Programação Concorrente

Semáforos

Introdução

- Soluções anteriores para a Seção Crítica usam apenas a linguagem pura de máquina, as quais são de baixo nível.
- Semáforos são construções de alto nível, geralmente implementados pelo sistema operacional.
- Simples e amplamente usados.

Estados do Processo

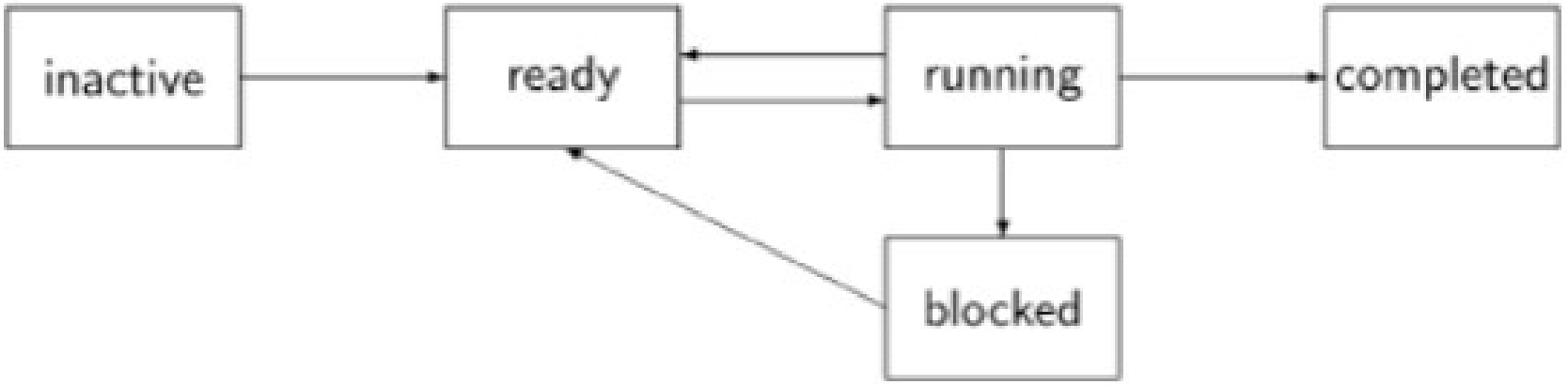
- Em um sistema de multiprocessador, há mais processadores que processos em um programa em execução. Ou seja, um processo está sempre executando em um processador.
- Um sistema multitarefa, vários processos devem compartilhar um único processador, gerando espera.
  - **Ready process**: processo que deseja executar.
  - **Running process**: processo em execução.

Estados do Processo

- Um programa chamado “scheduler” é responsável em decidir qual processo “ready” irá executar.
- Context switch: o scheduler troca um processo “running” por um “ready” e muda o estado dos processos processos para “ready” e “running”, respectivamente.
- Todo processo p, está associado a um estado (p.state). Por exemplo, se um processo p.state = running e q.state = ready, numa troca de contexto:
  - p.state ← ready
  - q.state ← running

Estados do Processo

- Possíveis estados de um processo:



Semáforo - Definição

- É um tipo de dados composto, formado por dois campos:
  - S.V : número não negativo;
  - S.L : conjunto de processos;
- Um semáforo S deve ser inicializado com um valor k>=0 para S.V e um conjunto vazio (Ø) para S.L.

semaphore S ← (k, Ø)

Semáforo - Operações

- Existem duas operações atômicas definidas em um semáforo S, onde p é um processo.
- Wait:

```
wait(S)

if S.V > 0
   S.V ← S.V - 1
else
   S.L ← S.L ∪ p
   p.state ← blocked
```

If the value of the integer component is nonzero, decrement its value (and the process p can continue its execution); otherwise—it is zero—process p is added to the set component and the state of p becomes blocked. We say that p is blocked on the semaphore S.

