Threads: concorrência em Java

Prof. Tiago Massoni

Concorrência

- Em um sistema multi-tarefa, várias atividades ocorrem de forma concorrente
- Distinção
  - Multi-tarefa de processos
    - Windows
  - Multi-tarefa de linhas de execução (threads)
    - dentro de um processo
    - Navegador, programa de animação

Benefícios de threads

- Explorar processadores múltiplos
- Simplicidade (threads são objetos em Java)
- Tratamento de eventos assíncronos
- GUI mais rápida

Produtor-consumidor

```
public void inserir(Object item) {
     while (count == BUFFER_SIZE);
     count++;
     buffer[in] = item;
     in = (in + 1) % BUFFER SIZE;
}
```

```
public Object remove() {
     Object item;
     while (count == 0);
     count--;
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
     return item;
}
```

Acesso perigoso

Na execução real...

```
count++;
```

```
count--;
```

Código Java


```
reg1=count;
reg1=reg1+1;
count=reg1;
```

```
reg2=count;
reg2=reg2-1;
count=reg2;
```

Linguagem de máquina

Interleaving na CPU

```
reg1=count;
reg1=reg1+1;
reg2=count;
reg2=reg2-1;
count=reg1;
count=reg2;
```

- Exemplo
  - Buffer com no máximo 5 objetos (size=4)
  - No final, um dos valores vai se sobrepor
  - Condições de corrida
    - Resultado depende da ordem de execução

Processos e threads

- Threads
  - Usam mesmo espaço de endereçamento
  - Comunicação entre elas é simples
- Em sistemas com apenas uma CPU
  - Contexto é chaveado entre as threads
  - Concorrência não é real (simulação)
  - Ordem de execução não pode ser prevista
- Algumas linguagens possuem modelo de concorrência próprio (ex: ADA e Java)
  - Outras linguagens utilizam extensões para lidar com threads (ex: C tipicamente usa Posix)

Threads em Java

- Diferentes linhas de execução no mesmo programa
- Podem compartilhar dados e código
- main(): thread principal
  - A partir desta as outras threads serão iniciadas

Declaração de threads

- Duas formas possíveis de se declarar threads
  - Herdando da classe java.lang.Thread
  - Implementando a interface java.lang.Runnable
- Antes de escolher, prestar atenção em:
  - Herança sem necessidade
  - Não tem herança múltipla em Java

Mais usado: interface Runnable

```
public class TesteThread implements Runnable {

  private int contador = 0;

  public void run() {
    try{
       while (contador <= 100){
         System.out.println(contador++);
         Thread.sleep(200);
        }
    }catch(InterruptedException ie){
         ...
    }
  }
}
```

Executor de threads (anterior a Java5)

```
public class Tester {
  public static void main(String [] args){
    TesteThread tt1 = new TesteThread();
    TesteThread tt2 = new TesteThread();
    TesteThread tt3 = new TesteThread();

    Thread t1 = new Thread(tt1);
    Thread t2 = new Thread(tt2);
    Thread t3 = new Thread(tt3);

    t1.start();t2.start();t3.start();
  }
}
```

Executor de threads (pool)

```
Import java.util.concurrent.*;

public class Tester {

  public static void main(String [] args){
    TesteThread t1 = new TesteThread();
    TesteThread t2 = new TesteThread();
    TesteThread t3 = new TesteThread();

    ExecutorService tExecutor=
                  Executors.newFixedThreadPool(3);
    tExecutor.execute(t1);
    tExecutor.execute(t2);
    tExecutor.execute(t3);

  }
}
```

Interrompendo threads

- Thread termina quando o método run() é executado até o final
- Método de instância interrupt() faz pedido para uma thread ser interrompida.
  - O status da thread é modificado
  - O programador é responsável por terminar a thread sempre testando - isInterrupted()
  - Se a thread for interrompida quando bloqueada (dormindo), a exceção InterruptedException é lançada

Sincronização entre threads

- Threads podem acessar recursos compartilhados
  - Ex: vários débitos simultâneos na mesma conta
- Comunicação entre threads normalmente utiliza dados compartilhados
- Alguns recursos devem ter acesso restrito
  - Um thread por vez

Monitores: exemplo de método não-sincronizado

```
public class Conta {
...
  public void debitar (double valor)
              throws SaldoNegativoException {
   if (valor <= this.saldo)
     this.saldo -= valor;
   else
     throw new SaldoNegativoException("insuficiente");
  }...
}
```

Perigo: teste feito por duas threads!

Como sincronizar

- Threads ganham acesso a um recurso adquirindo seu monitor
  - Só com o monitor a thread pode acessar partes restritas do código
  - Cada objeto Java possui um monitor
- Alguns métodos em um objeto podem exigir um monitor para executar
  - Apenas uma thread pode executá-los por vez
  - Bloqueio de um objeto para outras threads
- Estes métodos são chamados sincronizados
  - Declarados com synchronized
  - Apenas uma thread pode executar um método sincronizado de um objeto por vez

Exemplo de método sincronizado

```
public class Conta {
...
  public synchronized void debitar (double valor)
              throws SaldoNegativoException {
   if (valor <= this.saldo)
     this.saldo -= valor;
   else
     throw new SaldoNegativoException("insuficiente");
  }...
}
```

Deadlock

```
public class Account {

    private float balance;

    public synchronized void deposit(float amount) {
        balance += amount;
    }

    public synchronized void withdraw(float amount) {
        balance -= amount;
    }

    public synchronized void transfer(float amount,Account
target) {
        withdraw(amount);
        target.deposit(amount);
    }
}
```

