

Programação Concorrente

[Magee 1999] Concurrency – State Models and Java Programs, *Jeff Magee, Jeff Kramer*, John Wiley 1999.

[Gosling] “The Java Language Specification” James Gosling, Bill Joy and Guy Steele, Addison-Wesley.
<http://java.sun.com/docs/books/jls/index.html>

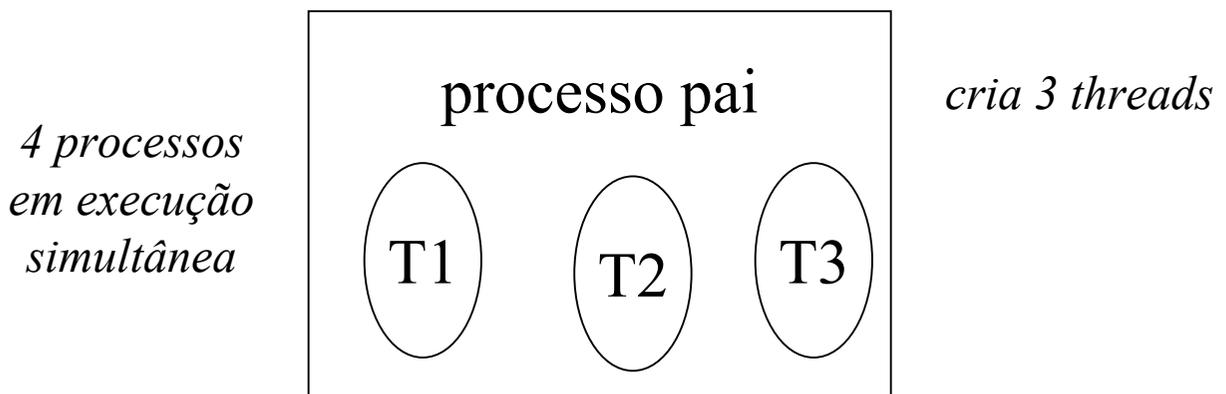
[Berg99] “Advanced Techniques for Java Developers” *Daniel J. Berg and J. Steven Fritzing*, John Wiley & Sons, 1999.

Programação Concorrente

Definição:

“Thread” – sequência de execução independente

As várias threads de um processo partilham o mesmo espaço de endereçamento que o processo (pai) que lhe deu origem.



Criação de Threads em Java

Hipótese 1:

```
public class MinhaThread_1 extends Thread {  
    public MinhaThread_1(){  
        Thread T = new Thread (this);  
        T.start();  
    }  
}
```

subclasse de Thread

objecto que contém o método run()

método definido na classe Thread, invoca o método run()

*≈ super();
start();*

Programação concorrente em Java

```
public void run() {
```

Ex.lo

```
while (true) {
```

```
....
```

```
if (isInterrupted() )
```

```
    break;
```

```
}
```

```
} // run
```

```
} // classe MinhaThread_1
```

```
public class Teste {
```

```
    public static void main (String [] args) {
```

```
        MinhaThread_1 T1 = new MinhaThread_1();
```

```
    }
```

```
}
```

Hipótese 2:

```
public class MinhaThread_2 extends Thread {
```

```
    public void run () {
```

```
        ...
```

```
    }
```

```
}
```

*iniciar a execução da thread na
classe Teste →*

Programação concorrente em Java

```
public class Teste {  
    public static void main (String [] args) {  
        MinhaThread_2 Ta, Tb;  
        Ta = new MinhaThread_2();  
        Tb = new MinhaThread_2();  
        Ta.start();  
        Tb.start();  
    }  
}
```

Se quisermos aceder a variáveis do processo principal?

