Modelos de Computação Concorrente
O problema da Seção Crítica

Com Slides do livro de Ben-Ari

Fernando Dotti

Síntese

- Soluções de SW para seção crítica
  - 2 processos
  - N processos
  - Raciocínio sobre corretude
    - Modelo e verificação em CSP (algoritmo de Peterson)
    - Remete a bibliografia (capítulo 4 de Ben-Ari) para prova dedutiva de algoritmos concorrentes
- Suporte de HW
- Semáforos
    - Modelo em CSP para semáforos
- Monitores
    - Conceito
    - Uso
    - Implementação de monitores com semáforos

Bibliografia Base

[disponível na biblioteca]

M. Ben-Ari

Principles of Concurrent and Distributed Programming

Second Edition

Addison-Wesley, 2006

- Discussão de nível de atomicidade de operações
  E
- As possíveis intercalações de operações atômicas

- Definição do problema da Seção Crítica

Algorithm 2.3: Atomic assignment statements

integer n ← 0

p	**q**
p1: n ← n + 1	q1: n ← n + 1

Scenario for Atomic Assignment Statements

Process p	Process q	n
p1: n←n+1	q1: n←n+1	0
(end)	**q1: n←n+1**	1
(end)	(end)	2

Process p	Process q	n
p1: n←n+1	**q1: n←n+1**	0
p1: n←n+1	(end)	1
(end)	(end)	2

Algorithm 2.4: Assignment statements with one global reference

integer n ← 0

p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Correct Scenario for Assignment Statements

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	q1: temp←n	0	0	?
(end)	**q1: temp←n**	1	0	?
(end)	**q2: n←temp+1**	1	0	1
(end)	(end)	2	0	1

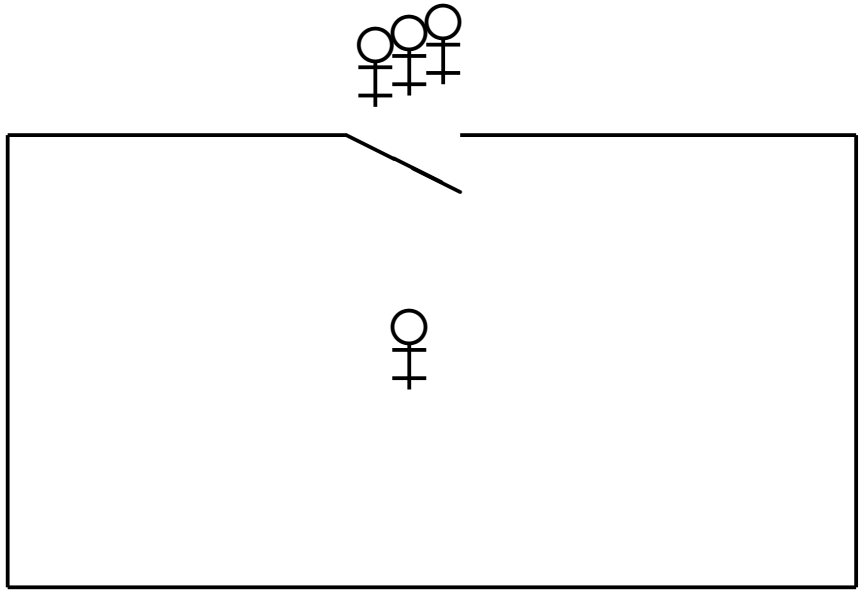
Incorrect Scenario for Assignment Statements

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	**q1: temp←n**	0	0	?
p2: n←temp+1	q2: n←temp+1	0	0	0
(end)	**q2: n←temp+1**	1	0	0
(end)	(end)	1	0	0

- Seção Crítica (SC)
  - sistema com N processos, N > 1
  - cada processo pode ter um código próprio
  - os processos compartilham dados variáveis, de qualquer tipo
  - cada processo possui SC's de código, onde atualizam os dados compartilhados
  - a execução de 1 SC deve ser de forma mutuamente exclusiva no tempo

- Seção Crítica
  - prover exclusão mútua
  - Progresso
    - não bloqueio
    - processos fora da SC não devem bloquear outros processos
    - somente os processos querendo entrar na SC devem participar da seleção do próximo a entrar
  - espera limitada (não postergação)
    - um processo espera um tempo limitado na *entry-section*
  - velocidades indeterminadas
    - não se faz suposições sobre a velocidade relativa dos processos