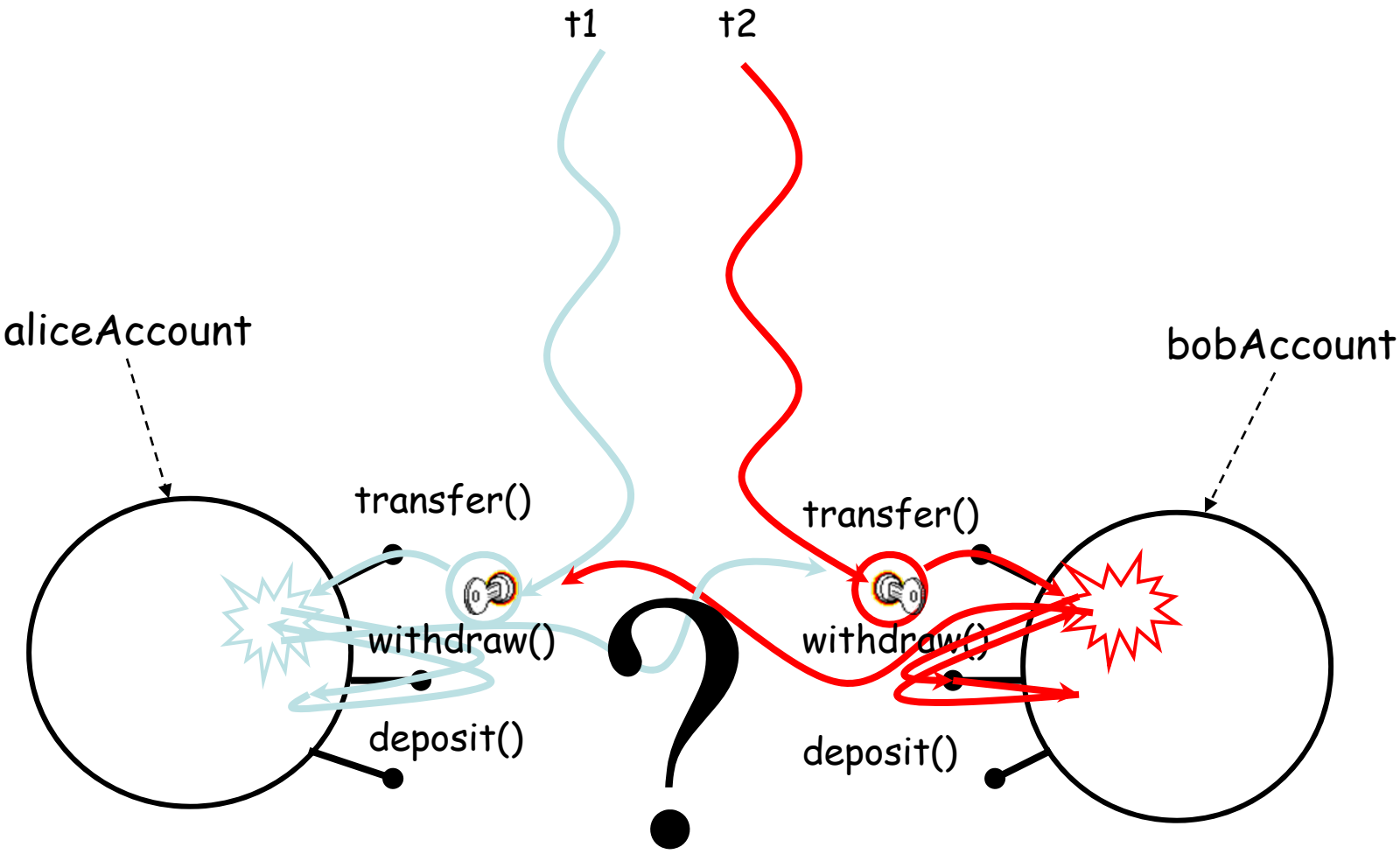
Deadlock

```
public class MoneyTransfer implements Runnable {

    private BankAccount from, to;
    private float amount;

    public MoneyTransfer(
        BankAccount from, BankAccount to, float amount)  {
        this.from = from;
        this.to = to;
        this.amount = amount;
    }

    public void run() {
        from.transfer(amount, to);
    }
}
```

Deadlock

```
BankAccount aliceAccount = new BankAccount();
BankAccount bobAccount = new BankAccount();
...

// At one place
Runnable transaction1 =
    new MoneyTransfer(aliceAccount, bobAccount, 1200);
//inicia transaction1

// At another place
Runnable transaction2 =
    new MoneyTransfer(bobAccount, aliceAccount, 700);
//inicia transaction2
```

Deadlock


t1
t2
aliceAccount
bobAccount
transfer()
transfer()
withdraw()
withdraw()
deposit()
deposit()
?

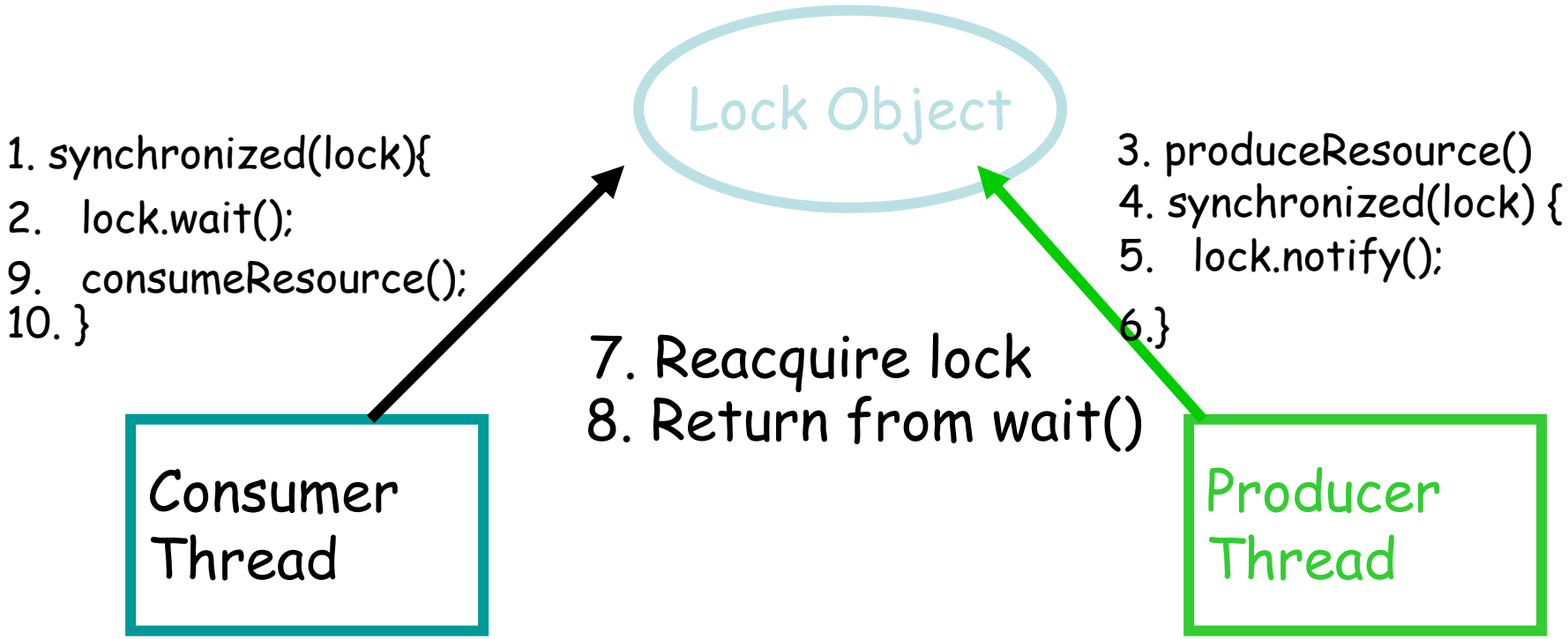
O método wait()

- **wait()** é parte de **java.lang.Object**
- Requer o lock do monitor do objeto para executar
- Melhor que seja executado de dentro de um método sincronizado. Por que?