- Passamos como parâmetros as referências dessas variáveis(objectos)

```
public class MinhaThread extends Thread {  
    ObjectoPartilhado O;  
    public MinhaThread ( ObjectoPartilhado o ) {  
        O = o ;  
        Thread T = new Thread( this);  
        T.start();  
    }  
    public void run() {  
        ...  
        Ö....  
    }  
}
```

*endereço do
↓ objecto partilhado*

aceder ao objecto partilhado

Programação concorrente em Java

Hipótese 3: - se a classe já for subclasse de outra classe?

```
public class MinhaThread_3 implements Runnable {  
    public void run(){  
        ...  
    }  
}
```

classe que implementa o método run()

```
public interface Runnable {  
    public abstract void run();  
}
```

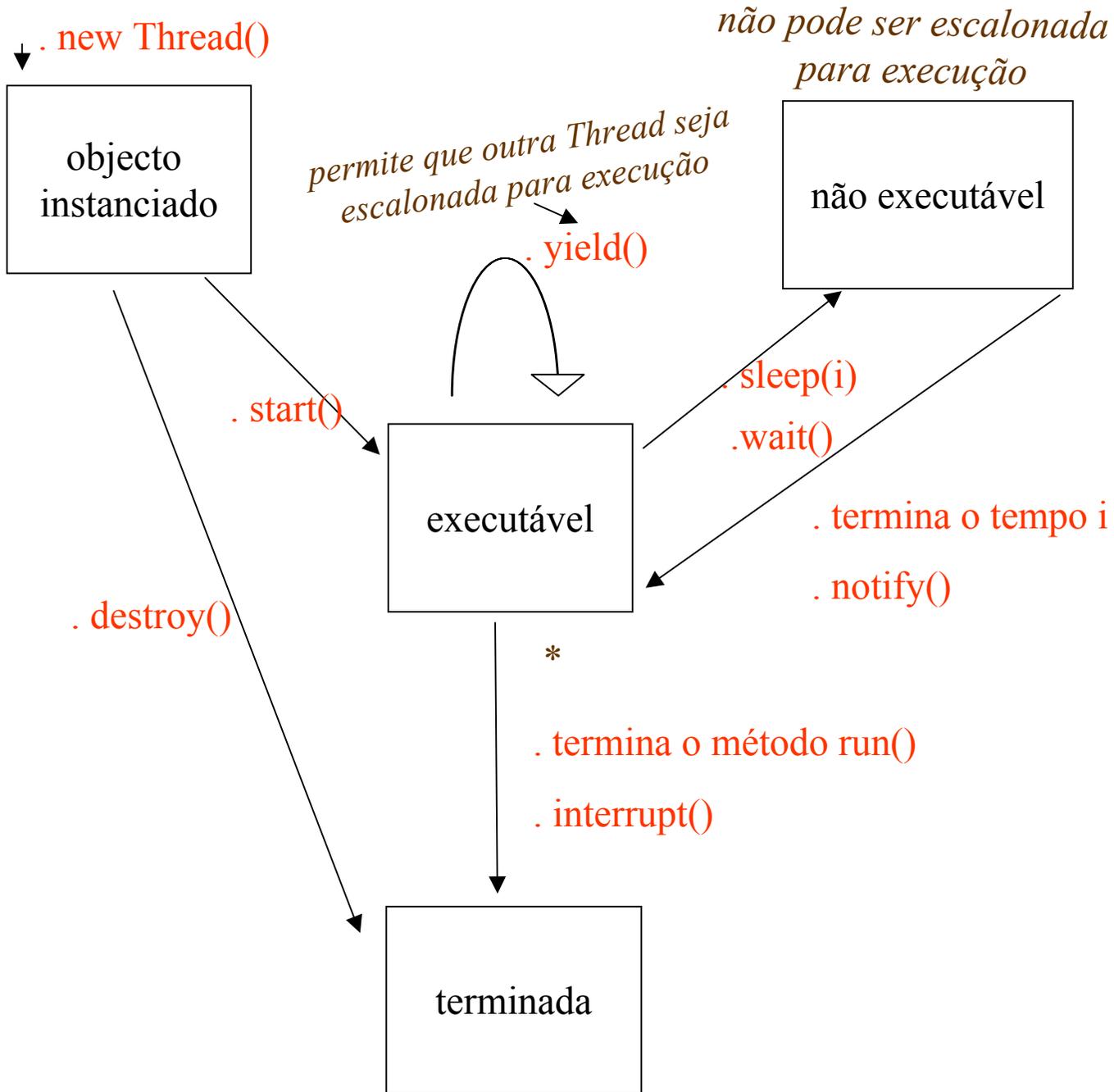
```
public class Teste {  
    public static void main (String [] args) {  
        MinhaThread_3 Tc;  
        Tc = new MinhaThread_3();  
        Thread T = new Thread (Tc);  
        T.start();  
    }  
}
```

