

Sistemas Operativos

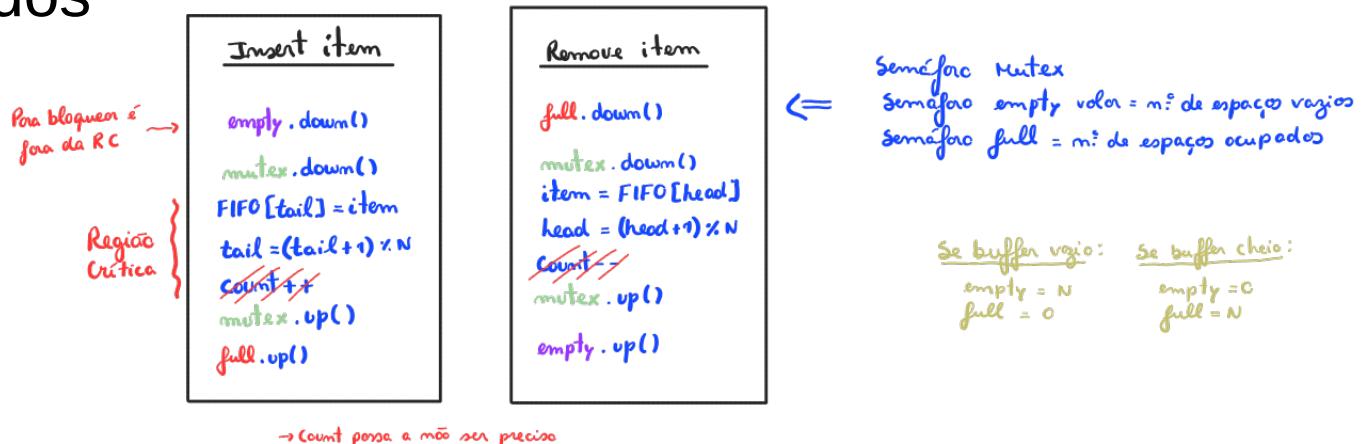
Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

Ano letivo 2023/2024

Nuno Lau (nunolau@ua.pt)

Bounded Buffer

- Para implementar um *Bounded Buffer* com capacidade N , podem ser usados 3 semáforos:
 - mutex**: para garantir a exclusão mútua no acesso à região crítica
 - empty**: cujo valor interno indica o número de espaços vazios
 - full**: cujo valor interno indica o número de espaços ocupados



Bounded Buffer

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private Object[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer() {
        // buffer is initially empty
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];

        mutex = new Semaphore(1);    ← 3 semáforos
        empty = new Semaphore(BUFFER_SIZE);
        full = new Semaphore(0);
    }
}
```

3 semáforos

inicialização

```
public void insert(Object item) {
    empty.acquire();
    mutex.acquire();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    mutex.release();
    full.release();
}

public Object remove() {
    full.acquire();
    mutex.acquire();

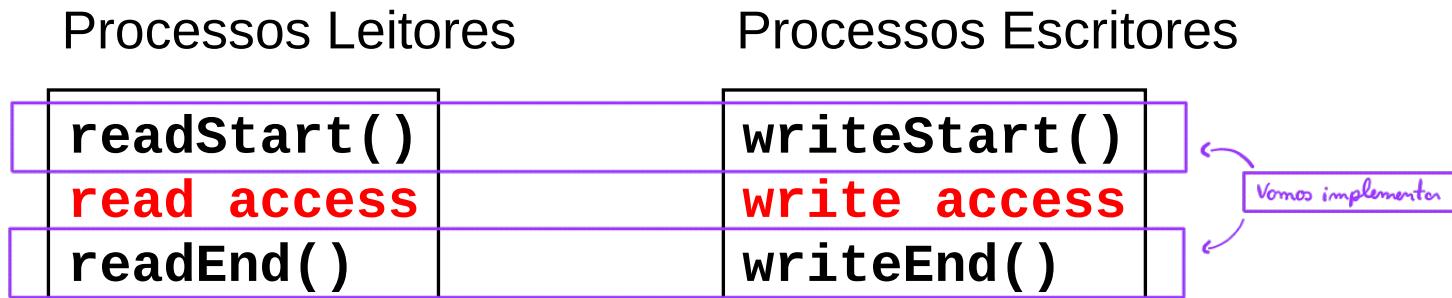
    // remove an item from the buffer
    Object item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.release();
    empty.release();