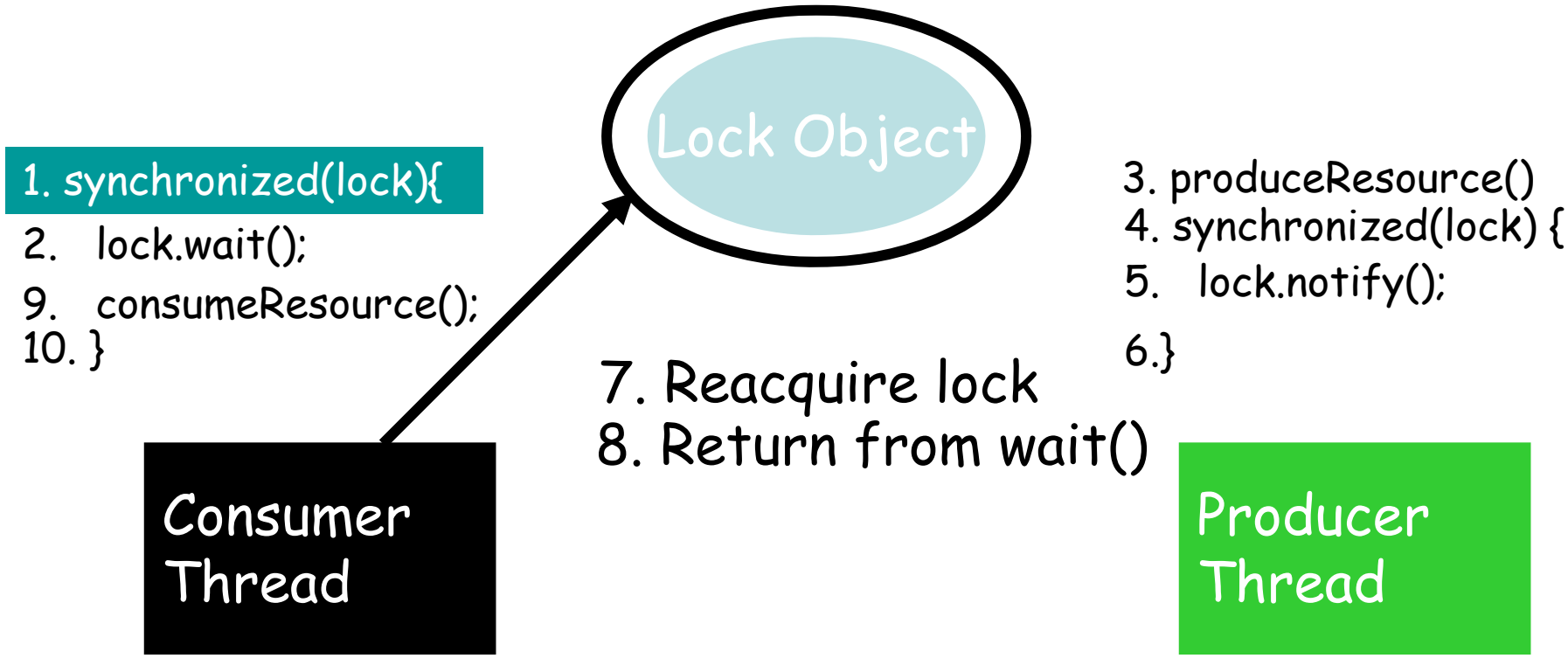
O método wait()

- wait() causa espera da thread atual até que outra thread invoque o método notify() ou notifyAll()
- Quando recebe wait(), a thread solta o lock do monitor