Critical Section

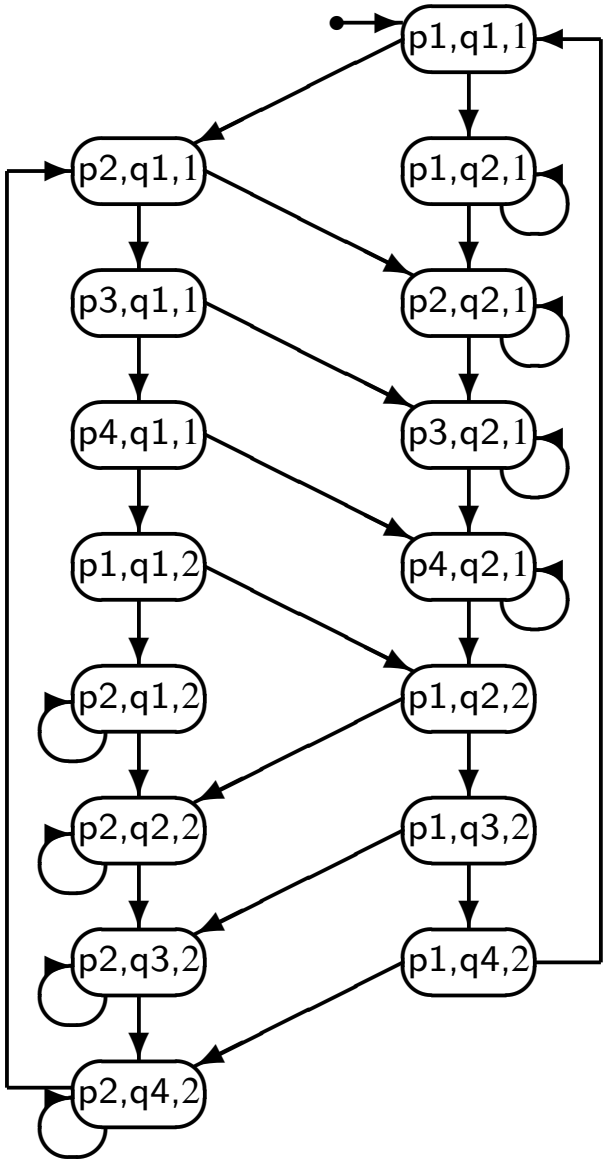


Algorithm 3.1: Critical section problem	
global variables	
p	**q**

local variables
loop forever
  non-critical section
  preprotocol
  critical section
  postprotocol
	local variables
	loop forever
	  non-critical section
	  preprotocol
	  critical section
	  postprotocol

Algorithm 3.2: First attempt	
integer turn $\leftarrow 1$	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await turn $= 1$	q2: await turn $= 2$
p3: critical section	q3: critical section
p4: turn $\leftarrow 2$	q4: turn $\leftarrow 1$

State Diagram for the First Attempt



– Progresso ?

