

Programação concorrente em Java

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 objeto
(isto é, dentro de um bloco synchronized)**

Método wait():

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

“excepção verificável”

“excepção não verificável”

Programação concorrente em Java

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

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

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 objeto.

a) Quando o objecto é criado,

- o “wait set” está vazio

Seja uma Thread T que adquiriu o lock do objeto,

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 objeto, e T seja a Thread escolhida para ser notificada,

ou

– alguma outra Thread invoque o método `notifyAll` para esse objeto,

ou

Programação concorrente em Java

(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 expirou.

Após c)

1 – T é removida do “wait set”

(volta ao estado executável)

2 – Quando T é escalonada para execução,

volta a competir/adquirir pelo lock sobre o objeto

3 – T retorna ao ponto imediatamente após a invocação do `wait`.

Programação concorrente em Java

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 reescaloadas.

Nota: o processo que executa o notify, não é suspenso (se possível, o notify deve ser a última instrução do método)

Caso de estudo:

2 – O problema do Produtor / Consumidor

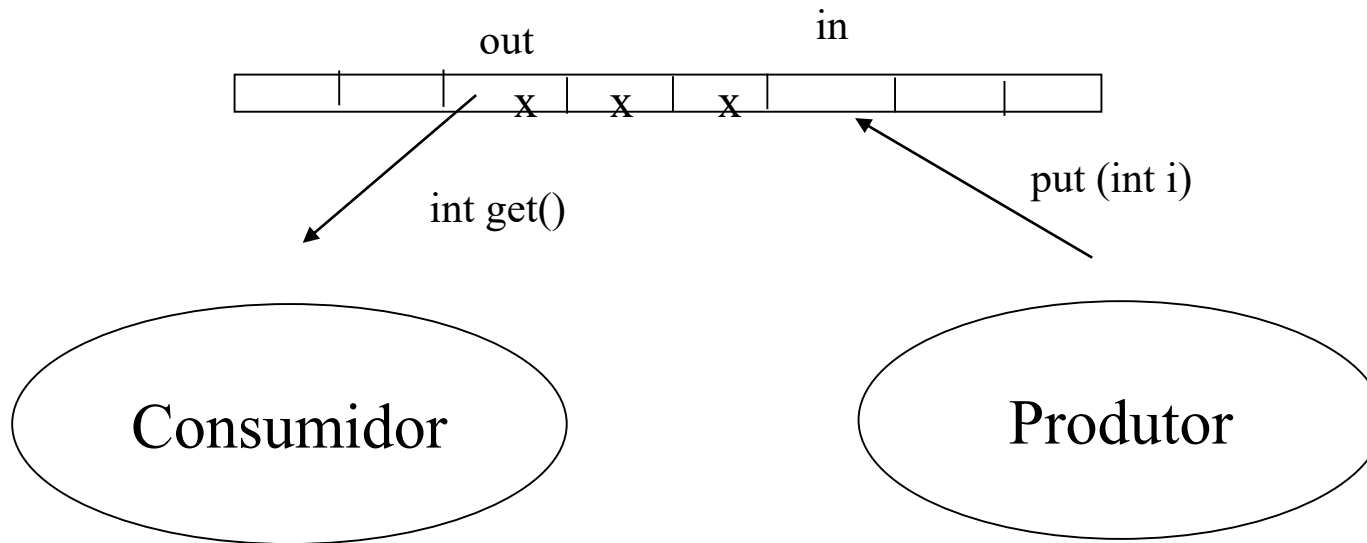
Dois processos, o Produtor e o Consumidor, partilham um bloco de memória comum (um “buffer”). O Produtor gera dados que coloca no buffer, de onde são retirados pelo Consumidor. Os itens de dados têm que ser retirados pela mesma ordem por que foram colocados.

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.

Programação concorrente em Java

Caso de estudo:

2 – O problema do Produtor / Consumidor (cont)



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) {
        super();
        B = b;
        super.start();
    }

    public void run () {
        int i ;
        while ( true) {
            i = (int) (Math.random()*100);
            B.put(i);
        }
    }
}
```


Programação concorrente em Java

```
public class Consumidor extends Thread {  
    Buffer B;  
  
    public Consumidor (Buffer b) {  
        super();  
        B = b;  
        super.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);  
    }  
}
```

Programação concorrente em Java

```
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:

