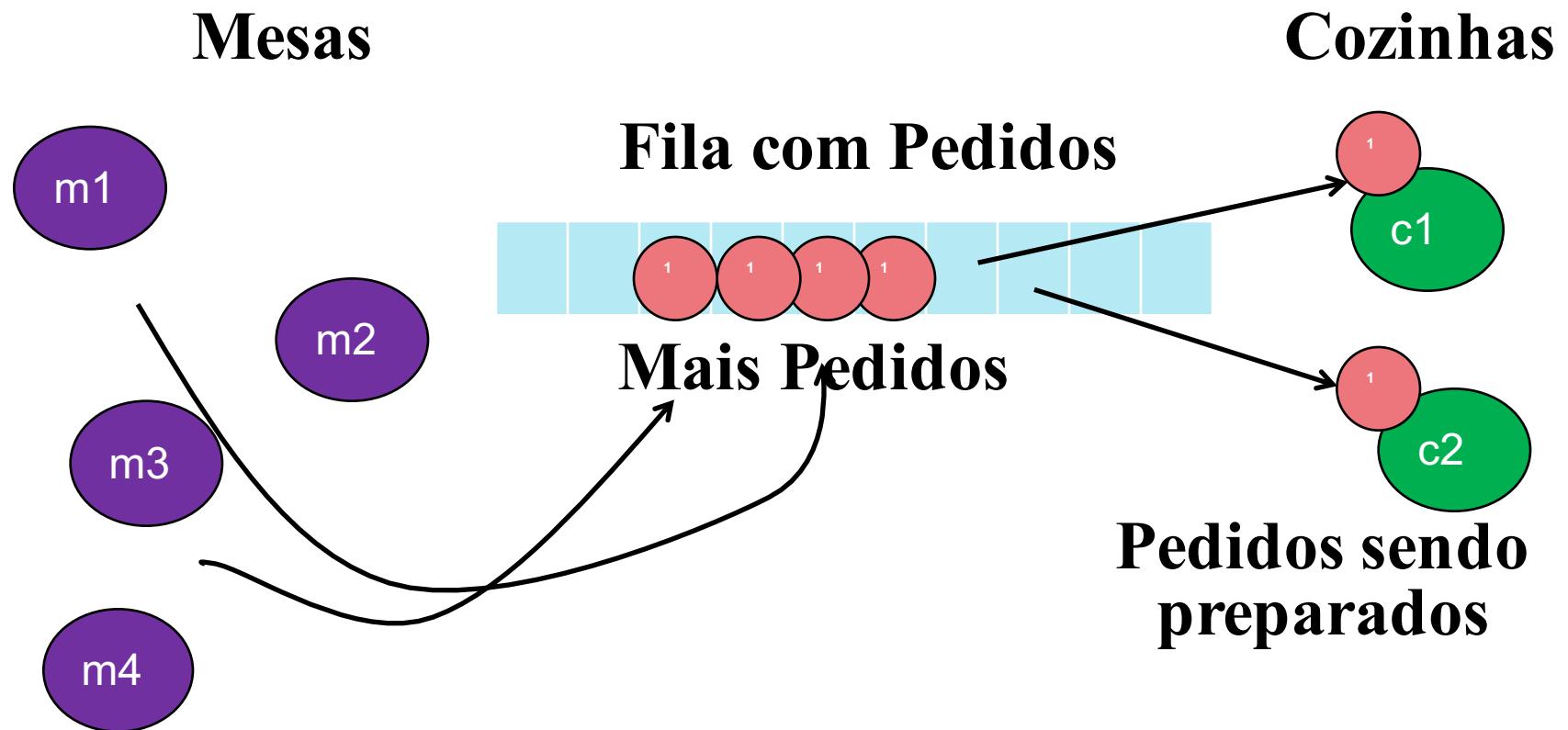
# Modelo produtor/consumidor

- Exemplo: Simulador de pedidos em um restaurante



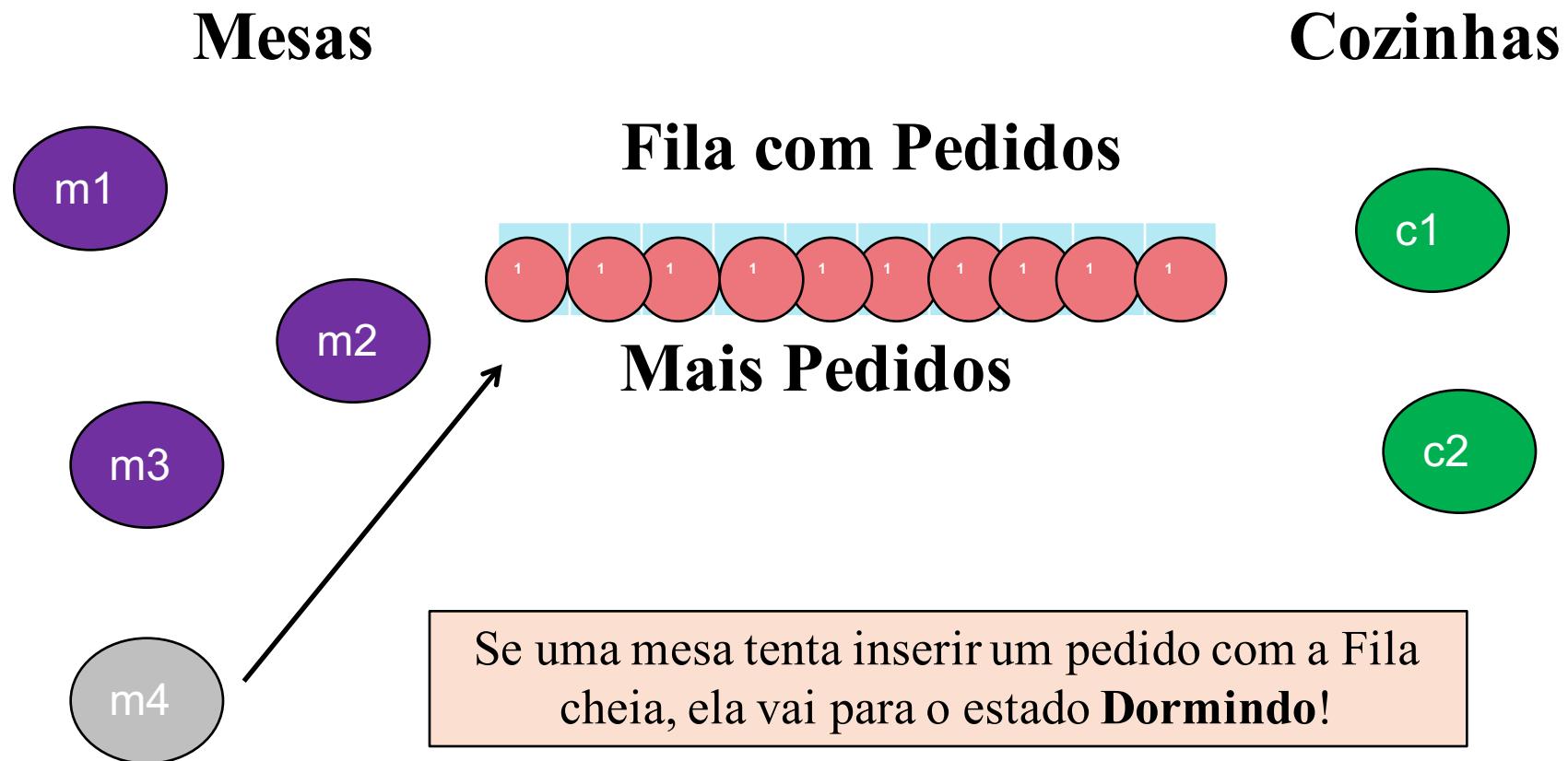
# Modelo produtor/consumidor

- Exemplo: Simulador de pedidos em um restaurante



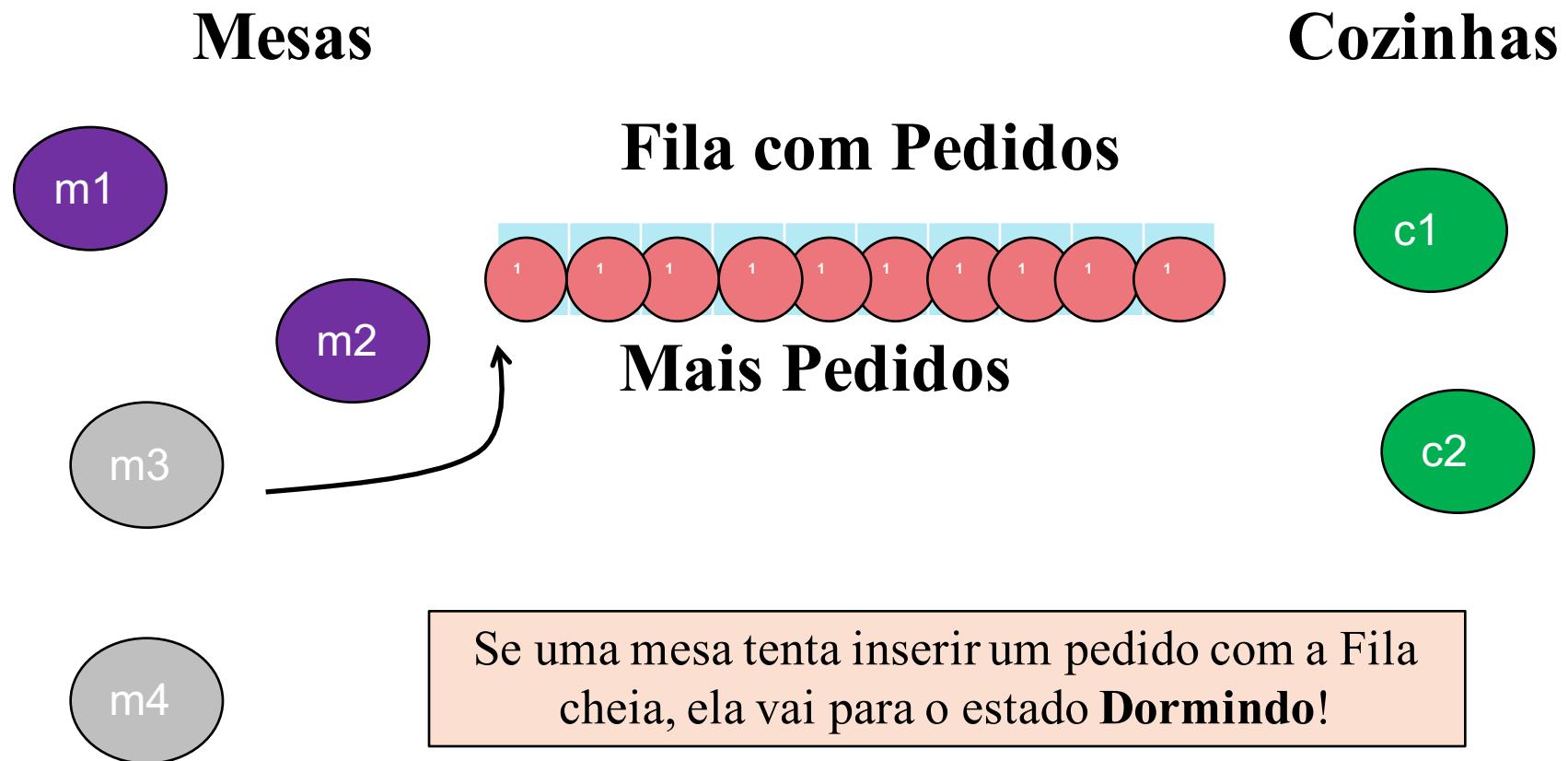
# Modelo produtor/consumidor

- Exemplo: Simulador de pedidos em um restaurante



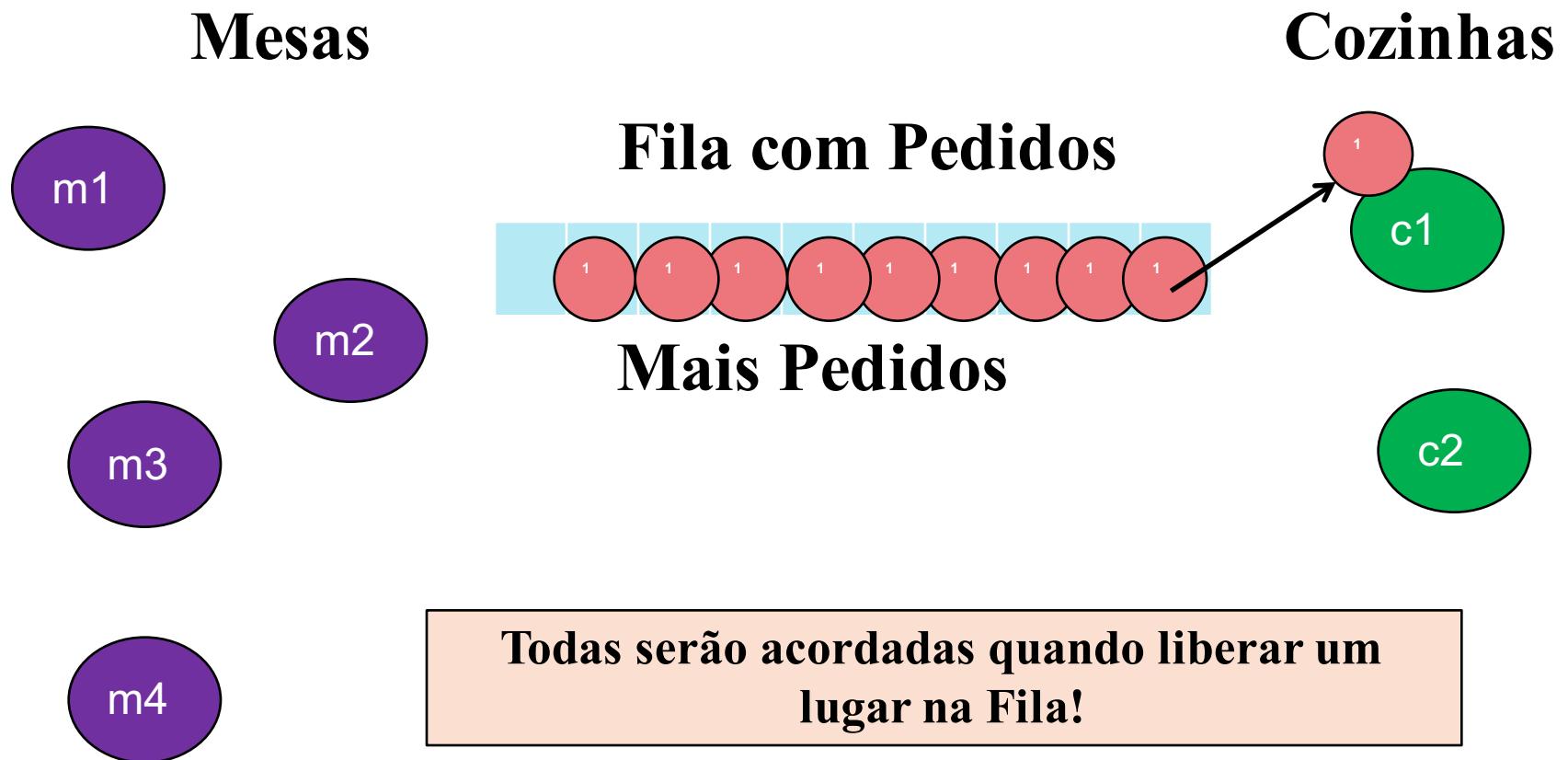
# Modelo produtor/consumidor

- Exemplo: Simulador de pedidos em um restaurante



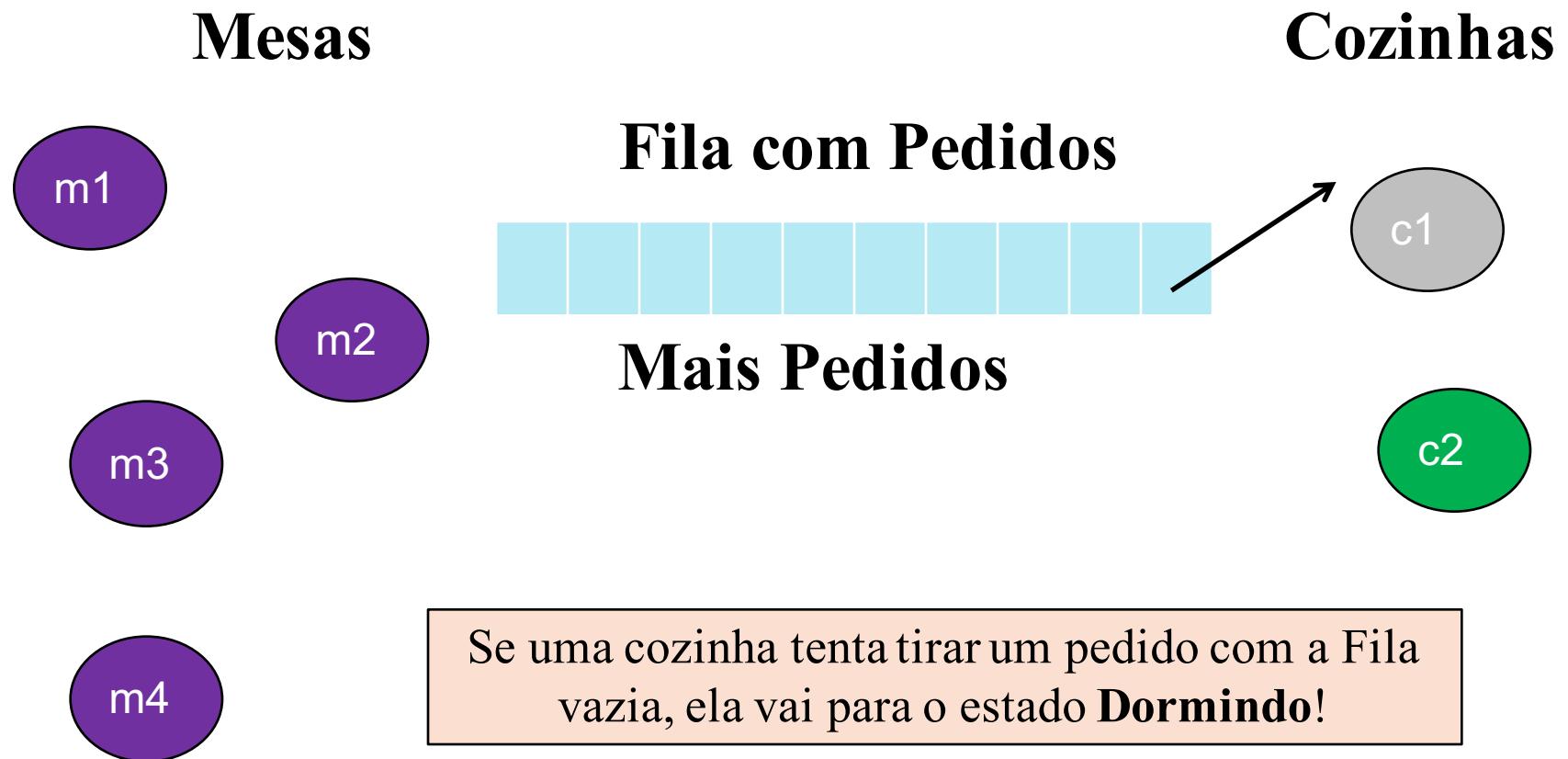
