Programação Concorrente

Monitores

Introdução

- O semáforo é uma primitiva que não requer **espera-ocupada (busy-wating)**.
- O semáforo é considerado de baixo-nível por não ser estruturado.
- Se fizermos um sistema grande baseado em semáforos, algum programador por esquecer um signal(S), podendo causar um deadlock difícil de detectar.

Introdução

- Monitores proveem uma estrutura primitiva de programação concorrente que concentra a responsabilidade da corretude em módulos.
- Monitores são generalizações de **supervisores** do sistema operacional.
- Nesta aula, monitores serão definidos separadamente para cada objeto, ou grupo de objetos, que queira sincronizar uma tarefa.
- Se operações de um mesmo monitor são chamadas por vários processos, a implementação garante a exclusão mútua. Operações de diferentes monitores podem se entrelaçar.
- Podemos dizer que monitores são uma abstração de programação orientada a objetos, os quais encapsulam suas operações dentro de uma classe (os campos de um monitor são todos privados).

Declarando e Usando Monitores

- Operações

Algorithm 7.1. Atomicity of monitor operations

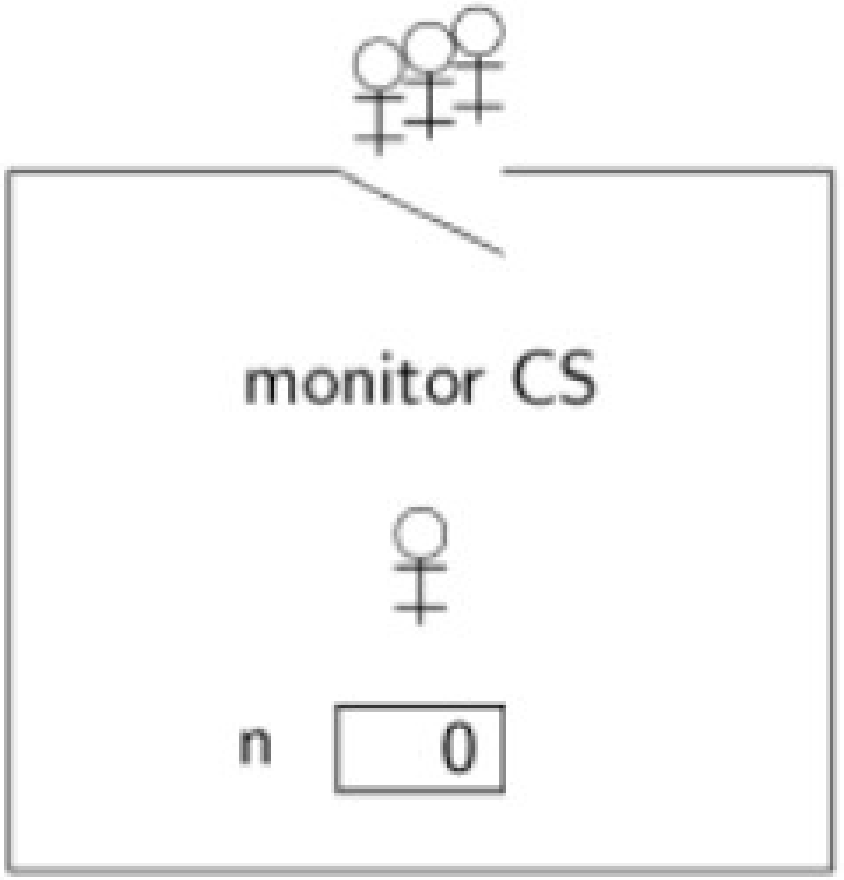
```
monitor CS
  integer n ← 0

  operation increment
    integer temp
    temp ← n
    n ← temp + 1
```

p	q
p1: CS.increment	q1: CS.increment

Declarando e Usando Monitores

- The monitor CS contains one variable **n** and one operation **increment**;
- **Two statements** are contained within this operation, together with the declaration of the **local variable**. The variable **n** is not accessible outside the monitor.
- Two processes, p and q, each call the monitor operation **CS.increment**. Since by definition only one process at a time can execute a monitor operation, we are ensured mutual exclusion in access to the variable, so the only possible result of executing this algorithm is that n receives the value 2.