Programação concorrente em Java

- 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,
prosegue com o get, retira último elemento do Buffer,
executa o notifyAll()
- 6 - C2 é 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();  
    }  
}
```

Testar ...

Programação concorrente em Java

Exercício 1:

1 – Pretende-se uma aplicação para gerir o dinheiro em **caixa de um clube recreativo**.

Para isso construa as seguintes classes:

a) Uma classe *ContaBancaria* que deverá permitir:

- Consultar o saldo disponível em cada instante;
- Simular um levantamento, cada vez que um utilizador pretenda levantar uma quantia menor ou igual à existente;
- Simular um depósito.

b) Uma classe *Financiador* que periodicamente vai depositando quantias num objeto do tipo *ContaBancaria*.

Programação concorrente em Java

Exercício 1 (cont.):

- c) Uma classe Utilizador que periodicamente vai levantando quantias do objeto do tipo *ContaBancaria*.

- d) Para testar as classes anteriores construa uma classe teste em que um objeto do tipo *ContaBancaria* seja partilhado concorrentemente por um objecto do tipo *Financiador* e por pelo menos 3 objectos do tipo *Utilizador*.

- e) **E se quiser suspender o utilizador sempre que o saldo da conta for inferior ao valor do levantamento?**

Exercício 2: “Readers-Writers Problem”

a) Construa uma classe em Java, RW, que possua um atributo do tipo `int`, XPTO, e dois métodos: ler e escrever. O método ler deve devolver o valor da variável XPTO; o método escrever deve adicionar o valor 100 à variável XPTO e seguidamente subtrair o mesmo valor à variável XPTO.

b) Pretende-se que um objecto da classe RW seja partilhado por vários processos

(Threads) de dois tipos:

- processos Leitores – que lêem o valor da variável XPTO usando o método ler;
 - processos Escritores – que alteram a variável XPTO usando o método escrever.
- Construa as classes Leitor e Escritor. Cada uma destas classes deve ter uma Thread de execução própria em que, num ciclo infinito, vão respectivamente lendo e alterando valores do objecto partilhado.

Exercício 2: “Readers-Writers Problem”

c) Construa uma classe de teste que crie um objecto do tipo RW, 3 objectos do tipo Leitor e 2 objectos do tipo Escritor.

Estude o comportamento do seu programa

d) Pretende-se que modifique as classes anteriores de forma a **que os vários processos Leitores possam executar concorrentemente o método ler, mas que quando um processo Escritor executar o método escrever o faça em exclusão mútua**. Isto é, quando um processo está a escrever, nenhum outro pode em simultâneo ler ou escrever a variável XPTO.

Programação concorrente em Java

Exercício 2 d) – (Uma solução)

```
public class RW {  
    private int nl; // nl representa o número de leitores em cada instante  
    private int XPTO;  
  
    private int ler () {  
        return XPTO;  
    }  
  
    public synchronized void escrever () {  
        while ( nl > 0 ) {  
            try {  
                wait();  
            } catch (InterruptedException e) { ... }  
        }  
  
        XPTO = XPTO + 100;  
        XPTO = XPTO - 100  
        notifyAll ();  
    }  
}
```

Programação concorrente em Java

Exercício 2 – (Uma solução ...)

```
public synchronized void startLer () {  
    nl ++;  
}  
  
public synchronized void endLer () {  
    nl --;  
    if (nl == 0) notifyAll ();  
}  
  
} // fim da classe RW
```

Programação concorrente em Java

Exercício 2 d – (Uma solução ...)

```
public class Leitor extends Thread {  
    private RW rw ;  
    public Leitor (RW rw) {  
        super();  
        this.rw = rw;  
        start ();  
    }  
    public void run () {  
        int i;  
        while (true) {  
            rw.startLer () ;  
            i = rw.Ler();  
            System.out.println ( “Leitor: “ + i);  
            rw.endLer();  
        }  
    }  
}
```


Programação concorrente em Java

Exercício 2 – (Uma solução ...)

```
public class Escritor extends Thread {  
    private RW rw ;  
    public Escritor (RW rw) {  
        super();  
        this.rw = rw;  
        start ();  
    }  
    public void run () {  
        int i;  
        while (true) {  
            rw.escrever();  
        }  
    }  
}
```

Programação concorrente em Java

Exercício 2 – (Uma solução ...)