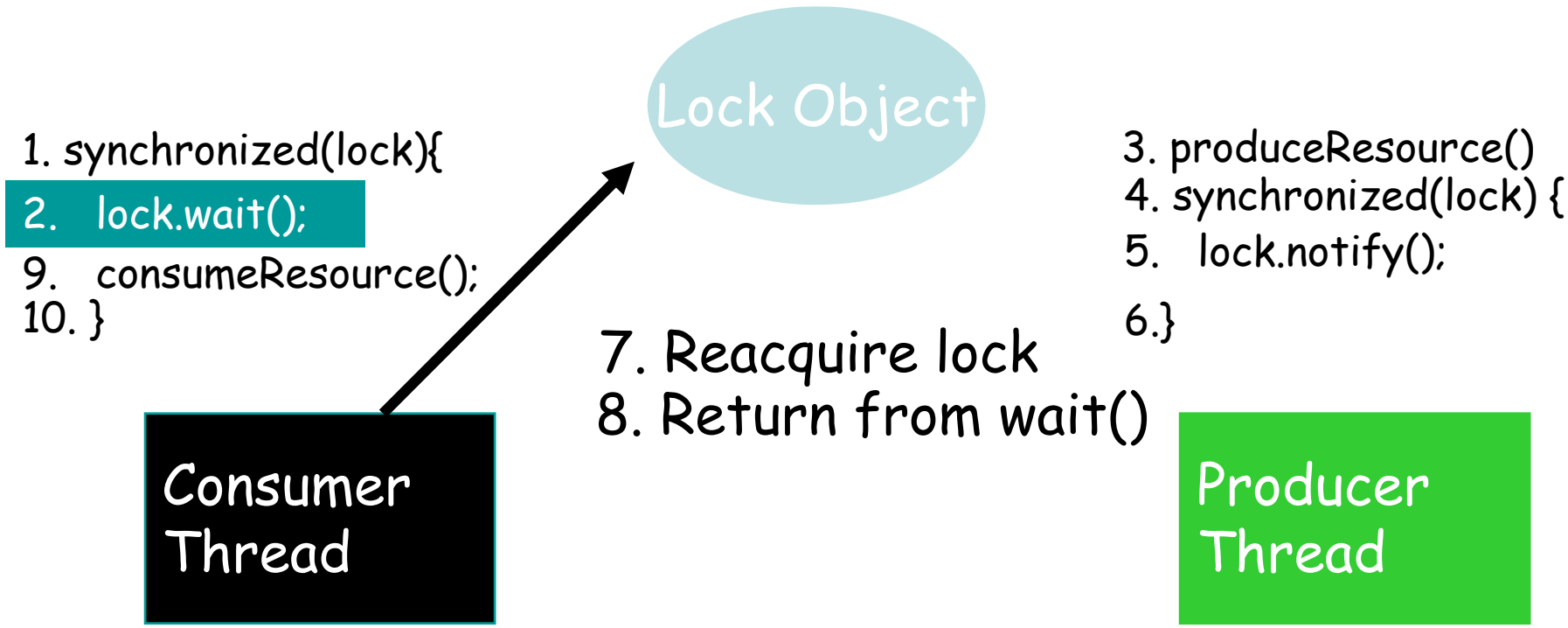
Sequência Wait/Notify



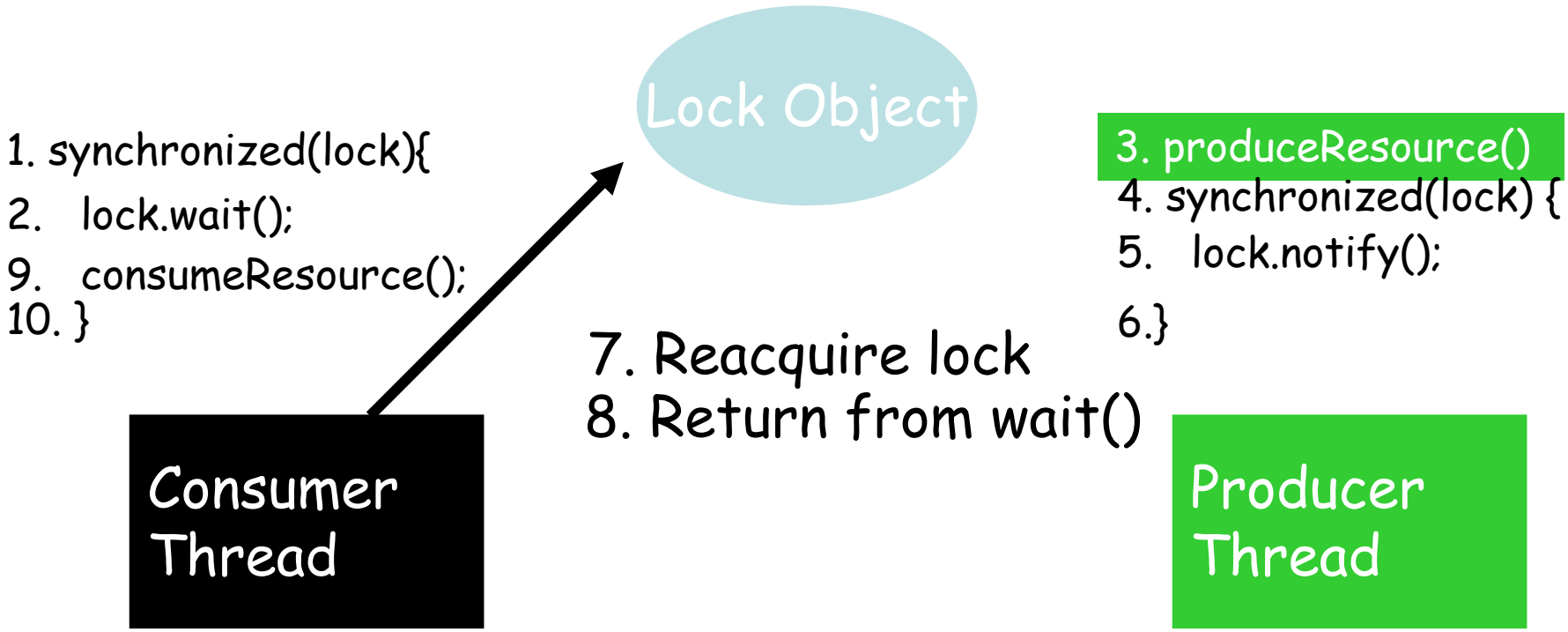
Sequência Wait/Notify



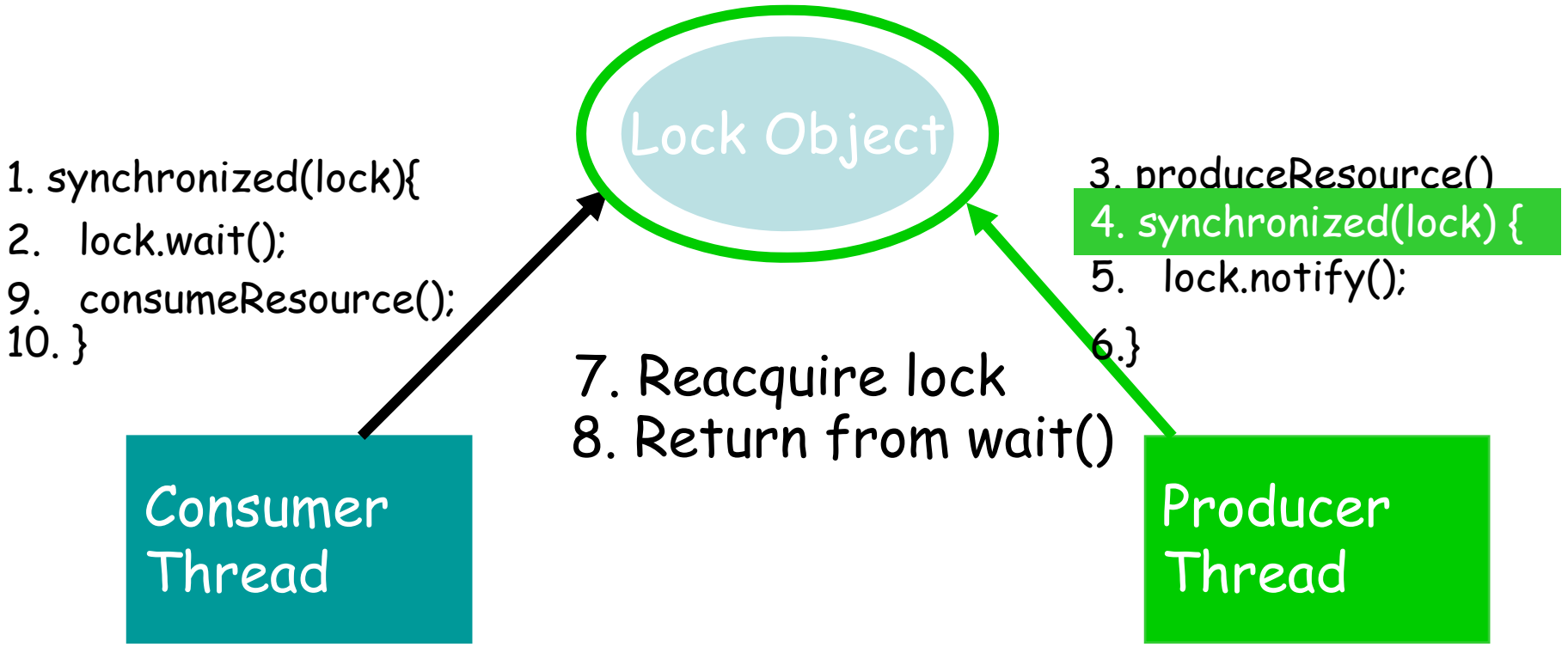
Sequência Wait/Notify



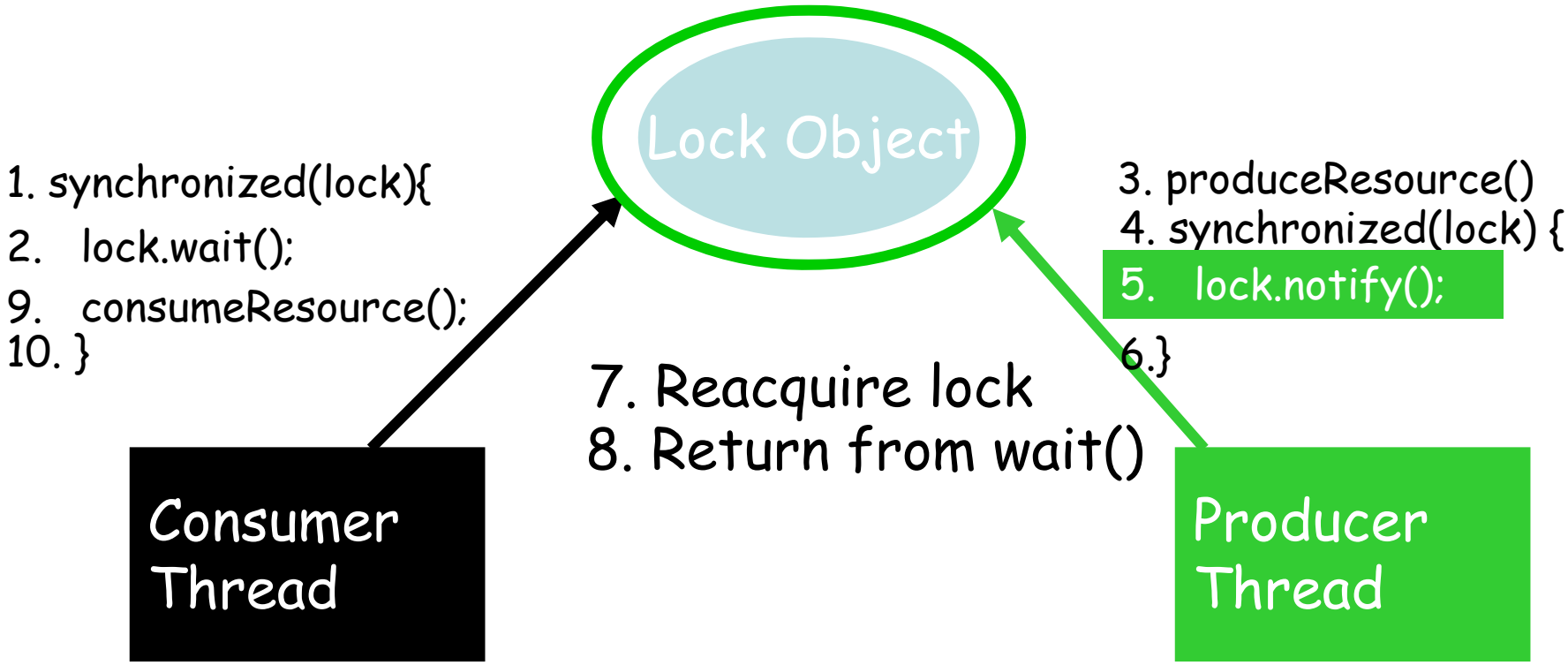
Sequência Wait/Notify



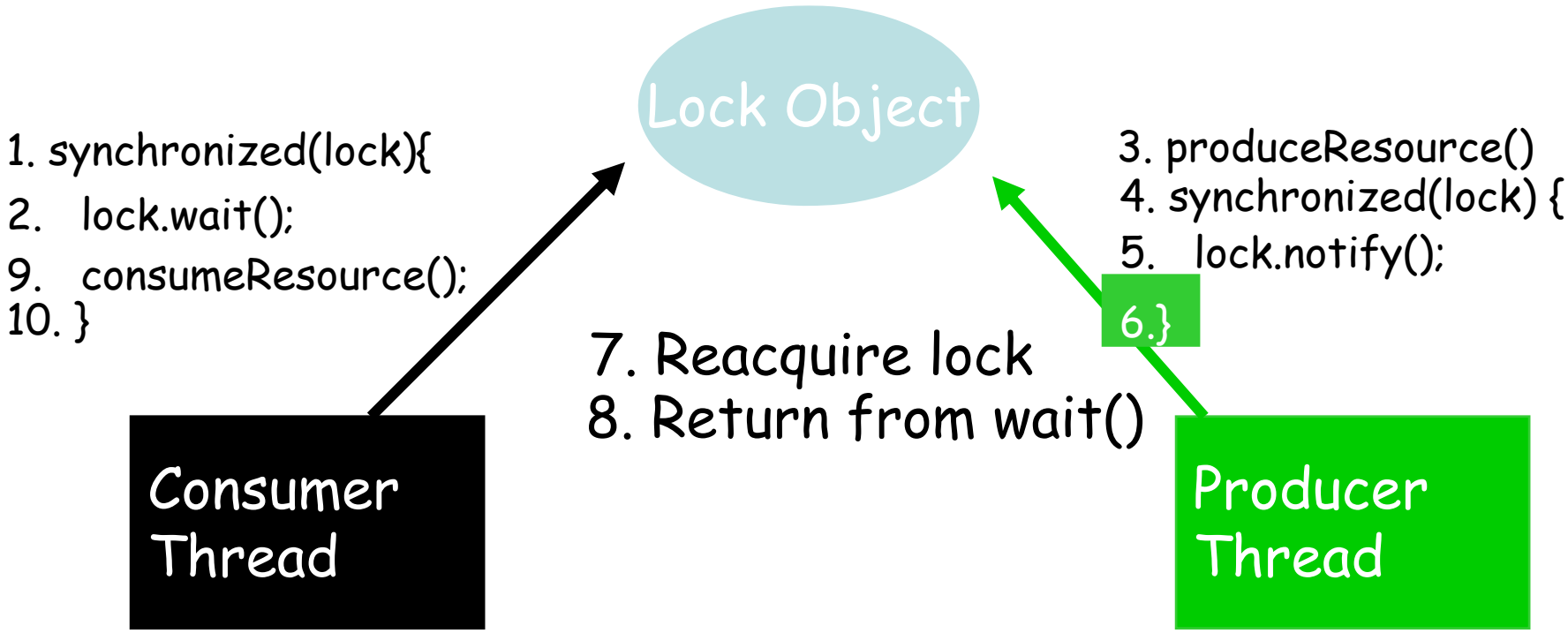
Sequência Wait/Notify



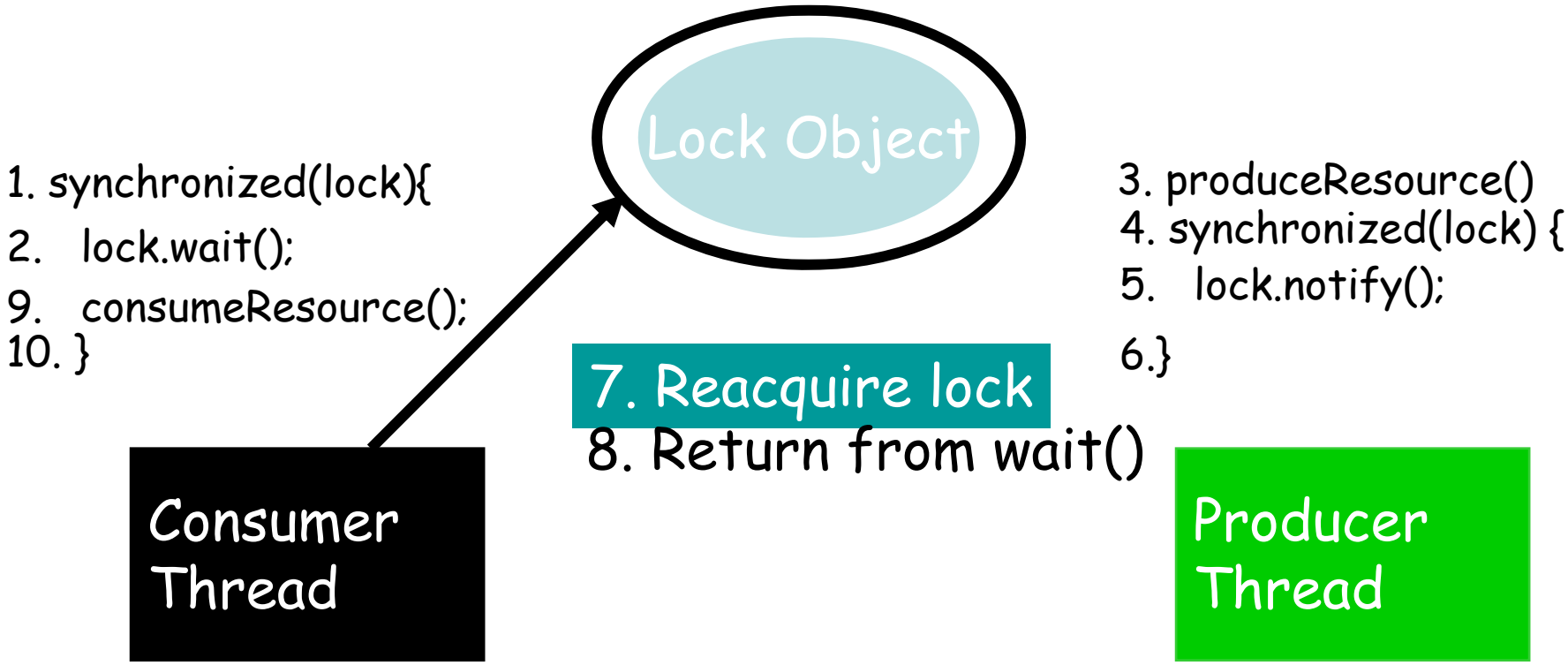
Sequência Wait/Notify



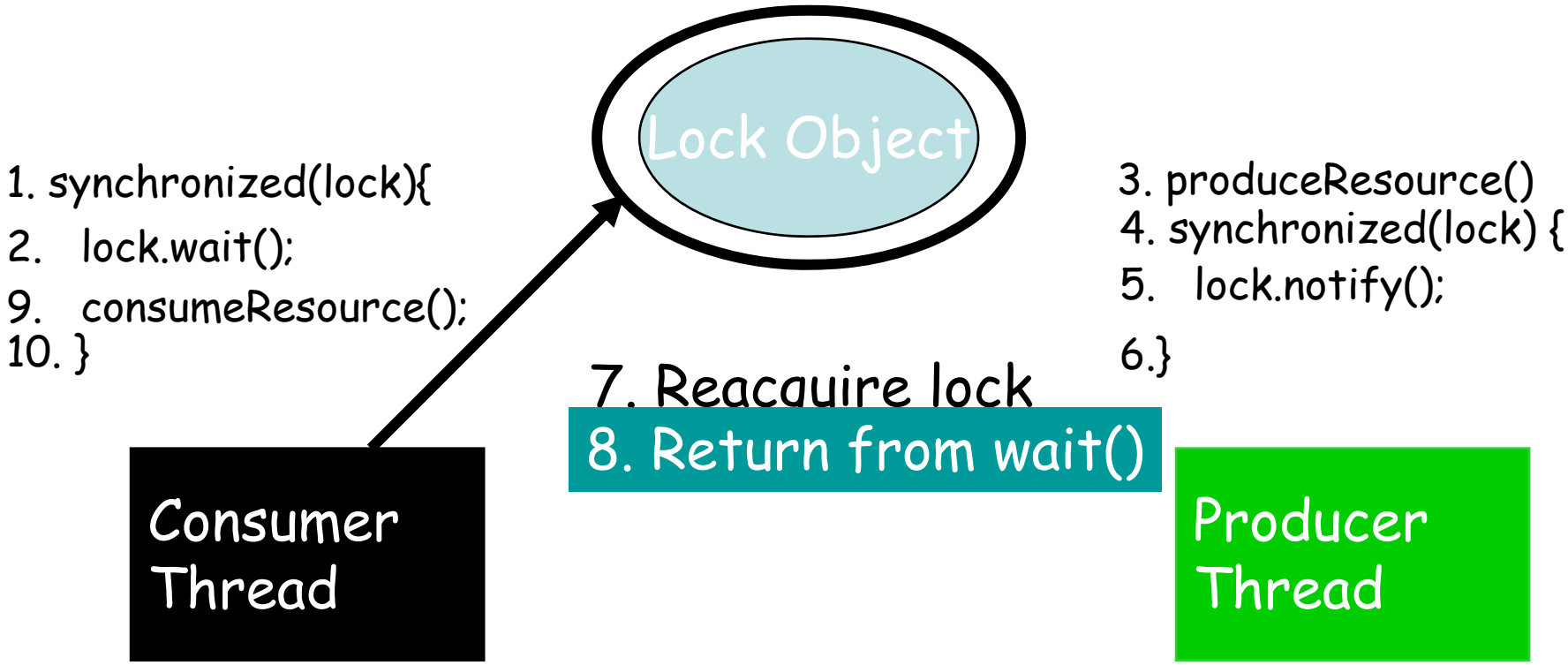
Sequência Wait/Notify



