Programação Concorrente

TRABALHO PRÁTICO

Marta Pereira - up202105713
Biana Oliveira - up202000139

Exclusão mútua

Usada para garantir que apenas um processo ou thread possa aceder a um recurso compartilhado ou seção crítica num determinado momento.

Propósito

Manter a integridade dos dados e preservar a consistência dos recursos compartilhados em ambientes multi-thread ou multi-processo. Diminuindo assim, o risco de operações conflitantes que levam a resultados incorretos ou comportamentos imprevisíveis.

Algoritmo de Hyman

Exemplo de algoritmo que garante exclusão mútua, por isso, garante que múltiplos processos ou threads não acedem simultaneamente a uma seção crítica do código.

Implementação CCS

```
B1f := b1wf?.B1f + b1rf!.B1f + b1wt?.B1t
B1t := b1wt?.B1t + b1wf?.B1f + b1rt!.B1t

B2f := b2wf?.B2f + b2rf!.B2f + b2wt?.B2t
B2t := b2wt?.B2t + b2wf?.B2f + b2rt!.B2t

K1 := kw1?.K1 + kw2?.K2 + kr1!.K1
K2 := kw2?.K2 + kw1?.K1 + kr2!.K2

P1 := b1wt!.P11
P11 := kr1?.P14 + kr2?.P12
P12 := b2rf?.P13 + b2rt?.P12
P13 := kw1!.P11
P14 := enter1.exit1.b1wf!.P1

P2 := b2wt!.P21
P21 := kr2?.P24 + kr1?.P22
P22 := b1rf?.P23 + b1rt?.P22
P23 := kw2!.P21
P24 := enter2.exit2.b2wf!.P2

P := (P1|P2|K1|B1f|B2f) \ {b1rf, b2rf, b1rt, b2rt, b1wf, b2wf, b1wt, b2wt, kr1, kr2, kw1, kw2}
P
```

Thread

É uma sequência de instruções que pode ser executada independentemente por um processador, enquanto compartilha recursos, como memória, com outras threads pertencentes ao mesmo processo.

```
@volatile var k: Int = 1
@volatile var b1: Boolean = true
@volatile var b2: Boolean = true

def nonCriticalActions(process: String): Unit = {
    println(s"P$process: Non-critical actions")
    // println(s"${LocalDateTime.now()}: $process")
    Thread.sleep(1000)
}

def criticalActions(process: String): Unit = {
    println(s"P$process: Entering critical section!")
    // println(s"${LocalDateTime.now()}: $process")
    Thread.sleep(1000)
    println(s"P$process: Exiting critical section!")
}

def process (id: Int) : Thread = {
    new Thread(() => {
        while (true) {
            // Non critical actions
            nonCriticalActions(s"$id")
            if (id == 1) b1 = true else b2 = true
            while (k != id) {
                while ((if (id == 1) b2 else b1) == true) {
                    // skip
                }
                k = id
            }
            // Critical actions
            criticalActions(s"$id")
            if (id == 1) b1 = false else b2 = false
        }
    })
}
```

```
P2: Non-critical actions
P1: Non-critical actions
P1: Entering critical section!
P1: Exiting critical section!
P1: Non-critical actions
P2: Entering critical section!
P2: Exiting critical section!
P2: Non-critical actions
P1: Entering critical section!
P1: Exiting critical section!
P1: Non-critical actions
P2: Entering critical section!
P2: Exiting critical section!
P2: Non-critical actions
P1: Entering critical section!
P1: Exiting critical section!
P1: Non-critical actions
```

Locks

Mecanismos de controle de concorrência usados para gerir o acesso a recursos compartilhados por várias threads. Quando uma thread entra num bloco synchronized, ela adquire o lock associado ao objeto. Outras threads que tentam aceder a blocos sincronizados no mesmo objeto são bloqueadas até que o lock seja libertado.