    return item;
}
```

Escritores e Leitores

- Mecanismo de sincronização através do qual:
 - Existem dados partilhados
 - Podem ocorrer várias leituras em simultâneo, desde que não estejam a ocorrer escritas
 - Durante a escrita não podem existir leituras, nem outras escritas concorrentes *→ vários leitores ou um escritor*



Escritores e Leitores

- Usando 2 semáforos (**mutex** e **nobody**) e um inteiro (**readers**)
(Número de leitores)
- mutex** e **nobody** inicializados a 1
- readers** inicializado a 0

bimáximos

(Mutual exclusion)

(alguém está a aceder aos dados?)

0 → não alguém a aceder
1 → Não está ninguém a aceder

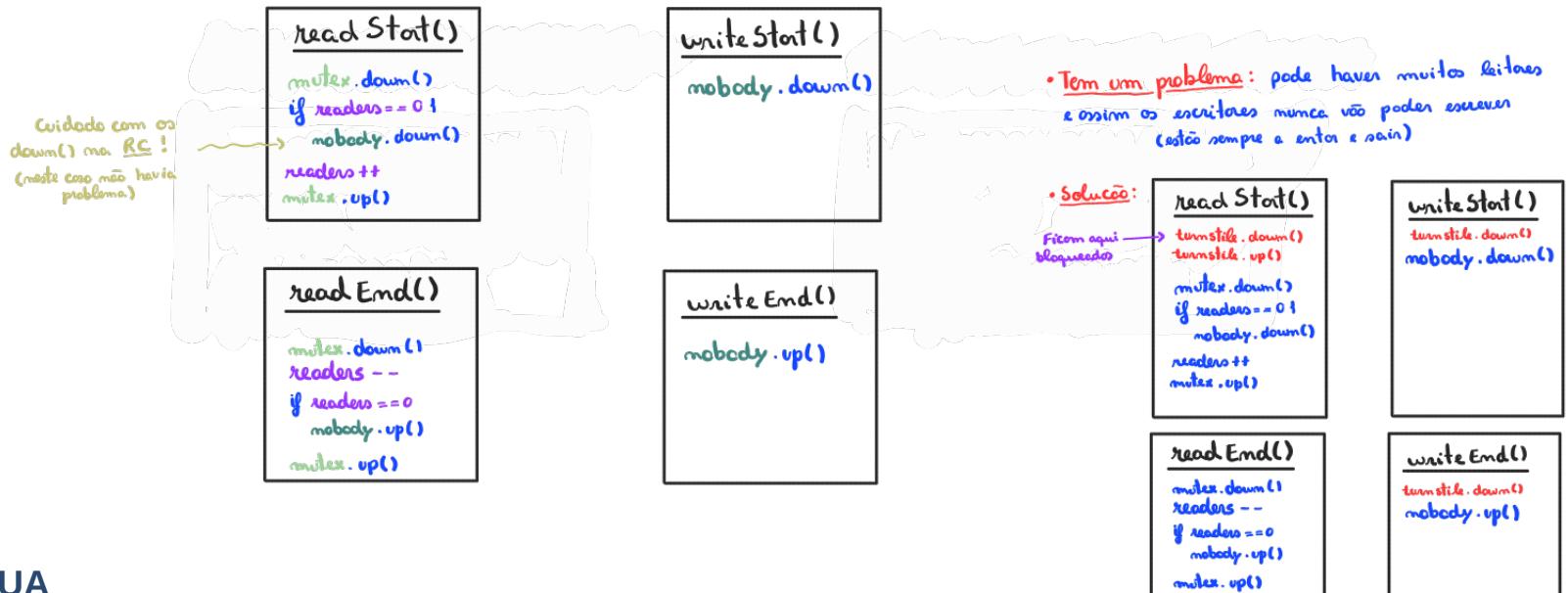
Não é possível!
→ Falta um... "turnstile"

Processos Leitores

readStart()
read access
readEnd()

Processos Escritores

writeStart()
write access
writeEnd()



Escritores e Leitores

- Usando 2 semáforos (**mutex** e **nobody**) e um inteiro (**readers**)

Processos Leitores