Semáforo - Operações

- Signal

signal(S)

```
if S.L = ∅
   S.V ← S.V + 1
else
   let q be an arbitrary element of S.L
   S.L ← S.L - {q}
   q.state ← ready
```

If S.L is empty, increment the value of the integer component; otherwise—S.L is nonempty—unblock q, an arbitrary element of the set of processes blocked on S.L. The state of process p does not change.

Semáforo - Operações

- Um semáforo que permite qualquer valor positivo para V, é chamado de semáforo geral. Já um semáforo que permite apenas 0 ou 1, é chamado de semáforo **binário**. A operação de signal, muda:

```
if S.V = 1
   // undefined
else if S.L = ∅
   S.V ← 1
else // (as above)
   let q be an arbitrary element of S.L
   S.L ← S.L - {q}
   q.state ← ready
```

A binary semaphore is sometimes called a mutex. This term is used in pthreads and in java.util.concurrent.

SC para dois processos

- A solução torna-se trivial.

Algorithm 6.1. Critical section with semaphores (two processes)

```
binary semaphore S ← (1, ∅)
```

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

SC para dois processos

- Versão abreviada.

Algorithm 6.2. Critical section with semaphores (two proc., abbrev.)

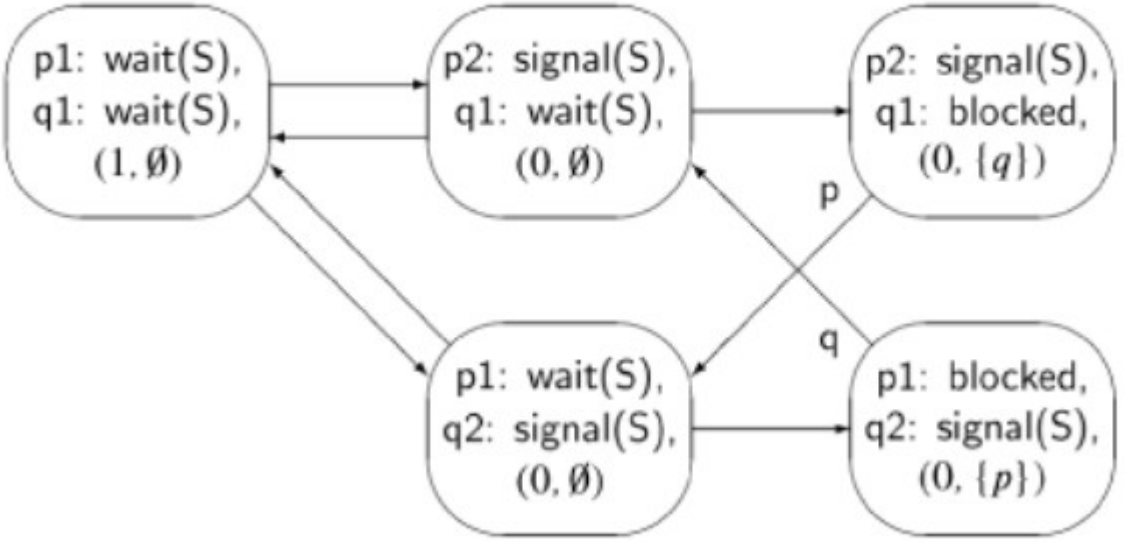
```
binary semaphore S ← (1, ∅)
```

p	q
loop forever p1: wait(S) p2: signal(S)	loop forever q1: wait(S) q2: signal(S)

SC para dois processos

- Diagrama.

Figure 6.1. State diagram for the semaphore solution



SC para dois processos

- A violation of the mutual exclusion requirement would be a state of the form (p2: signal(S), q2: signal(S), . . .). We immediately see that no such state exists, so we conclude that the solution satisfies this requirement.
- Similarly, there is no deadlock, since there are no states in which both processes are blocked.
- Finally, the algorithm is free from starvation, since if a process executes its wait statement (thus leaving the non-critical section), it enters either the state with the signal statement (and the critical section) or it enters a state in which it is blocked. But the only way out of a blocked state is into a state in which the blocked process continues with its signal statement.