↑ "runable object"

↑ iniciar a execução

Programação concorrente em Java

Diagrama de estados possíveis para uma Thread



** uma Thread no estado executável, não está necessariamente em execução, apenas pode ser escalonada para execução*

Programação concorrente em Java

Sincronização de Threads

. mecanismo baseado no conceito de monitor

Existe um **lock** associado a cada objecto *lock de objecto*

e um lock associado a cada classe. *lock de classe*

A instrução

synchronized (expressão)

{ instruções }

referência para o
objecto

a) após calcular a referência para o objecto, mas antes de executar o corpo de instruções:

- adquire o lock associado ao objecto,

(caso este não pertença já a alguma outra thread)

- executa o corpo de instruções

b) Após executar o corpo de instruções

- liberta o lock

*(se a execução do bloco de instruções termina anormalmente
i. é, falha a meio, o lock é libertado)*

Programação concorrente em Java

Notas:

1. Um método pode ser declarado como **synchronized**,
(comporta-se como se estivesse contido numa instrução synchronized)
2. O facto de uma Thread adquirir o lock associado a um objecto, não impede que outras Threads acessem aos campos do objecto ou possam invocar métodos não sincronizados.
3. Se o método sincronizado é um método de instância:
 - a Thread adquire o lock do objecto (associado a this);
 - todas as Threads que tentem executar esse mesmo método no mesmo objecto terão que esperar, competindo pela aquisição do lock.
4. Se o método sincronizado é um método de classe:
 - a Thread adquire o lock da classe;
 - todas as Threads que tentam executar esse método, em qualquer objecto da classe, terão que esperar que o lock seja libertado.

Programação concorrente em Java

Exemplo 1:

```
public class Exemplo {
    private int x;
    private static int s;
    public synchronized void M1()
    { x++; }
    public static synchronized void M2()
    { s++; }
}
```

É equivalente a,

```
public class Exemplo {
    private int x;
    private static int s;
    public void M1() {
        synchronized (this)
        { x++; }
    }
    public static void M2() {
        try { synchronized (Class.forName( "Exemplo" ) )
            { s++; }
        }
        catch (ClassNotFoundException e) { ... }
    }
}
```

Programação concorrente em Java

Exemplo 2:

i) public class Exemplo2{

private int a = 1, b = 2;

public void ab()

{ a = b; }

public void ba()

{ b = a; }

a=2,b=1 | a=2,b=2 | a=1,b=1

}

Sejam duas Threads, T1 e T2. Supondo que

T1 invoca o método ab e

T2 invoca o método ba no mesmo objecto,

quais são os possíveis valores finais para a e b em cada caso, i) e ii) ?

ii) public class Exemplo3{

private int a = 1, b = 2;

public **synchronized** void ab()

{ a = b; }

public **synchronized** void ba()

{ b = a; }

}

a=2,b=2 | a=1,b=1

Programação concorrente em Java

*E nos casos iii, iv e v
quais são os outputs
possíveis?*

```
iii) public class Exemplo4 {  
    private int a = 1, b = 2;  
    public void M1()  
    { a = 3; b = 4; }  
    public void M2()  
    { System.out.println ("a=" + a + "b=" + b); }  
}
```

a=3,b=4 | a=3,b=2 | a=1,b=2 | a=1,b=4

```
iv) public class Exemplo5 {  
    private int a = 1, b = 2;  
    public synchronized void M1()  
    { a = 3; b = 4; }  
    public void M2()  
    { System.out.println ("a=" + a + "b=" + b); }  
}
```