```
readStart()  
  mutex.down()  
  if(readers==0)  
    nobody.down()  
  readers++  
  mutex.up()  
read access  
readEnd()  
  mutex.down()  
  readers--  
  if(readers==0)  
    nobody.up()  
  mutex.up()
```

Processos Escritores

```
writeStart()  
  nobody.down()  
write access  
writeEnd()  
  nobody.up()
```

Deadlock impossível.
Adiamento indefinido de
escritores *possível.*



Porquê?

Escritores e Leitores

- Usando 3 semáforos (**mutex**, **nobody** e **turnstile**) e um inteiro (**readers**)

Processos Leitores

```
readStart()  
turnstile.down()  
turnstile.up()
```

igual a slide ant.

read access

```
readEnd()
```

igual a slide ant.

Processos Escritores

```
writeStart()  
turnstile.down()  
nobody.down()  
write access  
writeEnd()  
turnstile.up()  
nobody.up()
```

Escritores e Leitores

- Mecanismo de sincronização através do qual:
 - Existem dados partilhados
 - Podem ocorrer várias leituras em simultâneo, desde que não estejam a ocorrer escritas
 - Durante a escrita não podem existir leituras, nem outras escritas concorrentes

Processos Leitores

```
readStart()  
read access  
readEnd()
```

Processos Escritores

```
writeStart()  
write access  
writeEnd()
```

Escritores e Leitores

- Usando 2 semáforos (**mutex** e **nobody**) e um inteiro (**readers**)
- **mutex** e **nobody** inicializados a 1
- **readers** inicializado a 0

Processos Leitores

```
readStart()  
read access  
readEnd()
```

Processos Escritores

```
writeStart()  
write access  
writeEnd()
```

Escritores e Leitores

- Usando 2 semáforos (**mutex** e **nobody**) e um inteiro (**readers**)

Processos Leitores

```
readStart()  
  mutex.down()  
  if(readers==0)  
    nobody.down()  
  readers++  
  mutex.up()  
read access  
readEnd()  
  mutex.down()  
  readers--  
  if(readers==0)  
    nobody.up()  
  mutex.up()
```

Processos Escritores

```
writeStart()  
  nobody.down()  
write access  
writeEnd()  
  nobody.up()
```

Deadlock impossível.
Adiamento indefinido de escritores possível.

Porquê?

Escritores e Leitores

- Usando 3 semáforos (**mutex**, **nobody** e **turnstile**) e um inteiro (**readers**)

Processos Leitores

```
readStart()  
turnstile.down()  
turnstile.up()
```

igual a slide ant.

read access

```
readEnd()
```

igual a slide ant.

Processos Escritores

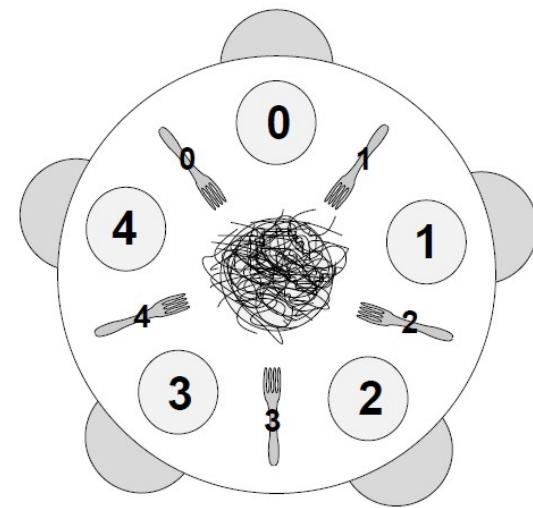
```
writeStart()  
turnstile.down()  
nobody.down()  
write access  
writeEnd()  
turnstile.up()  
nobody.up()
```

Jantar de filósofos

- Problema clássico de sincronização
 - Proposto por Dijkstra em 1965
 - Mesa redonda; 5 filósofos; 5 garfos
 - Filósofos alternam entre pensar e comer
 - Apenas conseguem comer se tiverem 2 garfos
 - Ciclo de vida dos filósofos

Processo Filósofo