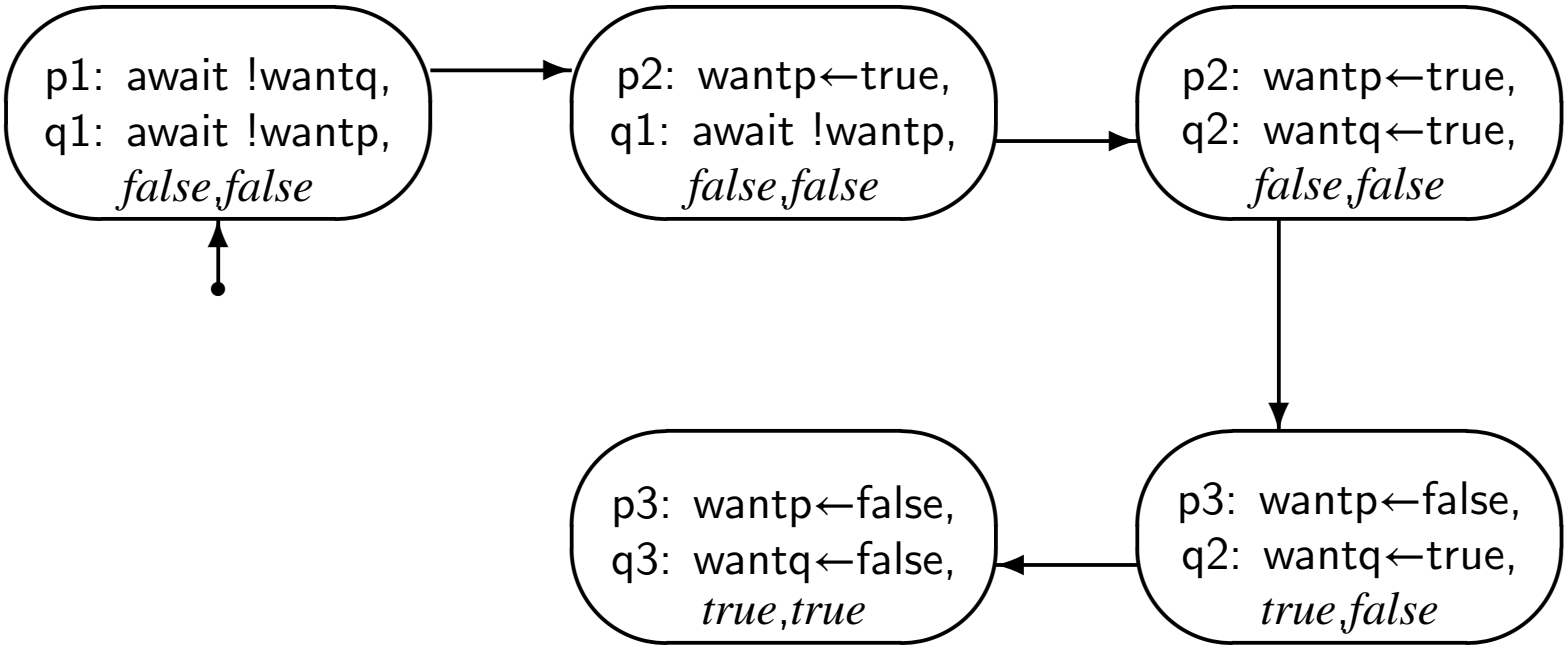
- processos fora da SC não devem bloquear outros processos
- Um processo tem que usar a seção crítica para passar a vez para outro

Algorithm 3.6: Second attempt

boolean wantp ← false, wantq ← false

p		q	
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	await wantq = false	q2:	await wantp = false
p3:	wantp ← true	q3:	wantq ← true
p4:	critical section	q4:	critical section
p5:	wantp ← false	q5:	wantq ← false

Fragment of the State Diagram for the Second Attempt



Scenario Showing that Mutual Exclusion Does Not Hold

Process p	Process q	wantp	wantq
p1: await wantq=false	q1: await wantp=false	*false*	*false*
p2: wantp←true	**q1: await wantp=false**	*false*	*false*
p2: wantp←true	q2: wantq←true	*false*	*false*
p3: wantp←false	**q3: wantq←true**	*true*	*false*
p3: wantp←false	q3: wantq←false	*true*	*true*

Algorithm 3.8: Third attempt	
boolean wantp ← false, wantq ← false	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: await wantq = false	q3: await wantp = false
p4: critical section	q4: critical section
p5: wantp ← false	q5: wantq ← false

Scenario Showing Deadlock in the Third Attempt

Process p	Process q	wantp	wantq
p1: non-critical section	q1: non-critical section	*false*	*false*
p2: wantp←true	**q1: non-critical section**	*false*	*false*
p2: wantp←true	q2: wantq←true	*false*	*false*
p3: await wantq=false	**q2: wantq←true**	*true*	*false*
p3: await wantq=false	q3: await wantp=false	*true*	*true*

Algorithm 3.13: Peterson's algorithm

boolean wantp ← false, wantq ← false
integer last ← 1

	p		**q**
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	wantp ← true	q2:	wantq ← true
p3:	last ← 1	q3:	last ← 2
p4:	await wantq = false or last = 2	q4:	await wantp = false or last = 1
p5:	critical section	q5:	critical section
p6:	wantp ← false	q6:	wantq ← false

Algorithm 3.10: Dekker's algorithm

```
boolean wantp ← false, wantq ← false
integer turn ← 1
```

p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: while wantq	q3: while wantp
p4: if turn = 2	q4: if turn = 1
p5: wantp ← false	q5: wantq ← false
p6: await turn = 1	q6: await turn = 2
p7: wantp ← true	q7: wantq ← true
p8: critical section	q8: critical section
p9: turn ← 2	q9: turn ← 1
p10: wantp ← false	q10: wantq ← false