a=3,b=4 | a=3,b=2 | a=1,b=2 | a=1,b=4

*- o facto de um método ser sincronizado não faz com
que se comporte como se fosse uma instrução atómica*

```
v) public class Exemplo6 {  
    private int a = 1, b = 2;  
    public synchronized void M1()  
    { a = 3; b = 4; }  
    public synchronized void M2()  
    { System.out.println ("a=" + a + "b=" + b); }  
}
```

a=3,b=4 | a=1,b=2

Transferência de controle entre Threads

Os métodos

wait(),

notify()

notifyAll(), da classe Object,

Permitem a transferência de controlo de uma Thread para outra.

Só podem ser executados por uma Thread que detenha o lock do objecto

Método wait():

*public final void wait() throws InterruptedException,
IllegalMonitorStateException;*

public final void wait(long timeout) throws ...

public final void wait(long timeout, int nanos) throws ...

“excepção verificável”



“excepção não verificável”



A Thread que executa o wait() suspende-se a si própria.

Cada objecto, além de ter um lock associado, tem também um “wait set”, que contém a referência de todas as Threads que executam um wait sobre o objecto.

Programação concorrente em Java

a) Quando o objecto é criado,

- o “wait set” está vazio

Seja uma Thread T que adquiriu o lock do objecto,

b) Ao executar o wait(), T:

1 – é adicionada ao “wait set”

2 – é suspensa (passa ao estado “não executável”)

3 – liberta o lock que detinha

c) A Thread T permanece no estado “não executável” até que:

– alguma outra Thread invoque o método notify() para esse objecto, e T seja a Thread escolhida para ser notificada,

ou

– alguma outra Thread invoque o método notifyAll para esse objecto,

ou

– alguma outra Thread invoque o método interrupt() na Thread T,

ou

– se a invocação do método wait pela Thread T especificava um intervalo de tempo, e esse tempo tempo expirou

Após c)

1 – T é removida do “wait set”
(*volta ao estado executável*)

2 – Quando T é escalonada para execução, volta a adquirir o lock sobre o objecto

3 – T retorna ao ponto onde foi invocado o wait

Métodos notify() e notifyAll():

```
public final void notify() throws  
IllegalMonitorStateException;
```

Se o “wait set” não está vazio,
é escolhida arbitrariamente uma Thread para ser retirada do conjunto e passar ao estado executável.

Se o “wait set” está vazio não tem qualquer efeito.

```
public final void notifyAll() throws  
IllegalMonitorStateException;
```

Todas as Threads do “wait set” são removidas e passam ao estado executável.

Quando a Thread que executou o notifyAll() libertar o lock do objecto, poderão ser reescalonadas

Nota: o processo que executa o notify, não é suspenso

Programação concorrente em Java

Caso de estudo:

1 - “Multithreaded server”

Suponhamos um servidor, que comunica a cada cliente que o solicita, a data e a hora do sistema:

Cliente

```
import java.net.*;
import java.io.*;
public class Cliente {
    private Socket s;
    public Cliente() {
        try {
            s = new Socket (“127.0.0.1”, 5432);
            InputStream in = s.getInputStream();
            BufferedReader is = new BufferedReader(
                new InputStreamReader(In));
            System.out.println(is.readLine());
            s.close();
        }
        catch ( IOException e)
        { System.out.println(e.getMessage());}
    }
    public static void main (String args[]) {
        Cliente c = new Cliente();
    }
}
```

Programação concorrente em Java

Servidor:

```
public class Servidor {
    private ServerSocket ss;
    private Socket s;
    private Connection c;
    public Servidor(){
        try {
            ss = new ServerSocket (5432);
        }
        catch ( IOException e){ ...}
        try {
            while (true) {
                s = ss.accept();
                c = new Connection (s);
            }
        }
        catch (IOException e) {...}
    }
    public static void main (String args[]){
        Servidor dataHora = new Servidor();
    }
}
```