```
while(true)
    think()
    getForks()
    eat()
    putForks()
```



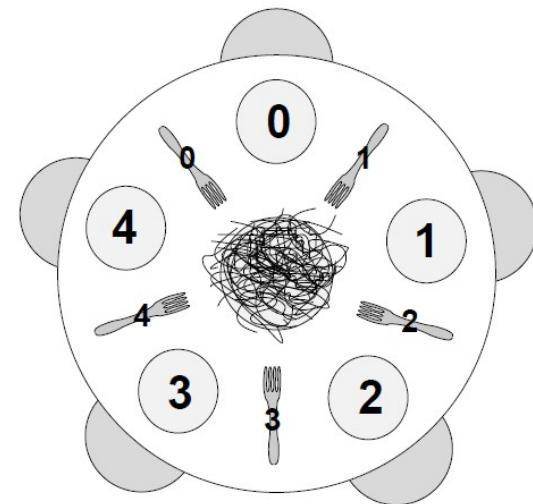
From: The Little Book of Semaphores; Allen B. Downey

Jantar de filósofos

- **getForks()** e **putForks()** devem respeitar:
 - Apenas 1 filósofo pode segurar 1 dado garfo
 - Deadlock deve ser impossível
 - Nenhum filósofo deve morrer à fome (adiamento indefinido)
 - Deve ser possível que mais do que 1 filósofo coma ao mesmo tempo

Processo Filósofo

```
while(true)
    think()
    getForks()
    eat()
    putForks()
```



From: The Little Book of Semaphores; Allen B. Downey

Jantar de filósofos

Usando 5 semáforos (array forks)

Processo Filósofo f

```
while(true)
    think()
    getForks()
        forks[left(f)].down()
        forks[right(f)].down()
    eat()
    putForks()
        forks[left(f)].up()
        forks[right(f)].up()
```

Outra solução...

Exemplo Deadlock:

- todos os filósofos pegam o garfo da esquerda.
- Pegando o down [left(f)]...
- como todos estão à espera ninguém vai comer e disponibilizar o semáforo

Semáforos

```
fork[0]=1
fork[1]=1
fork[2]=1
fork[3]=1
fork[4]=1
```

Começam livres (1)

Processo Filósofo f

```
while(true)
    think()
    getForks()
        mutex.down()
        forks[left(f)].down()
        forks[right(f)].down()
        mutex.up()
    eat()
    putForks()
        forks[left(f)].up()
        forks[right(f)].up()
```

Podem ficar presos no RC
e depois ocorrem Dead Locks



Processo Filósofo f

```
while(true)
    think()
    getForks()
        limit4.down()
        forks[left(f)].down()
        forks[right(f)].down()
    eat()
    putForks()
        forks[left(f)].up()
        forks[right(f)].up()
    limit4.up()
```

Outra solução

Não é perfeita pois pode
não ser possível 2 pessoas
comerem em simultâneo

F1, F2, F3, F4
✓ ✗ ✗ ✗
Bloqueados...

Processo Filósofo

```
while(true)
    think()
    getForks()
    eat()
    putForks()
```

Processo Filósofo f

```
while(true)
    think()
    getForks()
    eat()
    putForks()
```

Deadlock possível.

Porque?

right(f)
return f

left(f)
return (f+1)%5

Solução de Tonembau

- mutex
- 5 semáforos (filos)
- 5 variáveis de estado (array st)

Estado:
→ TK (thinkKing)
→ H6 (hungry)
→ ET (eating)

filo[0]=0	st[0]=TK
filo[1]=0	st[1]=TK
filo[2]=0	st[2]=TK
filo[3]=0	st[3]=TK
filo[4]=0	st[4]=TK

mutex=1

Processo Filósofo f