SC para N processos

Algorithm 6.3. Critical section with semaphores (*N* proc.)

```
binary semaphore S ← (1, ∅)

    loop forever
p1:   non-critical section
p2:   wait(S)
p3:   critical section
p4:   signal(S)
```

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

```
binary semaphore S ← (1, ∅)

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

SC para N processos

- Starvation para 3 processos

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

Line 7 is the same as line 3 and the scenario can continue indefinitely in this loop of states. The two processes p and r "conspire" to starve process q.

Ordem de execução

- Problemas de sincronização ocorrem quando os processos devem coordenar a ordem de execução de operações de diferentes processos.
- Considere o problema do mergesort para o vetor [5, 1, 10, 7, 4, 3, 12, 8].
- Ele é então dividido em [5, 1, 10, 7] e [4, 3, 12, 8].
- Ordenando, temos [1, 5, 7, 10] e [3, 4, 8, 12].
- Depois, fazemos um merge e obtemos [1, 3, 4, 5, 7, 8, 10, 12].

Ordem de execução

- 3 processos: 1 dois para o sorting e um para o merge.

Algorithm 6.5. Mergesort

```
integer array A
binary semaphore S1 ← (0, ∅)
binary semaphore S2 ← (0, ∅)
```

sort1	sort2	merge
p1: sort 1st half of A p2: signal(S1) p3:	q1: sort 2nd half of A q2: signal(S2) q3:	r1: wait(S1) r2: wait(S2) r3: merge halves of A

Produtor Consumidor

- Exemplo clássico de ordem de execução. Existem dois tipos de processos:
  - *Producers: A producer process executes a statement **produce** to create a data element and then sends this element to the consumer processes.*
  - *Consumers: Upon receipt of a data element from the producer processes, a consumer process executes a statement **consume** with the data element as a parameter.*

Produto Consumidor

- Exemplos

Producer	**Consumer**
Communications line	Web browser
Web browser	Communications line
Keyboard	Operating system
Word processor	Printer
Joystick	Game program
Game program	Display screen
Pilot controls	Control program
Control program	Aircraft control surfaces

Produto Consumidor

- Se a comunição for síncrona, o consumidor e produtor só trocam dados caso estejam prontos.
- No entanto, a comunicação assíncrona é mais comum. Os processos guardam os dados em um **buffer**. O produtor executa a operação **append** para colocar o dado na cauda da fila. Já o consumidor efetua a operação **take**, para tirar um elemento da cabeça da fila.

Produto Consumidor

- Buffer infinito

Algorithm 6.6. producer–consumer (infinite buffer)