- Como demonstrar que algoritmo funciona ?
  - Usando diagramas de estado
    - Brevemente exemplificado acima
  - Com modelo e avaliando propriedades
    - Em CSP por exemplo
  - Com prova dedutiva do seu funcionamento correto
    - Ler capítulo 4 de Ben-Ari, até seção 4.5

- Desenvolver modelo CSP do algoritmo de Peterson e avaliar:
  - Não bloqueio
  - Exclusão mútua
  - Não postergação

```
-- Peterson's Algorithm in CSP: version 2
-- Simon Gay, Royal Holloway, January 1999

channel flag1set, flag1read, flag2set, flag2read:{1..2}.{false,true}
channel turnset, turnread:{1..2}.{1..2}
channel enter, critical, leave:{1..2}

FLAG1(v) =  flag1set?x?y -> FLAG1(y)
       [] flag1read.1.v -> FLAG1(v)
       [] flag1read.2.v -> FLAG1(v)

FLAG2(v) =  flag2set?x?y -> FLAG2(y)
       [] flag2read.1.v -> FLAG2(v)
       [] flag2read.2.v -> FLAG2(v)

TURN(v) =  turnset?x?y -> TURN(y)
      [] turnread.1.v -> TURN(v)
      [] turnread.2.v -> TURN(v)

P1 = flag1set.1.true -> turnset.1.2 -> P1WAIT
P1WAIT =  flag2read.1.true -> (turnread.1.2 -> P1WAIT
                    [] turnread.1.1 -> P1ENTER)
     [] flag2read.1.false -> P1ENTER
P1ENTER = enter.1 -> critical.1 -> leave.1 -> flag1set.1.false -> P1

P2 = flag2set.2.true -> turnset.2.1 -> P2WAIT
P2WAIT =  flag1read.2.true -> (turnread.2.1 -> P2WAIT
                    [] turnread.2.2 -> P2ENTER)
     [] flag1read.2.false -> P2ENTER
P2ENTER = enter.2 -> critical.2 -> leave.2 -> flag2set.2.false -> P2

aP1 = {| flag1set.1, flag1read.1, flag2set.1, flag2read.1,
       turnset.1, turnread.1, enter.1, critical.1, leave.1 |}

aP2 = {| flag1set.2, flag1read.2, flag2set.2, flag2read.2,
       turnset.2, turnread.2, enter.2, critical.2, leave.2 |}

aF1 = {| flag1set,flag1read |}

aF2 = {| flag2set,flag2read |}

aT  = {| turnset,turnread |}

PROCS = P1 [ aP1 || aP2 ] P2

FLAGS = FLAG1(false) [ aF1 || aF2 ] FLAG2(false)

VARS = FLAGS [ union(aF1,aF2) || aT ] TURN(1)

SYSTEM = PROCS [ union(aP1,aP2) || union(union(aF1,aF2),aT) ] VARS

assert SYSTEM :[deadlock free]
```

- Soluções de software para N processos

Algorithm 5.2: Bakery algorithm (N processes)

integer array[1..n] number $\leftarrow$ [0,. . . ,0]

```
      loop forever
p1:       non-critical section
p2:       number[i] ← 1 + max(number)
p3:       for all other processes j
p4:           await (number[j] = 0) or (number[i] ≪ number[j])
p5:       critical section
p6:       number[i] ← 0
```

Algorithm 5.3: Bakery algorithm without atomic assignment

```
                boolean array[1..n] choosing ← [false,...,false]
                integer array[1..n] number ← [0,...,0]

    loop forever
p1:     non-critical section
p2:     choosing[i] ← true
p3:     number[i] ← 1 + max(number)
p4:     choosing[i] ← false
p5:     for all other processes j
p6:         await choosing[j] = false
p7:         await (number[j] = 0) or (number[i] ≪ number[j])
p8:     critical section
p9:     number[i] ← 0
```

Algorithm 5.8: Lamport's one-bit algorithm

boolean array[1..n] want ← [false,. . . ,false]

```
      loop forever
          non-critical section
p1:       want[i] ← true
p2:       for all processes j < i
p3:           if want[j]
p4:               want[i] ← false
p5:               await not want[j]
                  goto p1
p6:       for all processes j > i
p7:           await not want[j]
          critical section
p8:       want[i] ← false
```