Algoritmo de peterson

```
@volatile var flags: Array[Boolean] = Array(false, false)
@volatile var turn: Int = 0
var N = 4
var lock = new Object()

def nonCriticalActions(process: String): Unit = {
    println(s"P$process: Non-critical actions")
    // println(s"${LocalDateTime.now()}: $process")
    Thread.sleep(1000)
}

def criticalActions(process: String): Unit = {
    println(s"$process: In critical section!")
    // println(s"${LocalDateTime.now()}: $process")
    Thread.sleep(1000)
    println(s"$process: Exiting critical section!")
}

def process (i: Int) : Thread = {
    new Thread(() => {
        val j = 1 - i
        for (n <- 0 until N) {
            flags(i) = true
            turn = j;
            while (flags(j) && turn == j) {}
            // critical section, busy wait
            lock.synchronized {
                println(s"Process $i entering in CS iteration number: $n")
                criticalActions(s"Process $i")
                println(s"Process $i done CS") // unlock
            }
            flags(i) = false
        }
    })
}
```

```
Process 0 entering in CS iteration number: 0
Process 0: In critical section!
Process 0: Exiting critical section!
Process 0 done CS
Process 1 entering in CS iteration number: 0
Process 1: In critical section!
Process 1: Exiting critical section!
Process 1 done CS
Process 0 entering in CS iteration number: 1
Process 0: In critical section!
Process 0: Exiting critical section!
Process 0 done CS
Process 1 entering in CS iteration number: 1
Process 1: In critical section!
Process 1: Exiting critical section!
Process 1 done CS
```

Variáveis atómicas

Permitem operações de leitura e escrita indivisíveis, ou seja quando uma operação atómica é executada numa variável atómica, essa operação ocorre de maneira completa sem possibilidade de interferência de outras operações, garantindo a consistência dos dados em ambientes concorrentes, como em programas multithread.

Em vez de usar locks, os algoritmos lock-free dependem de operações atómicas para garantir que, em algum ponto no tempo, pelo menos uma das threads em execução progredirá. Isso minimiza o tempo gasto em estado de espera e evita problemas comuns associados a locks, como deadlocks.

```
val lock = new AtomicBoolean(false) // false lock está livre
val N = 4

// Dummy critical
def criticalActions(processName: String): Unit = {
  println(s"$processName is executing critical actions")
  Thread.sleep(1000)
}

def process(id: Int): Thread = {
  new Thread(() => {
    for (n <- 0 until N) {
      // Non-critical actions
      println(s"Process $id performing non-critical actions in iteration $n")
      Thread.sleep(1000) // Simulate non-critical work
      // Try to acquire the lock
      while (!lock.compareAndSet(false, true)) {
          // Busy-wait until the lock is acquired
      }

      // Critical section
      try {
          println(s"Process $id in critical section iteration number: $n")
          criticalActions(s"Process $id")
      }
      finally {
          // Release the lock
          println(s"Process $id done with critical section")
          lock.set(false)
      }
    }
  })
}
```