```
infinite queue of dataType buffer ← empty queue
semaphore notEmpty ← (0, ∅)
```

producer	consumer

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

If there is an infinite buffer, there is only one interaction that must be synchronized: the consumer must not attempt a take operation from an empty buffer. This is an order-of-execution problem like the mergesort algorithm; the difference is that it happens repeatedly within a loop.

Produto Consumidor

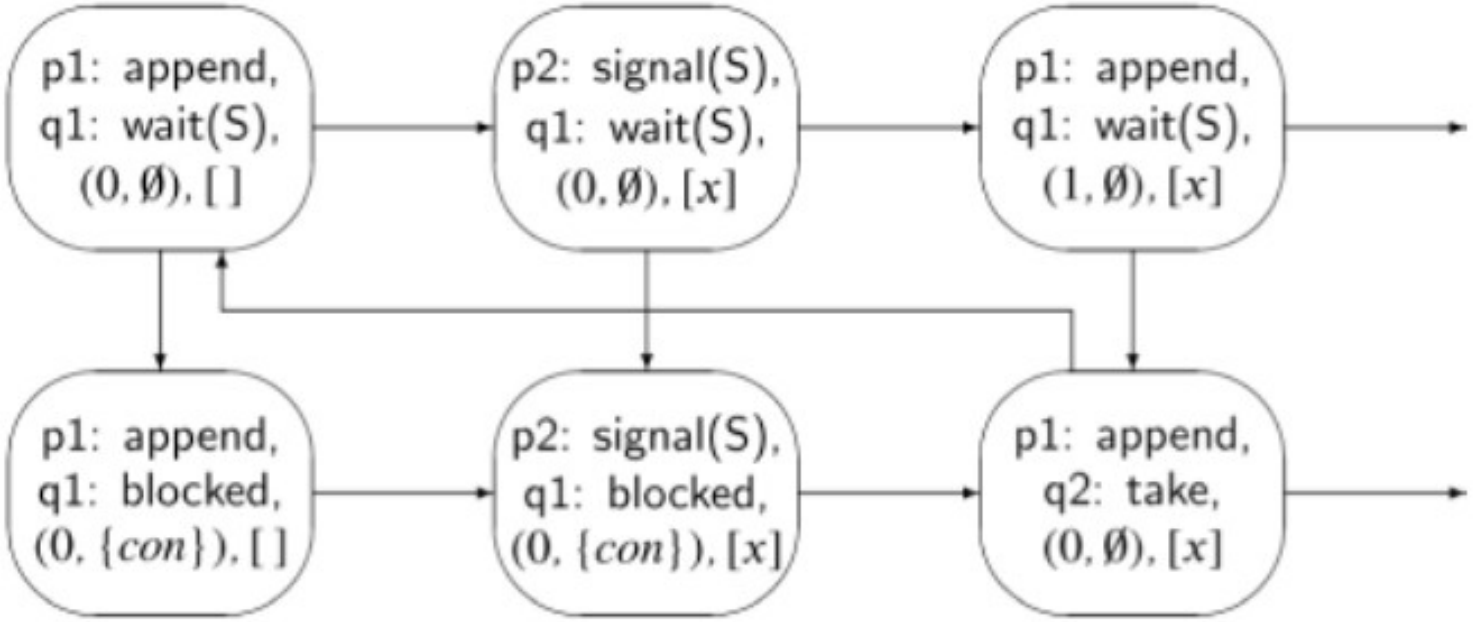
Algorithm 6.7. producer–consumer (infinite buffer, abbreviated)

```
infinite queue of dataType buffer ← empty queue
semaphore notEmpty ← (0, ∅)
```

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

Produto Consumidor

Figure 6.2. Partial state diagram for producer–consumer with infinite buffer



Produto Consumidor

- Buffer finito.

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

Semáforos

- Outras definições.
- Semáforos Fortes: S.L é uma fila.

wait(S)

```
if S.V > 0
   S.V ← S.V - 1
else
   S.L ← append(S.L, p)
   p.state ← blocked
```

signal(S)

```
if S.L = ∅
   S.V ← S.V + 1
else
   q ← head(S.L)
   S.L ← tail (S.L)
   q.state ← ready
```

Semáforos

- Busy-wait: não tem o componente S.L

wait(S)

```
await S > 0
S ⟵ S - 1
```

signal(S)