- Suporte de operações atômicas
  - Test and set (local, compartilhada)

    local <- compartilhada

    Compartilhada <- 1
  - Exchange ou swap(a,b)

Algorithm 3.11: Critical section problem with test-and-set

integer common ← 0

p	**q**
integer local1	integer local2
loop forever	loop forever
p1: non-critical section	q1: non-critical section
repeat	repeat
p2: test-and-set(common, local1)	q2: test-and-set(common, local2)
p3: until local1 = 0	q3: until local2 = 0
p4: critical section	q4: critical section
p5: common ← 0	q5: common ← 0

Algorithm 3.12: Critical section problem with exchange

integer common ← 1

	p		**q**
	integer local1 ← 0		integer local2 ← 0
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
	repeat		repeat
p2:	exchange(common, local1)	q2:	exchange(common, local2)
p3:	until local1 = 1	q3:	until local2 = 1
p4:	critical section	q4:	critical section
p5:	exchange(common, local1)	q5:	exchange(common, local2)

- Semáforos
  - Construção de sincronização com
    - V: valor inteiro não negativo
    - L: lista de processos
    - Operações atômicas Wait [ou P] e signal [ou V], onde

```
S(val,list) { v:=val; l:=list}   /* construtor */
wait(s): if (s.v > 0 )
        then s.c := s.v -1
        else {
                s.l := s.l U p  /* adicionar o processo na lista s.l */
                block(p) ;      /*bloqueia o processo p no SO*/
         }
signal(s): if (s.l = {} )
          then s.v := s.v+1
            else {
           tome um elemento q de l
           s.l := s.l – q  /*remover o processo "q" da lista s.l */
           wakeup(q)  /* acorda o processo p no SO */
            }
```

Algorithm 6.1: Critical section with semaphores (two processes)

binary semaphore S $\leftarrow (1, \emptyset)$

p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wait(S)	q2: wait(S)
p3: critical section	q3: critical section
p4: signal(S)	q4: signal(S)

Algorithm 6.4: Critical section with semaphores (N proc., abbrev.)

binary semaphore S $\leftarrow (1, \emptyset)$

```
      loop forever
p1:     wait(S)
p2:     signal(S)
```

Scenario for Starvation

n	Process p	Process q	Process r	S
1	**p1: wait(S)**	q1: wait(S)	r1: wait(S)	$(1, \emptyset)$
2	p2: signal(S)	**q1: wait(S)**	r1: wait(S)	$(0, \emptyset)$
3	p2: signal(S)	q1: blocked	**r1: wait(S)**	$(0, \{q\})$
4	**p1: signal(S)**	q1: blocked	r1: blocked	$(0, \{q, r\})$
5	**p1: wait(S)**	q1: blocked	r2: signal(S)	$(0, \{q\})$
6	p1: blocked	q1: blocked	**r2: signal(S)**	$(0, \{p, q\})$
7	p2: signal(S)	q1: blocked	**r1: wait(S)**	$(0, \{q\})$

- Semáforos fortes
  - Construção de sincronização com
    - V: valor inteiro não negativo
    - L: lista de processos administrada como fila
    - Operações atômicas Wait [ou P] e signal [ou V], onde

```
S(val,list) { v:=val; l:=list}   /* construtor */
wait(s): if (s.v > 0 )
            then s.c := s.v -1
            else {
                    s.l := append(s.l,p)  /* adicionar o processo no final da fila s.l */
                    block(p) ;     /*bloqueia o processo p no SO*/
              }
 signal(s): if (s.l = {} )
               then s.v := s.v+1
               else {
                  q := head(s.l)  /* tome o primeiro elemento */
                  s.l := tail(s.l)  /*remover o processo "q" da lista s.l */
                  wakeup(q)  /* acorda o processo p no SO */
               }
```