Declarando e Usando Monitores



Declarando e Usando Monitores

- A seção crítica (o incremento de n) é encapsulada dentro de um objeto monitor.
- O objeto garante que o acesso ao método increment, é **sincronizado,** ou seja, apenas um processo de cada vez pode executar o método.
- A sincronização é **implícita,** não havendo necessidade do uso das operações signal e wait.
- As with semaphores, if there are several processes attempting to enter a monitor, only one of them will succeed. There is no explicit queue associated with the monitor entry, so starvation is possible.

Variáveis de Condição (condition)

- Simulando semáforos.

Algorithm 7.2. Semaphore simulated with a monitor

```
monitor Sem
  integer s ← k
  condition notZero

  operation wait
    if s = 0
       waitC(notZero)
    s ← s - 1

  operation signal
    s ← s + 1
    signalC(notZero)
```

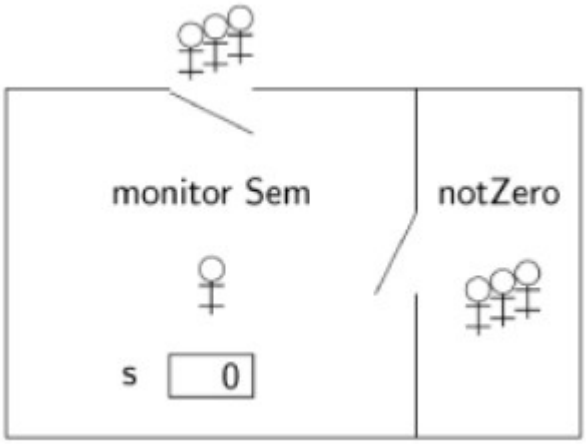
p	q
loop forever non-critical section	loop forever non-critical section
p1: Sem.wait critical section p2: Sem.signal	q1: Sem.wait critical section q2: Sem.signal

waitC(cond)

```
append p to cond
p.state ← blocked
monitor.lock ← release
```

signalC(cond)

```
if cond ≠ empty
    remove head of cond and assign to q
    q.state ← ready
```



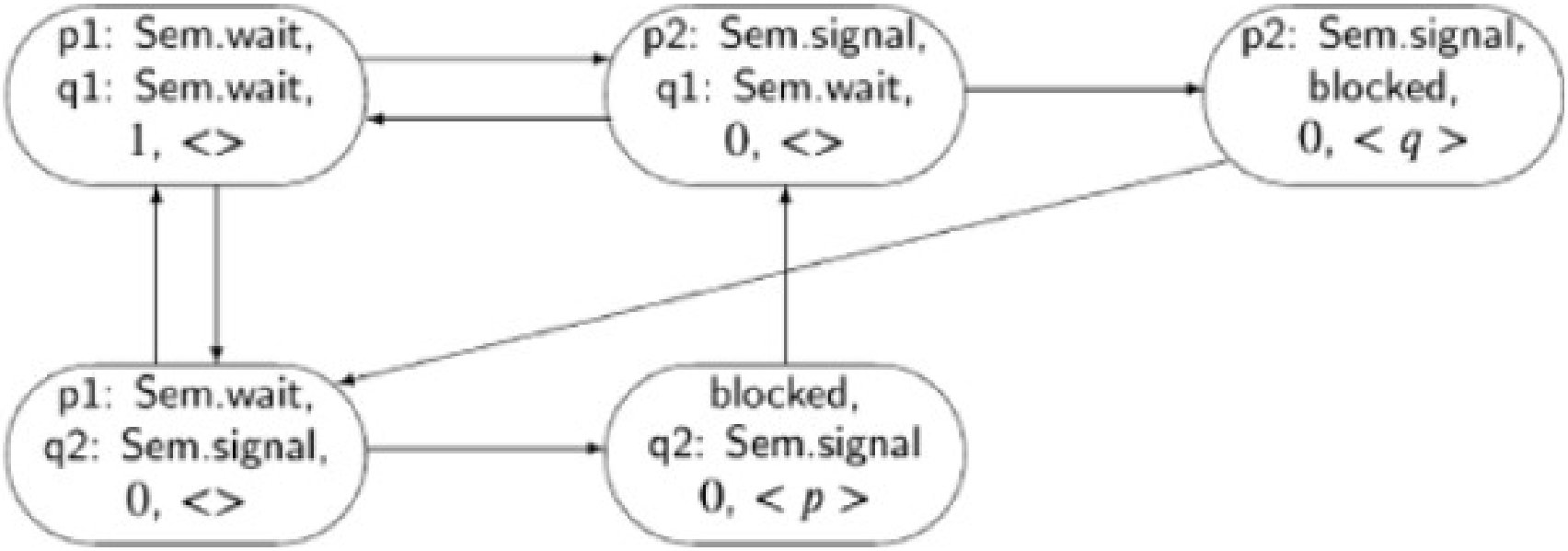
Variáveis de Condição (condition)