```
S ⟵ S + 1
```

n	Process p	Process q	S
1	**p1: wait(S)**	q1: wait(S)	1
2	p2: signal(S)	**q1: wait(S)**	0
3	**p2: signal(S)**	q1: wait(S)	0
4	**p1: wait(S)**	q1: wait(S)	1

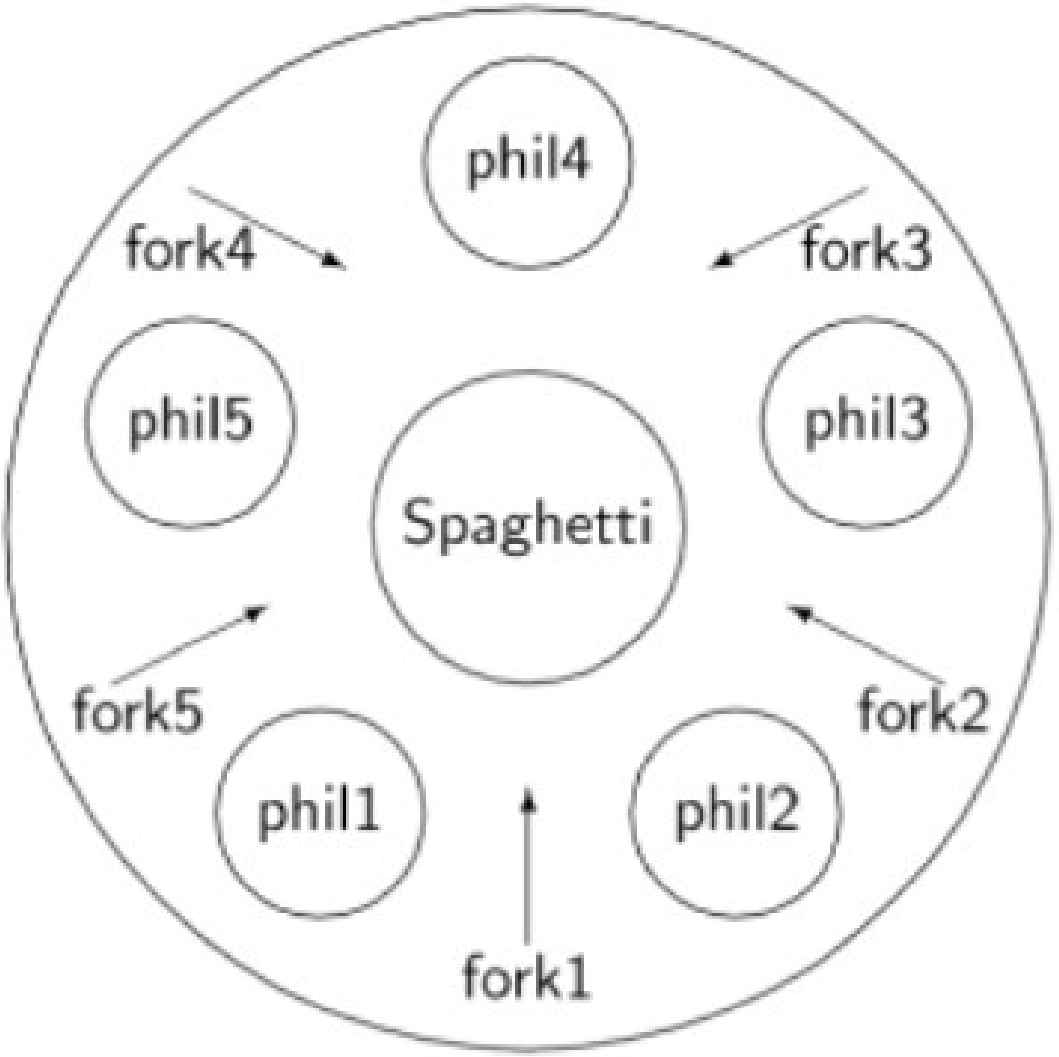
Os Filósofos

- The problem is set in a secluded community of five philosophers who engage in only two activities—**thinking** and **eating**:

Algorithm 6.9. Dining philosophers (outline)

```
    loop forever
p1:   think
p2:   preprotocol
p3:   eat
p4:   postprotocol
```

Os Filósofos



Os Filósofos

- The correctness properties are:
  - A philosopher eats only if she has two forks.
  - Mutual exclusion: no two philosophers may hold the same fork simultaneously.
  - Freedom from deadlock.
  - Freedom from starvation (pun!).

Os Filósofos

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])
```

Os Filósofos

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)
```

Os Filósofos

Algorithm 6.12. Dining philosophers (third attempt)

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

philosopher 4

```
    loop forever
p1:   think
p2:   wait(fork[0])
p3:   wait(fork[4])
p4:   eat
p5:   signal(fork[0])
p6:   signal(fork[4])
```

Exercícios

- 1,2,3,4,5,6, 9 e 10.