Sistemas Operativos

Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

Ano letivo 2024/2025

Nuno Lau (nunolau@ua.pt)

Exclusão mútua

- Usando 1 semáforo (**`mutex`**)
- Semáforo inicializado com valor 1
- Processos executam **`mutex.down()`** antes da **`região crítica`** e **`mutex.up()`** depois da **`região crítica`**

Processo 1

```
Código A
mutex.down()
região crítica
mutex.up()
Código B
```

Processo 2

```
Código C
mutex.down()
Região crítica
mutex.up()
Código D
```

Signaling

- 1 semáforo (**`sem`**) é suficiente
  - Semáforo inicializado com valor 0
  - Processo 2 faz **`down()`** do semáforo antes de **`Código D`**
    - Garantindo que espera por um up
  - Processo 1 faz **`up()`** depois de **`Código A`**
    - Sinalizando processo 2 de que pode executar D.

Processo 1

```
Código A
sem.up()
Código B
```

Processo 2

```
Código C
sem.down()
Código D
```

Rendezvous

- Usando 2 semáforos (**`arrived1`** e **`arrived2`**)
  - Semáforos inicializados com valor 0
  - Processo 1 faz **`arrived1.up()`** e **`arrived2.down()`** após **`Código A`**
    - Garantindo que espera por um **`arrived2.up()`**
  - Processo 2 faz **`arrived1.up()`** e **`arrived2.down()`** após **`Código C`**
    - Garantindo que espera por um **`arrived1.up()`**

Processo 1

```
Código A
arrived1.up()
arrived2.down()
Código B
```

Processo 2

```
Código C
arrived2.up()
arrived1.down()
Código D
```

Barreira

- Solução genérica
  - 1 semáforo para exclusão mútua (**`mutex`**), N semáforos para barreira (*array* **`barrier`**), 1 inteiro partilhado (**`count`**)

Processo j j∈1..N

```
Código j.A
mutex.down()
count++;
if(count == N) {
    for(i=1..N) barrier[i].up()
    count=0;
}
mutex.up ()
barrier[j].down()
Código j.B
```

Barreira

- Solução genérica
  - 1 semáforo para exclusão mútua (**`mutex`**), 2 semáforos para barreira (array **`barrier`**), 2 inteiros partilhados (**`count`** e **`turn`**)

Processo j j∈1..N

```
Código j.A
mutex.down()
count++; localt=turn;
if(count == N) {
    for(i=1..N) barrier[localt].up()
    count=0; turn=1-turn;
}
mutex.up ()
barrier[localt].down()
Código j.B
```

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);
       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

Processos Leitores

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

Processos Escritores

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

- Implementação usando semáforos?

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()
```

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()
```

- Implementação usando semáforos?

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

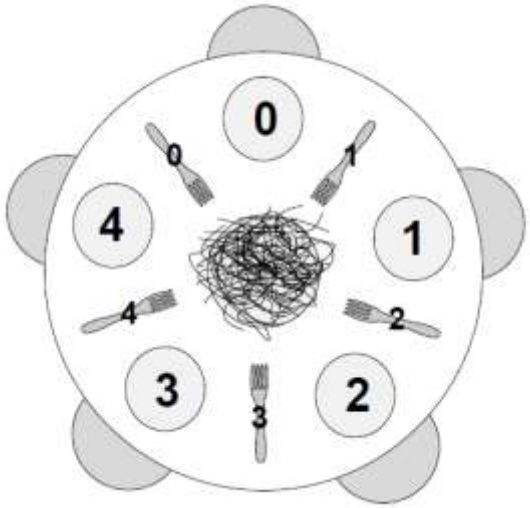
```
writeStart()
   turnstile.down()
   nobody.down()
write access
writeEnd()
   turnstile.up()
   nobody.up()
```

Jantar de filósofos

- Problema clássico de sincronização
  - Proposto por Dikjstra 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

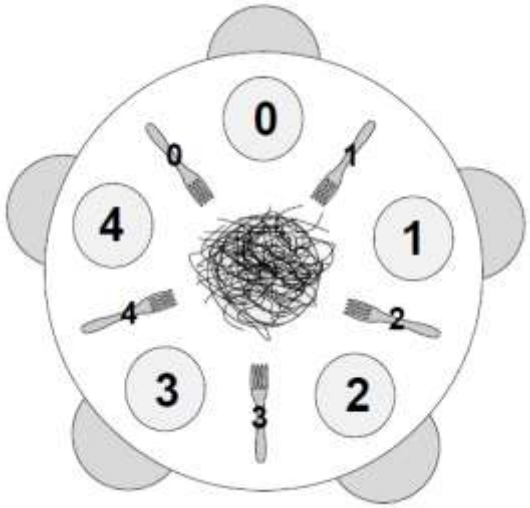
  - Implementação de **`getForks()`** e **`putForks()`** usando semáforos?

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

- Implementação de **`getForks()`** e **`putForks()`** usando semáforos?

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()
```

Deadlock possível.

Porquê?

Deadlock

- 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()
```

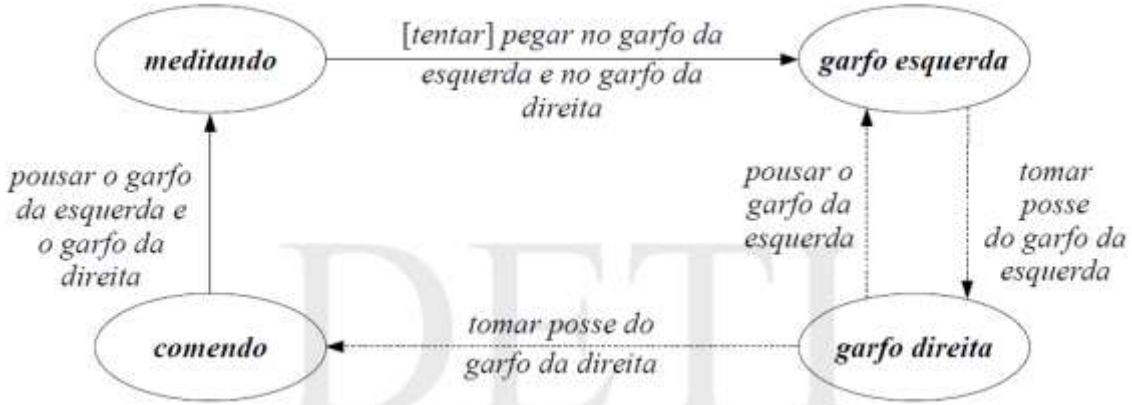
```
test(f)
  if(st[f]==HG)
    and st[left(f)]!=ET
    and st[right(f)]!=ET
    st[f]=ET
    filos[f].up()
```

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
- Impondo libertação de recursos
  - Ex: Se não consegue ambos os garfos, liberta o que conseguiu



- Negando a espera circular
  - Ordenando os recursos e fazendo com que a requisição dos recursos seja efectuada por ordem
    Ex: um filosófo começa pelo garfo direito; todos os outros pelo esquerdo;