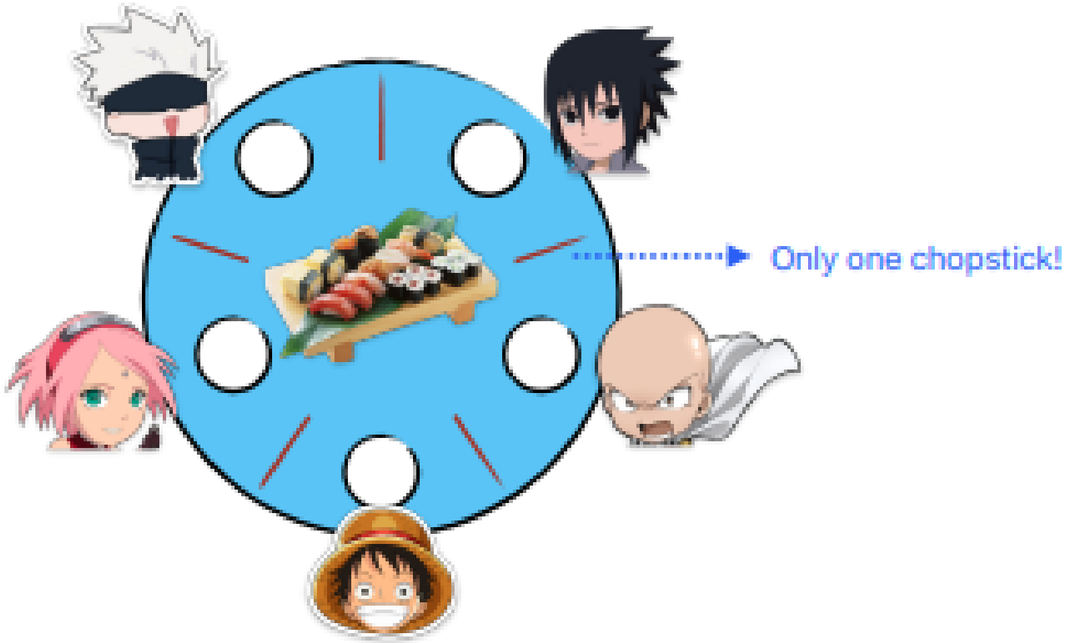
```
Process 0 performing non-critical actions in iteration 0
Process 1 performing non-critical actions in iteration 0
Process 1 in critical section iteration number: 0
Process 1 is executing critical actions
Process 1 done with critical section
Process 0 in critical section iteration number: 0
Process 0 is executing critical actions
Process 1 performing non-critical actions in iteration 1
Process 0 done with critical section
Process 0 performing non-critical actions in iteration 1
Process 1 in critical section iteration number: 1
Process 1 is executing critical actions
Process 1 done with critical section
Process 1 performing non-critical actions in iteration 2
Process 0 in critical section iteration number: 1
Process 0 is executing critical actions
Process 0 done with critical section
Process 1 in critical section iteration number: 2
Process 1 is executing critical actions
Process 0 performing non-critical actions in iteration 2
Process 1 done with critical section
Process 0 in critical section iteration number: 2
Process 0 is executing critical actions
Process 1 performing non-critical actions in iteration 3
Process 0 done with critical section
Process 1 in critical section iteration number: 3
Process 1 is executing critical actions
Process 0 performing non-critical actions in iteration 3
Process 1 done with critical section
Process 0 in critical section iteration number: 3
Process 0 is executing critical actions
Process 0 done with critical section
```

Implementação de Threads

A utilização de threads é fundamental em programação concorrente, especialmente, devido a execução eficiente de múltiplas tarefas simultaneamente.

Um problema do Sushi Meal é um exemplo de utilização das threads.

O problema é a representação de 5 amigos que realizam ações de conversar, pegar o prato com sushi, pegar o palitinho a esquerda, pegar o palitinho a direita, devolver o prato com sushi para o centro da mesa, comer e devolver os palitinhos.