```
while(true)
    think()
    getForks()
        mutex.down()
        st[f]=HG
        test(f)
        mutex.up()
        file[f].down()
    eat()
    putForks()
        mutex.down()
        st[f]=TK
        test(left(f))
        test(right(f))
        mutex.up()
```

test(f):
if(st[f]==HG &&
st[left(f)]!=ET &&
st[right(f)]!=ET)
→ st[f]=ET
→ file[f].up()

Deadlock

???: Importante

- Quando ocorre deadlock há quatro condições que se verificam:
 - Condição de exclusão mútua
 - Cada recurso ou está livre ou foi atribuído a um e um só processo
 - Condição de espera com retenção
 - Cada processo, ao requerer um novo recurso, mantém na sua posse os recursos anteriormente solicitados
 - Condição de não libertação
 - Ninguém, a não ser o próprio processo, pode decidir da libertação de um recurso que lhe tenha sido atribuído
 - Condição de espera circular
 - Formou-se uma cadeia circular de processos e recursos em que cada processo requer um recurso que está na posse do processo seguinte na cadeia

Jantar de filósofos

- Usando 5 semáforos (array **forks**) e 1 semáforo (**limit4**)
- **limit4** impede que os 5 filósofos tentem pegar em garfos ao mesmo tempo.

Processo Filósofo f

```
while(true)
    think()
    getForks()
        limit4.down()
        forks[left(f)].down()
        forks[right(f)].down()
    eat()
    putForks()
        forks[left(f)].up()
        forks[right(f)].up()
    limit4.up()
```

Deadlock impossível.

Porquê?

Jantar de filósofos

- Usando 5 semáforos (array **forks**)
- Se pelo menos 1 filósofo pegar primeiro no garfo à esquerda e pelo menos 1 filósofo pegar primeiro no garfo à direita

Deadlock impossível.

Porquê?

Jantar de filósofos

- Solução de Tanembaum
- Usando 1 semáforo (**mutex**), 5 semáforos (**array filos**) e 5 variáveis de estado (**array st**)
- Estados possíveis: thinking (TK), hungry (HG), eating (ET)

```
getForks()
mutex.down()
st[f]=HG
test(f)
mutex.up()
filos[f].down()
```

```
putForks()
mutex.down()
st[f]=TK
test(left(f))
test(right(f))
mutex.up()
```

```
test(f)
if(st[f]==HG)
    and st[left(f)]!=ET
    and st[right(f)]!=ET
    st[f]=ET
    filos[f].up()
```

Deadlock impossível.

Porquê?

Prevenção de deadlock

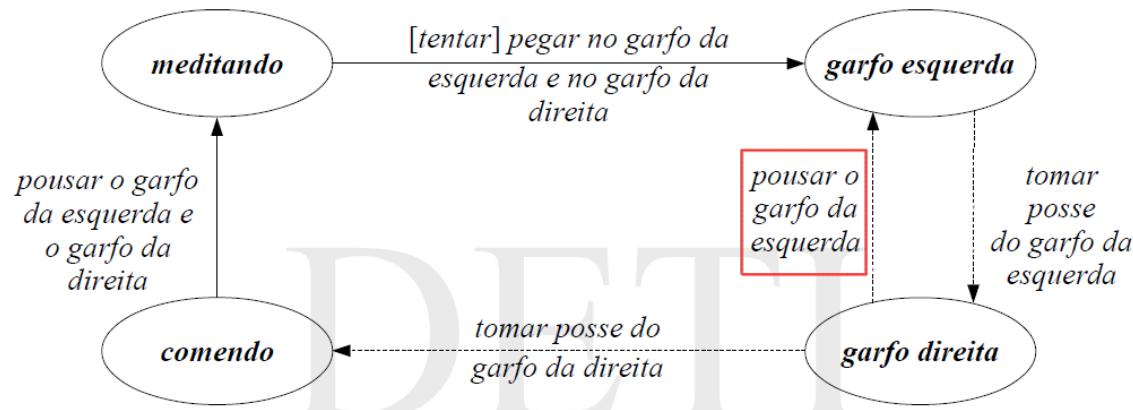
- Negar condição de espera com retenção

- Solicitar todos os recursos de uma só vez;
Ex: algoritmo de Tanembaum do Jantar de Filósofos

→ Ou pega mos
2 ou não pega ?
em menturm

- Impondo libertação de recursos

- Ex: Se não consegue ambos os garfos, liberta o que conseguiu



Um pegaom primeiro
pela esquerda e
outros pela diraita

- Negando a espera circular

- Ordenando os recursos e fazendo com que a requisição dos recursos seja