- Defina modelos CSP para Semáforos
- Tente construir um modelo para semáforos fortes

```
-- Exemplos baseados no livro de Maggee para testar
-- modelos baseados em semáforos e produtor-consumidor


Max_Sema = 3
channel ups, downs

SIMPLE_SEMA(s) =  (if s>0
                then downs->SIMPLE_SEMA(s-1)
                else STOP) [] ups -> SIMPLE_SEMA(s+1)

BOUNDED_SIMPLE_SEMA(s) =  if (s>0 and s<Max_Sema)
              then
               (ups->BOUNDED_SIMPLE_SEMA(s+1)
                []downs->BOUNDED_SIMPLE_SEMA(s-1))
               else if s==0
                then ups->BOUNDED_SIMPLE_SEMA(s+1)
                else  downs->BOUNDED_SIMPLE_SEMA(s-1)

INIT_SEMA= SIMPLE_SEMA(3)




Number_Processes = 3
nametype Processes = {0..(Number_Processes - 1)}
channel up,down : Processes
GEN_SEMA(s) = up?j:Processes->GEN_SEMA(s+1)
              [] (if s>0 then down?j:Processes->GEN_SEMA(s-1)
                    else STOP)

INIT_GEN_SEMA = GEN_SEMA(3)




channel critical:Processes

PROC(j)=down.j->critical.j->up.j->PROC(j)

PAR_PROC = ||| i: Processes @ PROC(i)

PROC_INT ={|down,up,critical|}
SEMA_INT ={|up,down|}

SEMA_MUTEX_DEMO = PAR_PROC [PROC_INT||SEMA_INT] GEN_SEMA(1)
```

- Modelo Produtor/Consumidor

Algorithm 6.6: Producer-consumer (infinite buffer)

infinite queue of dataType buffer ← empty queue
semaphore notEmpty ← $(0, \emptyset)$

	producer		**consumer**
	dataType d		dataType d
	loop forever		loop forever
p1:	d ← produce	q1:	wait(notEmpty)
p2:	append(d, buffer)	q2:	d ← take(buffer)
p3:	signal(notEmpty)	q3:	consume(d)

Algorithm 6.8: Producer-consumer (finite buffer, semaphores)

```
finite queue of dataType buffer ← empty queue
semaphore notEmpty ← (0, ∅)
semaphore notFull ← (N, ∅)
```

producer	**consumer**
dataType d	dataType d
loop forever	loop forever
p1: d ← produce	q1: wait(notEmpty)
p2: wait(notFull)	q2: d ← take(buffer)
p3: append(d, buffer)	q3: signal(notFull)
p4: signal(notEmpty)	q4: consume(d)

- Jantar dos filósofos
  - Garfos são semáforos, filósofos são processos

Algorithm 6.10: Dining philosophers (first attempt)

semaphore array [0..4] fork ← [1,1,1,1,1]

```
      loop forever
p1:       think
p2:       wait(fork[i])
p3:       wait(fork[i+1])
p4:       eat
p5:       signal(fork[i])
p6:       signal(fork[i+1])
```

- Exclusao mutua – ok
- Starvation – ok
- Deadlock - ☹
  - Permitir 1 filósofo a menos entrando na sala de jantar

Algorithm 6.11: Dining philosophers (second attempt)

```
                    semaphore array [0..4] fork ← [1,1,1,1,1]
                    semaphore room ← 4
     loop forever
p1:    think
p2:    wait(room)
p3:    wait(fork[i])
p4:    wait(fork[i+1])
p5:    eat
p6:    signal(fork[i])
p7:    signal(fork[i+1])
p8:    signal(room)
```

- Readers and writers
  - Enquanto processos readers estiverem lendo, outros readers podem ler
  - Writers tem que escrever quando nao houverem processos lendo ou escrevendo

Algorithm 6.21: Readers and writers with semaphores

```
                semaphore readerSem ← 0, writerSem ← 0
                integer delayedReaders ← 0, delayedWriters ← 0
                semaphore entry ← 1
                integer readers ← 0, writers ← 0
```

SignalProcess

```
if writers = 0 or delayedReaders > 0
    delayedReaders ← delayedReaders − 1
    signal(readerSem)
else if readers = 0 and writers = 0 and delayedWriters > 0
    delayedWriters ← delayedWriters − 1
    signal(writerSem)
else signal(entry)
```

Algorithm 6.21: Readers and writers with semaphores

StartWrite

```
p11: wait(entry)
p12: if writers > 0 or readers > 0
p13:     delayedWriters ← delayedWriters + 1
p14:     signal(entry)
p15:     wait(writerSem)
p16: writers ← writers + 1
p17: SignalProcess
```