← aceita uma ligação pedida por um cliente

- O pedido é tratado por uma nova Thread;

- O ServerSocket pode aceitar outra(s) ligações

Programação concorrente em Java

```
public class Connection extends Thread {  
    private Socket S;  
    public Connection ( Socket s) {  
        S = s;  
        Thread T = new Thread (this);  
        T.start();  
    }  
    public void run () {  
        try {  
            PrintWriter os = new PrintWriter( S.getOutputStream(),  
                                                true);  
            os.println ( “A data e hora do sistema: ” +  
                        new java.util.Date().toString());  
        }  
        catch ( IOException e) {...}  
    }  
}
```

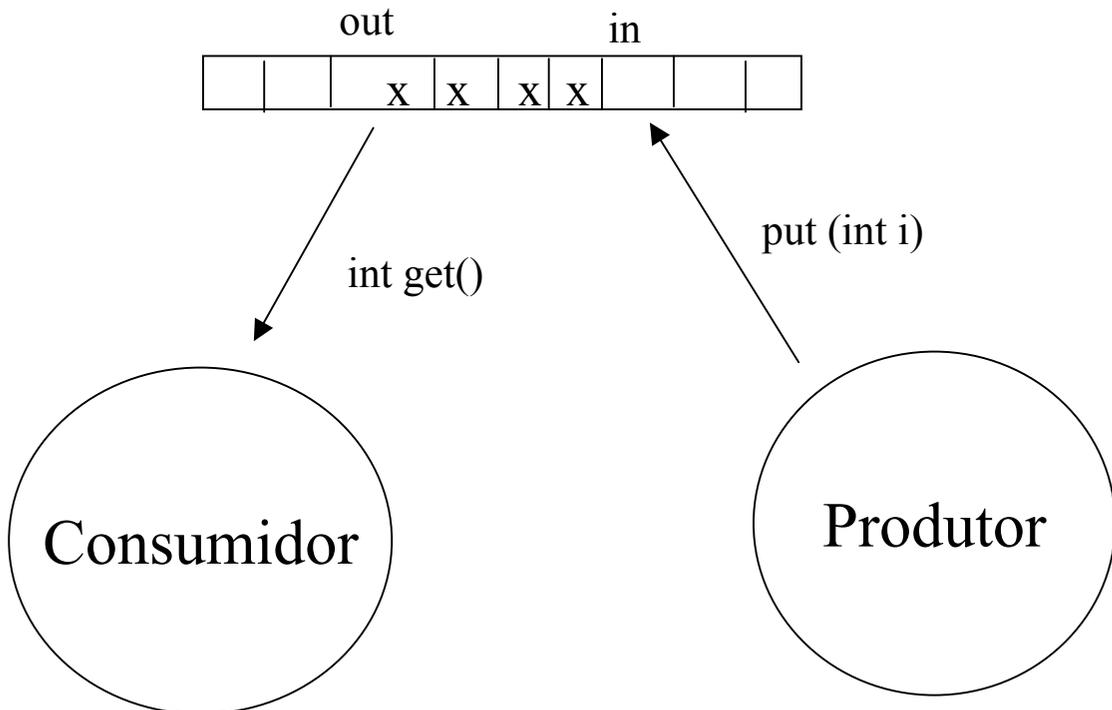
Programação concorrente em Java

Caso de estudo:

2 – O problema do Produtor / Consumidor

Dois processos, o Produtor e o Consumidor, partilham um bloco de memória comum (um “buffer”). O Produtor gera dados que coloca no buffer, de onde são retirados pelo Consumidor. Os items de dados têm que ser retirados pela mesma ordem por que foram colocados.

Implementar o problema considerando que o buffer tem capacidade finita, o que significa que o Produtor é suspenso quando o buffer está cheio, analogamente, o Consumidor é suspenso quando o buffer está vazio.



Nota: a existência de um buffer permite que variações na velocidade a que os dados são produzidos não tenham reflexo directo na velocidade a que os mesmos são consumidos.