Implementação Variância 1 CSS

```
F1f := fome1wf?.F1f + fome1rf!.F1f +
fome1wt?.F1t
F1t := fome1wt?.F1t + fome1wf?.F1f +
fome1rt!.F1t
F2f := fome2wf?.F2f + fome2rf!.F2f +
fome2wt?.F2t
F2t := fome2wt?.F2t + fome2wf?.F2f +
fome2rt!.F2t
F3f := fome3wf?.F3f + fome3rf!.F3f +
fome3wt?.F3t
F3t := fome3wt?.F3t + fome3wf?.F3f +
fome3rt!.F3t
F4f := fome4wf?.F4f + fome4rf!.F4f +
fome4wt?.F4t
F4t := fome4wt?.F4t + fome4wf?.F4f +
fome4rt!.F4t
F5f := fome5wf?.F5f + fome5rf!.F5f +
fome5wt?.F5t
F5t := fome5wt?.F5t + fome5wf?.F5f +
fome5rt!.F5t

Ka1 := kaw1?.Ka1 + kaw2?.Ka2 + kar1!.Ka1
Ka2 := kaw2?.Ka2 + kaw1?.Ka1 + kar2!.Ka2
Kb2 := kbw2?.Kb2 + kbw3?.Kb3 + kbr2!.Kb2
Kb3 := kbw3?.Kb3 + kbw2?.Kb2 + kbr3!.Kb3
Kc3 := kcw3?.Kc3 + kcw4?.Kc4 + kcr3!.Kc3
Kc4 := kcw4?.Kc4 + kcw3?.Kc3 + kcr4!.Kc4
Kd4 := kdw4?.Kd4 + kdw5?.Kd5 + kdr4!.Kd4
Kd5 := kdw5?.Kd5 + kdw4?.Kd4 + kdr5!.Kd5
Ke5 := kew5?.Ke5 + kew1?.Ke1 + ker5!.Ke5
Ke1 := kew1?.Ke1 + kew5?.Ke5 + ker1!.Ke1
```

```
P1 := conversation1.fome1wt!.P11
P11 := kar1?.P14 + kar2?.P12
P12 := fome2rf?.P13 + fome2rt?.P12
P13 := kaw1!.P11
P14 := take_sticka.P15
P15 := ker1?.P18 + ker2?.P16
P16 := fome5rf?.P17 + fome5rt?.P16
P17 := kew1!.P15
P18 := take_sticke.eat.fome1wf!.P1

P := (P1|P2|P3|P4|P5)
P
```

Implementação Variância 2 CSS

```
//proceso para o amigo canhoto
P1 := conversation1.fome1wt!.P11
P11 := ker1?.P14 + ker2?.P12
P12 := fome5rf?.P13 + fome5rt?.P12
P13 := kew1!.P11
P14 := take_sticke.P15
P15 := kar1?.P18 + kar2?.P16
P16 := fome2rf?.P17 + fome2rt?.P16
P17 := kaw1!.P15
P18 := take_sticka.eat.fome1wf!.P1

P := (P1|P2|P3|P4|P5)
P
```

Implementação Variância 3 CSS

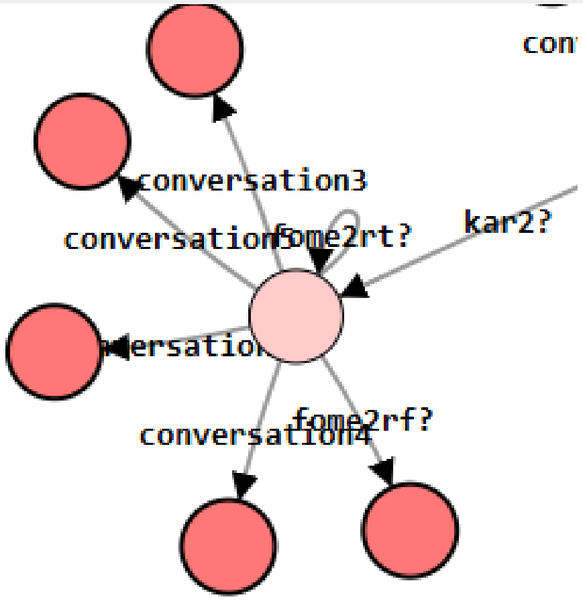
```
T0 := tw0?.T0 + tw1?.T1 + tr0!. T0
T1 := tw1?.T1 + tw0?.T0 + tr1!.T2

P1 := conversation1.P111
P111 := tr0?.P112 + tr1?.P111
P112 := tw1!.grab_sushi_tray.fome1wt!.P11
P11 := kar1?.P14 + kar2?.P12
P12 := fome2rf?.P13 + fome2rt?.P12
P13 := kaw1!.P11
P14 := take_sticka.P15
P15 := ker1?.P18 + ker2?.P16
P16 := fome5rf?.P17 + fome5rt?.P16
P17 := kew1!.P15
P18 := take_sticke.return_sushi_tray.P19
P19 := tw0!.P110
P110 := eat.fome1wf!.P1

P := (P1|P2|P3|P4|P5)
P
```