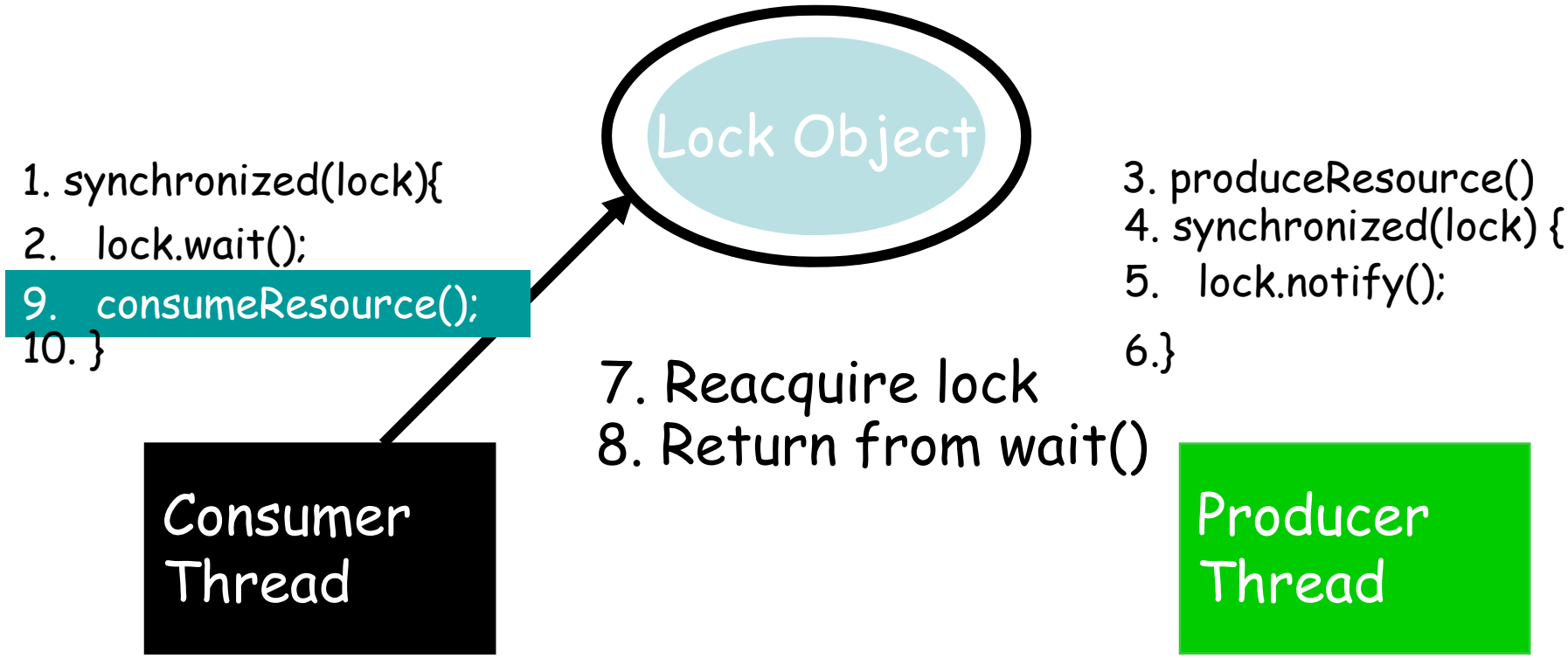
Sequência Wait/Notify



Sequência Wait/Notify



Sequência Wait/Notify



Sequência Wait/Notify


```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
10. }
```

Consumer Thread

7. Reacquire lock
8. Return from wait()

```
3. produceResource()
4. synchronized(lock) {
5.    lock.notify();
6.}
```

Producer Thread

Outro exemplo

```
public class Account {

    private float balance;