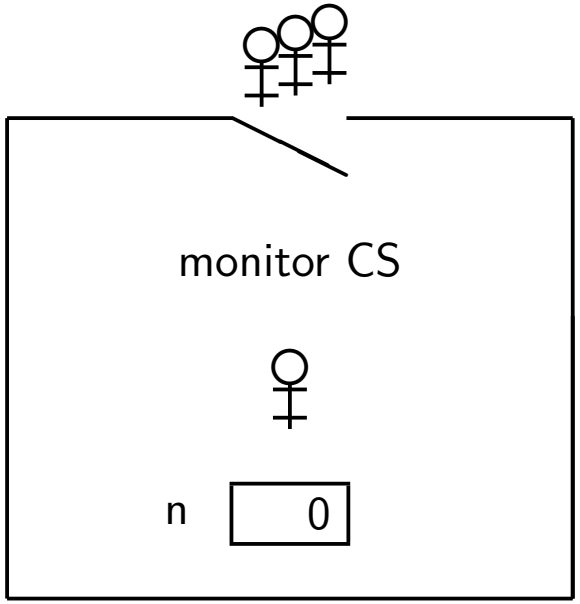
EndWrite

```
p18: wait(entry)
p19: writers ← writers − 1
p20: SignalProcess
```

- Semáforos
  - Construções de baixo nível de abstração
  - Depende da construção e uso correto nos processos que usam a estrutura compartilhada

- Monitores
  - Provêem estrutura que concentra responsabilidade pelo acesso concorrente correto junto à estrutura representada
  - Encapsulamento

Executing a Monitor Operation



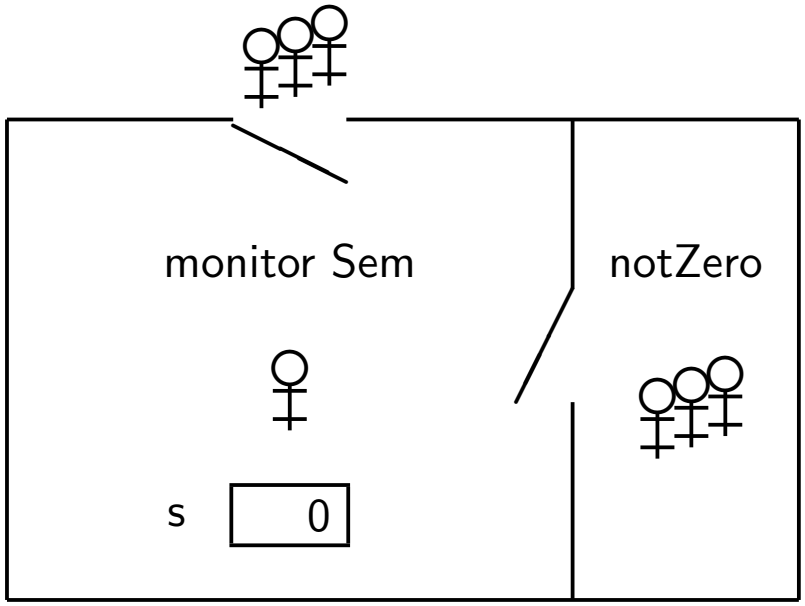
Algorithm 7.1: Atomicity of monitor operations

```
monitor CS
    integer n ← 0

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

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

Condition Variable in a Monitor



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

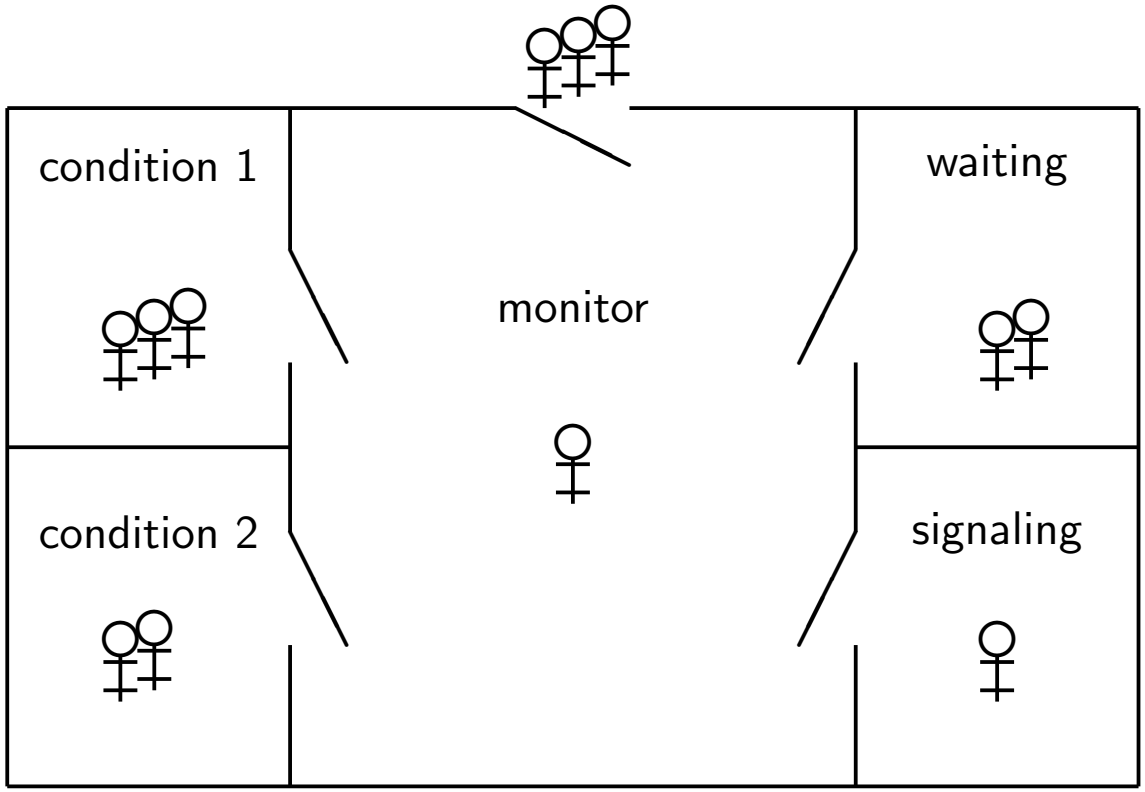
Algorithm 7.3: Producer-consumer (finite buffer, monitor)