DeadLocks

Um deadlock é uma situação em sistemas concorrentes onde dois ou mais threads ou processos ficam permanentemente bloqueados, esperando uns pelos outros para liberar recursos.



$$\dfrac{\dfrac{\dfrac{}{fome2rt?.P12 \xrightarrow{fome2rt?} P12}\,\text{prefix}}{fome2rf?.P13 + fome2rt?.P12 \xrightarrow{fome2rt?} P12}\,\text{choice-r} \qquad \Gamma(P12) = fome2rf?.P13 + fome2rt?.P12}{P12 \xrightarrow{fome2rt?} P12}\,\text{rec}$$

$$\dfrac{P12 \xrightarrow{fome2rt?} P12}{P12 \mid P2 \xrightarrow{fome2rt?} P12 \mid P2}\,\text{par-l}$$

$$\dfrac{P12 \mid P2 \xrightarrow{fome2rt?} P12 \mid P2}{P12 \mid P2 \mid P3 \xrightarrow{fome2rt?} P12 \mid P2 \mid P3}\,\text{par-l}$$

$$\dfrac{P12 \mid P2 \mid P3 \xrightarrow{fome2rt?} P12 \mid P2 \mid P3}{P12 \mid P2 \mid P3 \mid P4 \xrightarrow{fome2rt?} P12 \mid P2 \mid P3 \mid P4}\,\text{par-l}$$

$$\dfrac{P12 \mid P2 \mid P3 \mid P4 \xrightarrow{fome2rt?} P12 \mid P2 \mid P3 \mid P4}{P12 \mid P2 \mid P3 \mid P4 \mid P5 \xrightarrow{fome2rt?} P12 \mid P2 \mid P3 \mid P4 \mid P5}\,\text{par-l}$$

IMPLEMENTANDO VARIAÇÃO 3 COM SCALA

```
    // @volatile garante que as variáveis
sejam visíveis a todos os threads de
forma consistente
    @volatile var t: Int = 0
    @volatile var ka: Int = 1
    @volatile var kb: Int = 2
    @volatile var kc: Int = 3
    @volatile var kd: Int = 4
    @volatile var ke: Int = 5
    @volatile var fome1: Boolean = false
    @volatile var fome2: Boolean = false
    @volatile var fome3: Boolean = false
    @volatile var fome4: Boolean = false
    @volatile var fome5: Boolean = false
```

```
// Função que simula ações não críticas
    def conversation(process: String):
Unit = {
        println(s"$process: conversation
actions")
        Thread.sleep(1000) // Simula
ações não críticas com sleep
    }
    // Função que simula ações críticas
    def eat(process: String): Unit = {
        println(s"$process: eating..")
        Thread.sleep(1000) // Simula
ações críticas com sleep
    }
```

```
 class P1 extends Thread {
        override def run () : Unit = {
            while (true) {
                conversation("P1")
                while (t != 0){
                    // skip
                }
                t = 1;
                println("P1:
grab_sushi_tray")
                fome1 = true
                while (ka != 1) {
                    while (fome2 == true)
{
                        // skip
                    }
                    ka = 1
                }
                //grab_stick_left
                while (ke != 1) {
                    while (fome5 == true)
{
                        // skip
                    }
                    ke = 1
                }
                //grab_stick_right
                println("P1:
return_sushi_tray")
                t = 0;
                eat("P1")
                fome1 = false
            }}}
```

```
P1: conversation actions
P5: conversation actions
P3: conversation actions
P4: conversation actions
P2: conversation actions
P3: grab_sushi_tray
P3: return_sushi_tray
P3: eating..
P5: grab_sushi_tray
P5: return_sushi_tray
P5: eating..
P4: grab_sushi_tray
P1: grab_sushi_tray
P4: return_sushi_tray
P2: grab_sushi_tray
P4: eating..
P1: return_sushi_tray
P1: eating..
P2: return_sushi_tray
P2: eating..
P3: conversation actions
P4: conversation actions
P5: conversation actions
```

QUESTÕES?