    public synchronized void deposit(float amount) {
        balance += amount;
    }

    public synchronized void withdraw(float amount) {
      if (balance < amount)
            balance -= amount;
    }

    …
}
```

Perigo: teste feito por duas threads!

Outro exemplo

```
public class Account {

    private float balance;

    public synchronized void deposit(float amount) {
        balance += amount;
    }

    public synchronized void withdraw(float amount) {
      while (balance < amount);
      balance -= amount;
    }

    …
}
```

Possível livelock

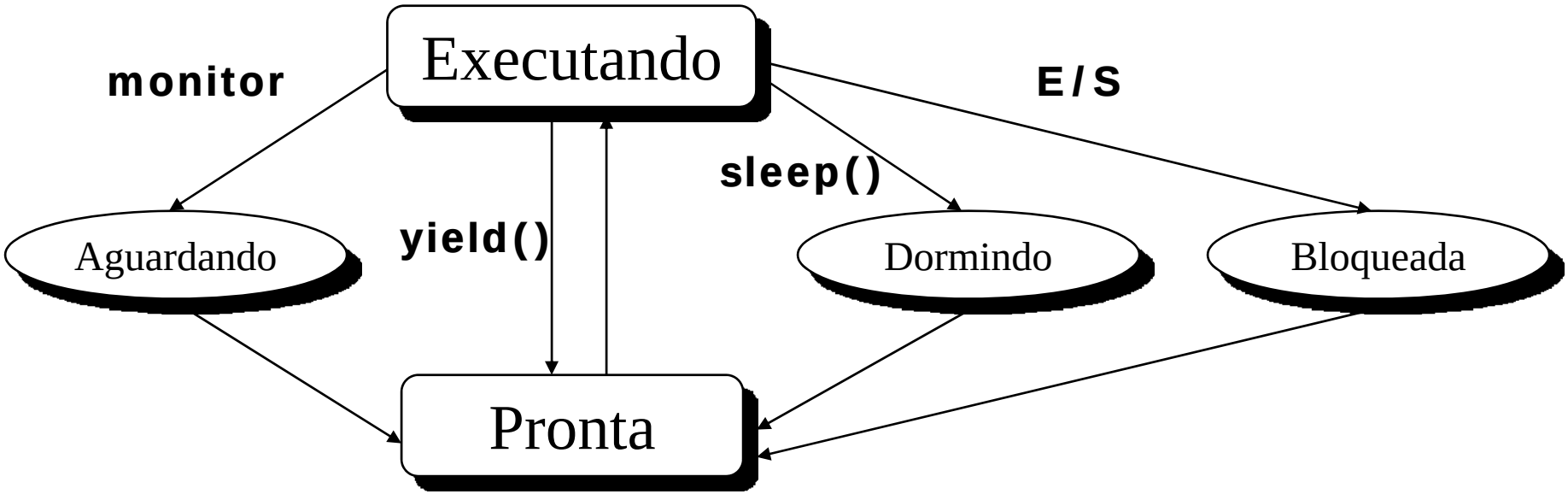
Uso de wait, notify

```
public class Account {

    private float balance;

    public synchronized void deposit(float amount) {
       balance += amount;
       notifyAll();
    }

    public synchronized void withdraw(float amount) {
       while (balance < amount)
              wait();
       balance -= amount;
    }

    …
}
```