```
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
```

Algorithm 7.3: Producer-consumer (finite buffer, monitor) (continued)

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

The Immediate Resumption Requirement



Monitor

(Brinch Hansen, Hoare) 73, 74

- É um modelo que permite o compartilhamento de dados
- possui valores que representam o estado do objeto e as procedures que manipulam os valores
- as procedures são executadas de forma mutuamente exclusiva
- variáveis especiais (condição) permitem a um processo se bloquear a espera de uma condição (wait)
- a condição é sinalizada por um outro processo (operação signal)

Monitor

(Brinch Hansen, Hoare) 73, 74

- Variáveis Condição:
  - ex.: var x, y : condition ;
  - x.wait:
    - o processo que executa essa operação é suspenso até que um outro processo execute a operação x.signal
  - x.signal:
    - acorda um único processo
    - se não existem processos bloqueados, a operação não produz efeitos

Monitor

(Brinch Hansen, Hoare) 73, 74

- Implementação de Monitor com Semáforo:
  - cada monitor é representado por um semáforo **mutex** inicializado com 1.
    wait (mutex): entrar no monitor
    signal (mutex): liberar monitor
  - semáforo **next** inicializado com 0
    para um processo sinalizador se bloquear
  - next count: contém o No. de processos bloqueados em next

```
wait ( mutex ) ;
   " corpo da procedure entry"
if next-count > 0 then
     signal ( next )
  else
     signal ( mutex ) ;
```

Monitor

(Brinch Hansen, Hoare) 73, 74

– Para cada variável condition x, associar um semáforo **x-sem** e uma variável inteira **x-count**

```
X.wait :  x-count = x.count + 1      /* vou me bloquear */
          if  next-count > 0 then    /* se algum proc estava no monitor e foi bloq, libera */
              signal ( next )
          else
              signal ( mutex )       /* senão libera proc querendo entrar no monitor */
            wait ( x-sem )           /* bloqueia */
            x-count = x-count-1      /* depois de desbloquear, diminui nro de bloqueados */
```

```
X.signal : if x-count > 0 them       /* se há processo bloqueado */
           begin
             next-count = next-count + 1
             signal ( x-sem )        /* desbloqueia processo */
             wait ( next )           /* se bloqueia em next com preferencia sobre procs fora */
             next-count = next-count – 1
           end
```