Programação concorrente em Java

```
public class Produtor extends Thread {
    Buffer B;
    public Produtor (Buffer b) {
        B = b;
        Thread P = new Thread (this);
        P.start();
    }
    public void run () {
        int i ;
        while ( true){
            i = (int) (Math.random()*100);
            B.put(i);
        }
    }
}

public class Consumidor extends Thread {
    Buffer B;
    public Produtor (Buffer b) {
        B = b;
        Thread C = new Thread (this);
        C.start();
    }
    public void run () {
        int i ;
        while ( true){
            i = B.get();
            System.out.println( "Valor Consumido: " + i );
        }
    }
}
```

Programação concorrente em Java

```
public class teste {  
    public static void main (String args[]) {  
        Buffer B = new Buffer(100);  
        Produtor P1, P2;  
        Consumidor C1, C2;  
        P1 = new Produtor (B);  
        P2 = new Produtor(B);  
        C1 = new Consumidor (B);  
        C2 = new Consumidor(B);  
    }  
}
```

```
public class Buffer {  
    private int b[];  
    private int dim, in, out, elementos;  
    public Buffer (int n) {  
        dim =n;  
        b = new int [dim];  
        in = 0;  
        out = 0;  
        elementos = 0;  
    }  
    private boolean cheio(){  
        return (elementos == dim);  
    }  
    private boolean vazio(){  
        return (elementos == 0);  
    }  
}
```

Programação concorrente em Java

```
public synchronized int get () {  
    while ( vazio() ) {  
        try {  
            wait ();  
        }  
        catch (InterruptedException e)  
        { ... }  
    }  
  
    int i = b[out];  
  
    out = (out + 1 )% dim;  
  
    elementos --;  
  
    notifyAll();  
  
    return(i);  
}
```

Programação concorrente em Java

```
public synchronized void put (int i){  
    while ( cheio() ) {  
        try {  
            wait ();  
        }  
        catch (InterruptedException e)  
        { ...}  
    }  
  
    b[in] = i;  
  
    in = (in + 1 )% dim;  
  
    elementos ++;  
  
    notifyAll();  
}  
  
} // class Buffer
```

Programação concorrente em Java

Nos métodos anteriores, porque não usar uma instrução `if` em vez de um `while`:

```
while (cheio())
```

```
    wait()
```

?

```
if (cheio())
```

```
    wait()
```

```
while (vazio())
```

```
    wait()
```

```
if (vazio())
```

```
    wait()
```

Suponham-se 2 Produtores, P1 e P2, 2 Consumidores, C1,C2, e a sequência de execução:

1 - num dado instante o Buffer B está vazio

2 – C1 executa um `get (B.get())` -----C1 é suspenso

3 – C2 executa um `get (B.get())` -----C2 é suspenso

4 – P1 executa um `put (B.put(i))` -----última instrução é `notifyAll()`

5 - C1 é retirado do "wait set", escalonado para execução, prossegue com o `get`, retira último elemento do Buffer, executa o `notifyAll()`

6 - C2 é retirado do "wait set", escalonado para execução, retira o último ???? erro !!!

Programação concorrente em Java

Caso de estudo:

3 – Implementação de uma classe Semáforo em Java

Definição de semáforo (Dijkstra, 1968):

É uma variável, s , que apenas pode assumir valores positivos ou nulos e à qual está associada uma fila de processos.

Após a inicialização da variável apenas são permitidas as operações atómicas:

wait (s) : Se ($s > 0$) Então

$$s = s - 1$$

Senão

- o processo que executa o wait é suspenso

signal (s) : Se (um processo foi suspenso por execução de um wait anterior) Então

- é restabelecido

Senão

$$s = s + 1$$

Programação concorrente em Java

```
public class Semaforo {
    private int s;
    public Semaforo (int i){
        s = i;
    }
    public synchronized void semWait () {
        while (s <= 0 ) {
            try {
                wait();
            }
            catch (InterruptedException e)
            {...}
        }
        s = s - 1;
    }

    public synchronized void semSignal () {
        s = s + 1 ;
        notify();
    }
}
```