# Modelo produtor/consumidor

- Exemplo: Simulador de pedidos em um restaurante



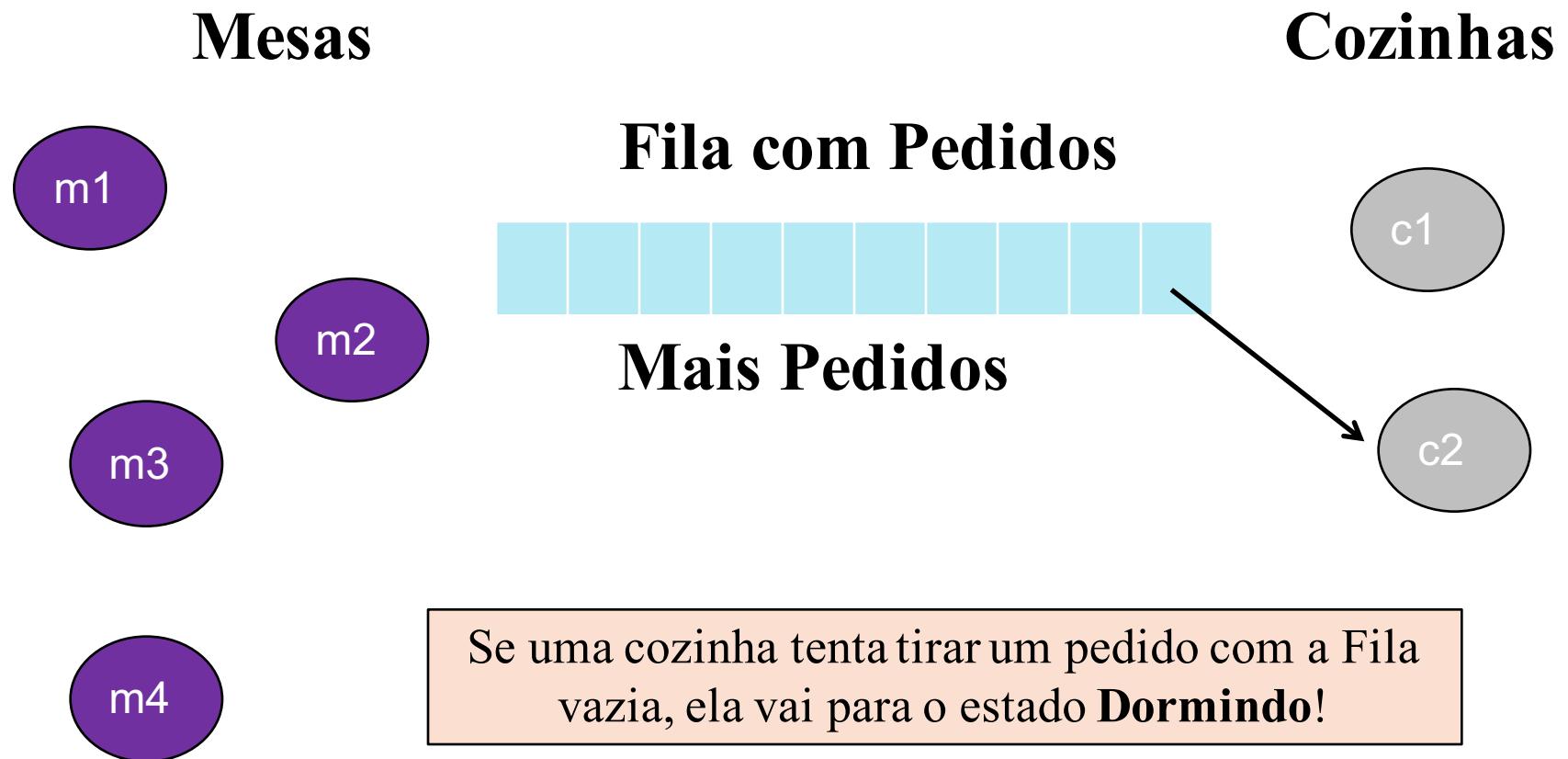
# Modelo produtor/consumidor

- Exemplo: Simulador de pedidos em um restaurante



# Modelo produtor/consumidor

- Exemplo: Simulador de pedidos em um restaurante



# Modelo produtor/consumidor

- Exemplo: Simulador de pedidos em um restaurante



# Classe Pedido

```
public class Pedido {  
  
    int numero;  
    String descrição;  
  
    static int count = 0;  
    static Random ran = new Random();  
  
    public Pedido() {  
  