- Diferenças.

Semaphore	**Monitor**
`wait` may or may not block	`waitC` always blocks
`signal` always has an effect	`signalC` has no effect if queue is empty
`signal` unblocks an arbitrary blocked process	`signalC` unblocks the process at the head of the queue
a process unblocked by `signal` can resume execution immediately	a process unblocked by `signalC` must wait for the signaling process to leave monitor

Variáveis de Condição (condition)

- Diagrama



Produtor-Consumidor

Algorithm 7.3. producer–consumer (finite buffer, monitor)

```
monitor PC
  bufferType buffer ← empty
  condition notEmpty
  condition notFull

  operation append(datatype v)
    if buffer is full
       waitC(notFull)
    append(v, buffer)
    signalC(notEmpty)

  operation take()
    datatype w
    if buffer is empty
       waitC(notEmpty)
    w ← head(buffer)
    signalC(notFull)
    return w
```

producer	consumer
datatype d loop forever	datatype d loop forever
p1: d ← produce p2: PC.append(d)	q1: d ← PC.take q2: consume(d)

Leitor-Escritor

Algorithm 7.4. Readers and writers with a monitor

```
monitor RW
  integer readers ← 0
  integer writers ← 0
  condition OKtoRead, OKtoWrite

  operation StartRead
    if writers ≠ 0 or not empty(OKtoWrite)
       waitC(OKtoRead)
    readers ← readers + 1
    signalC(OKtoRead)

operation EndRead
  readers ← readers - 1
  if readers = 0
     signalC(OKtoWrite)

operation StartWrite
  if writers ≠ 0 or readers ≠ 0
     waitC(OKtoWrite)
  writers ← writers + 1

operation EndWrite
   writers ← writers - 1
   if empty(OKtoRead)
      then signalC(OKtoWrite)
      else signalC(OKtoRead)
```

reader	writer
p1: RW.StartRead p2: read the database p3: RW.EndRead	q1: RW.StartWrite q2: write to the database q3: RW.EndWrite

Filósofos

```
monitor ForkMonitor
  integer array[0..4] fork ← [2, . . . , 2]
  condition array[0..4] OKtoEat
  operation takeForks(integer i)
    if fork[i] ≠ 2
       waitC(OKtoEat[i])
    fork[i+1] ← fork[i+1] - 1
    fork[i-1] ← fork[i-1] - 1
  operation releaseForks(integer i)
    fork[i+1] ← fork[i+1] + 1
    fork[i-1] ← fork[i-1] + 1
    if fork[i+1] = 2
       signalC(OKtoEat[i+1])
    if fork[i-1] = 2
       signalC(OKtoEat[i-1])
```

philosopher i

```
    loop forever



p1:    think
p2:    takeForks(i)
p3:    eat
p4:    releaseForks(i)
```

Exercícios

- 1,2,3,4