Estados de uma Thread

- Executando: estado de execução
- Estados de espera: Aguardando, Dormindo, Bloqueada
- Pronta: Esperando CPU desocupar

Transições de Estado de uma Thread



Transições de Estado

- Método Thread.yield()
  - Thread executando atualmente dá lugar a outra thread
  - Volta ao estado de pronta
- Método Thread.sleep(tempo)
  - Thread atual vai para o estado Dormindo
  - Quando termina o tempo, vai para estado Pronta

Transições de Estado

- Thread bloqueada
  - Entrada e saída
  - Quando termina espera, thread fica pronta
- Thread aguardando (em espera)
  - Espera pelo monitor de um objeto
  - Acesso a código sincronizado
  - Chamada a wait()

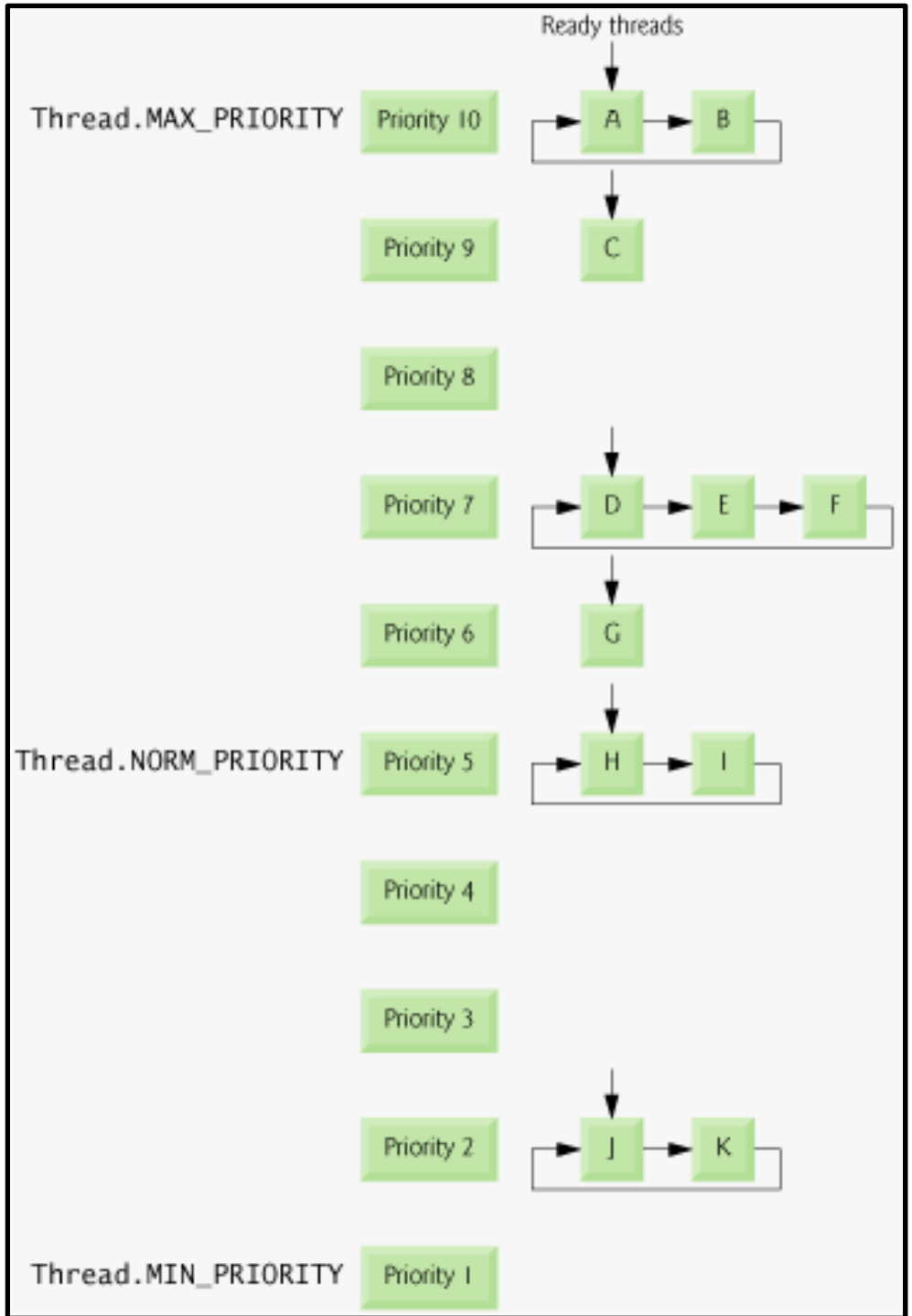
Prioridades de Threads

- Toda thread tem uma prioridade
  - Número de 1 a 10
- Quanto maior, maior prioridade
- Vale para a competição pela CPU que a JVM coordena
  - Threads no estado Pronta
  - Menor prioridade só é executada quando não existir prioridades maiores prontas
- Método
  - setPriority(int num)
  - getPriority()

Prioridades

- Constantes da classe Thread

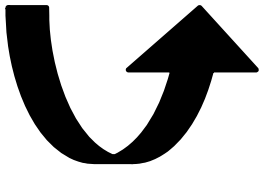
MIN_PRIORITY
NORM_PRIORITY
MAX_PRIORITY



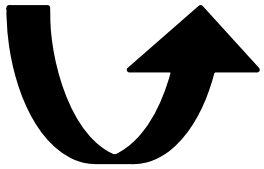
Exercícios de threads

Exercício 1

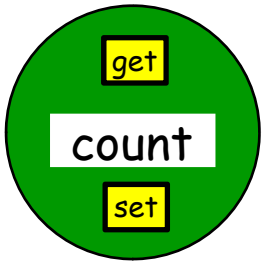
- Produtor e consumidor em um buffer compartilhado



Thread produtor



Thread consumidor



buffer

Especificação

- Interface Buffer (get e set com valor inteiro)
- Produtor:
  - Dorme tempo aleatório (entre 0 e 3s)
  - Seta número inteiro (1 a 10) no buffer
  - Soma os números (total tem que ser 55)
- Consumidor:
  - Dorme tempo aleatório (entre 0 e 3s)
  - Recupera número do buffer
  - Soma os números e imprime no final

Especificação

- Fazer dois buffers: um sem sincronização e outro com sincronização
  - Comparar os resultados
- Com sincronização
  - wait e notify
  - synchronized