        String opções [] = {"arroz", "feijão", "bife", "lanche"};  
        numero = ++count;  
        descrição = opções[ran.nextInt(opções.length)];  
    }  
}
```

# Programa Principal

```
public static void main(String[] args) {  
  
    BlockingQueue<Pedido> pedidos = new ArrayBlockingQueue<Pedido>(100);  
  
    Mesa m1 = new Mesa(1, pedidos);  
    Mesa m2 = new Mesa(2, pedidos);  
    Mesa m3 = new Mesa(3, pedidos);  
    Mesa m4 = new Mesa(4, pedidos);  
    Mesa m5 = new Mesa(5, pedidos);  
  
    Cozinha c1 = new Cozinha(pedidos);  
    Cozinha c2 = new Cozinha(pedidos);  
  
    m1.start();  
    m2.start();  
    m3.start();  
    m4.start();  
    m5.start();  
  
    c1.start();  
    c2.start();  
}
```

# Produtor

```
public class Mesa extends Thread{
    BlockingQueue<Pedido> pilha;
    int numero;

    public Mesa(int numero, BlockingQueue<Pedido> pilha) {
        this.numero = numero;           this.pilha = pilha;
    }

    @Override
    public void run() {
        Random ran = new Random();
        for (int i = 0; i < 40; i++) {
            Pedido p = new Pedido();
            try {
                pilha.put(p);
                System.out.println("Novo pedido nº " + p.numero + " da mesa" + numero);
                Thread.sleep(ran.nextInt(1000));
            } catch (InterruptedException ex) {
                break;
            }
        }
    }
}
```

# Consumidor

```
public class Cozinha extends Thread {  
    BlockingQueue<Pedido> pilha;  
  
    public Cozinha(BlockingQueue<Pedido> pilha) {  
        this.pilha = pilha;  
    }  
  
    @Override  
    public void run() {  
        Random ran = new Random();  
        for (int i = 0; i < 5000; i++) {  
            try {  
                Pedido p = pilha.take();  
                Thread.sleep(ran.nextInt(3000));  
                System.out.println("Pedido nº " + p.numero + "(" + p.descrição+ ") pronto");  
            } catch (InterruptedException ex) {  
                break;  
            }  
        }  
    }  
}
```

# Exercício para casa

- ▶ Programar o simulador de pedidos de restaurante e experimentar os parâmetros de tempo e os números de objetos produtores e consumidores