```
public class Teste {  
  
public static void main ( String [] args) ) {  
  
    RW rw = new RW ();  
    Leitor l1 = new Leitor (rw);  
    Leitor l2 = new Leitor (rw);  
    Leitor l3 = new Leitor (rw);  
    Escritor e1 = new Escritor (rw);  
    Escritor e2 = new Escritor (rw);  
}  
}
```

Programação concorrente em Java

Exercício 3

Suponha dois processos **p1** e **p2** que partilham uma variável comum, **variavelPart**.

Pretende-se construir um exemplo que ilustre a violação de uma secção crítica, sem usar qualquer tipo de mecanismo de sincronização,

- Considere que o processo **p1** possui duas variáveis locais, **x** e **y**, inicializadas com valores simétricos, e que dentro de um ciclo infinito transfere a quantidade armazenada em **variavelPart** de **x** para **y**. O processo 2 vai, em cada iteração, incrementar a variável partilhada.
- Pretende-se que a condição $x + y = 0$ seja verdadeira durante toda a execução do programa. Quando, no processo **p1**, se detecta que a secção crítica foi violada (porque $x + y \neq 0$) o processo deve terminar e acabar o programa.

Programação concorrente em Java

Exercício 3 (cont.)

- Supondo a estrutura que se segue para os processos P1 e P2, comece por criar duas classes que permitam criar os processos (Threads) P1 e P2. Estes dois processos deverão, partilhar um objeto com um valor inteiro.

Processo 1

```
x = M; y = - M;
While (true){
  //secção crítica 1
  x = x - variavelPart;
  y = y + variavelPart;
  <parte restante 1>
  if (x+y != 0 ){
    print “Secção crítica violada”
    break;
  }//fim do if
  ...
} // fim do While
```

Processo 2

```
...
While (true){
  //secção crítica 2
  variavelPart =
    variavelPart +1;
  <parte restante 2>
}
...
```

Programação concorrente em Java

Exercício 3 (cont.)

b) Construa uma classe de teste que, instanciando os processos p1 e p2, permita simular a violação da secção crítica.

c) Para que o processo p2 termine, após a violação da secção crítica, transforme-o numa Thread daemon.

d) Modifique o programa de maneira a garantir a execução de cada secção crítica em exclusão mútua.

d1) Usando a instrução synchronized.

d2) Usando a classe Semáforo (página 17)

semWait() **Sec.Critica** sem Signal()

Programação concorrente em Java

Exercício 3 – (Uma solução para **d1**)

```
public class P1 extends Thread {  
    private int[] vp;  
    private int x = 1000000, y = -1000000;  
    public P1 (int [] vp){  
        super();  
        this.vp = vp;  
        start();  
    }  
}
```



Programação concorrente em Java

Exercício 3 – (Uma solução ...)

```
public void run () {
    int a;
    while (true) {
        //secção crítica 1
        synchronized (vp) {
            x = x + vp[0];
            y = y - vp[0];
        }
        // <parte restante 1>
        if (x+y != 0 ) {
            System.out.println(" Secção crítica violada" );
            break;
        } //fim do if
    } // end while
} // end run
} // end P1
```

Programação concorrente em Java

Exercício 3 – (Uma solução ...)

```
public class P2 extends Thread {  
    private int [] vp;  
    public P2 (int[] vp) {  
        super();  
        this.vp = vp;  
        setDaemon(true)  
        start();  
    }  
}
```


Programação concorrente em Java

Exercício 3 – (Uma solução ...)

```
public void run(){
    int a;
    while (true) {
        //secção crítica 2
        synchronized (vp){
            vp[0] = vp[0] + 1;
        }
        //<parte restante 2>
        System.out.println("*****P2" + vp[0]);
    } // while
} // run
} P2
```

Programação concorrente em Java

Exercício 3 – (Uma solução)

```
public class MainP1P2 {  
  
    public static void main(String[] args) {  
        int[] vp = new int [1];  
        vp[0] = 0;  
  
        P2 p2 = new P2(vp);  
        P1 p1 = new P1 (vp);  
  
        System.out.println(vp[0]);  
    }  
}
```

d2) ????

Programação concorrente em Java

Exercício 4

Suponha a classe exemplo esquematizada abaixo.

```
Class Exemplo {  
    ...  
    public void mX() { ... }  
}
```

Supondo que se pretende construir uma aplicação cliente-servidor, em que o processo servidor contém um objecto partilhado do tipo Exemplo e para cada processo cliente que lhe acede lança uma nova thread que irá executar o método mX.

- Construa a classe ThreadExemplo de que será criada uma instância sempre que um cliente acede ao servidor. Uma instância de ThreadExemplo terá uma sequência de execução própria, deverá invocar o método mX do objecto partilhado e garantir que nunca haverá **mais de três clientes** em simultâneo a executar o método mX. Quando um quarto cliente tenta executar o método, a thread correspondente deverá ser suspensa até que menos de três clientes estejam a executar mX.

Que modificações terá de fazer na classe Exemplo.