Sistemas Operativos

Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

Ano letivo 2023/2024

Nuno Lau (nunolau@ua.pt)

Semáforos

- Tipo de dados abstrato que permite sincronização de *threads*/processos sem *busy waiting*
- Semáforo tem um estado interno que é um **valor inteiro**
- Podem realizar-se **operações atómicas** de incremento e decremento da variável interna
- Semáforo **bloqueia se a operação torna o valor do semáforo negativo**

Deadlock e Adiamento indefinido

- *Deadlock*
  - 2 ou mais processos estão bloqueados à espera de um evento que apenas pode ser despoletado por um dos processos em bloqueio
  - Se S e Q forem 2 semáforos inicializados a 1

P0	P1
S.acquire()	Q.acquire()
Q.acquire()	S.acquire()
…	…
Q.release()	S. release()
S.release()	Q. release()

- Adiamento indefinido (*starvation*)
  - Um processo pode nunca ser removido da fila de espera de um semáforo

Exclusão mútua

- Mecanismo básico de sincronização através do qual se garante o acesso em exclusão mútua a determinadas zonas de código (**região crítica**)

semáforo inicializado a 1

Processo 1

```
Código A
down(s)
Região crítica
up(s)
Código B
```

Processo 2

```
Código C
down(s)
Região crítica
up(s)
Código D
```

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

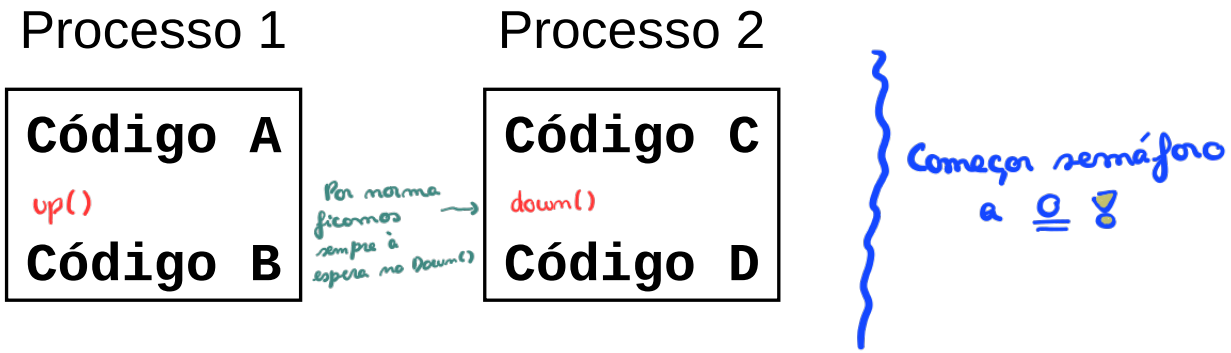
```
Código A
mutex.down()
região crítica
mutex.up()
Código B
```

Processo 2

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

Signaling

- Mecanismo básico de sincronização através do qual um processo avisa outro de que algo aconteceu
- Permite impôr que determinadas secções de código sejam precedidas pela execução de secções de código em processos distintos.
  - Serialização de código em processos distintos
  - Ex: **`Código D`** apenas pode executar depois de **`Código A`**



Implementação usando semáforos?

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

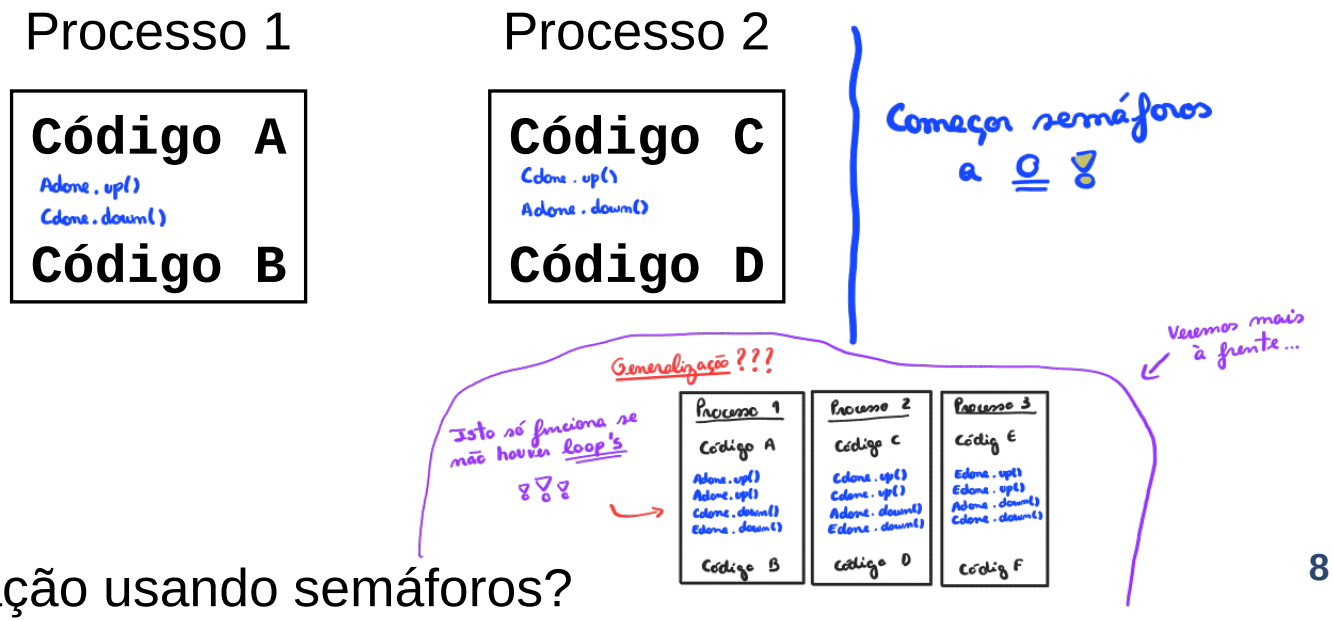
```
Código A
sem.up()
Código B
```

Processo 2

```
Código C
sem.down()
Código D
```

Rendezvous

- Mecanismo básico de sincronização através do qual dois processos se “encontram” antes de continuar
- Permite sincronizar determinadas operações em processos distintos
  - Ex: Processo 1 e 2 apenas avançam para **`Código B`** e **`Código D`** se **`Código A`** e **`Código C`** estiverem concluídos



- Implementação usando semáforos?

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 **`arrived2.up()`** e **`arrived1.down()`** após **`Código C`**
    - Garantindo que espera por um **`arrived1.up()`**

Processo 1	Processo 2
`Código A` ***`arrived2.down()`*** ***`arrived1.up()`*** **`Código B`**	**`Código C`** ***`arrived1.down()`*** ***`arrived2.up()`*** **`Código D`**

Deadlock!

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

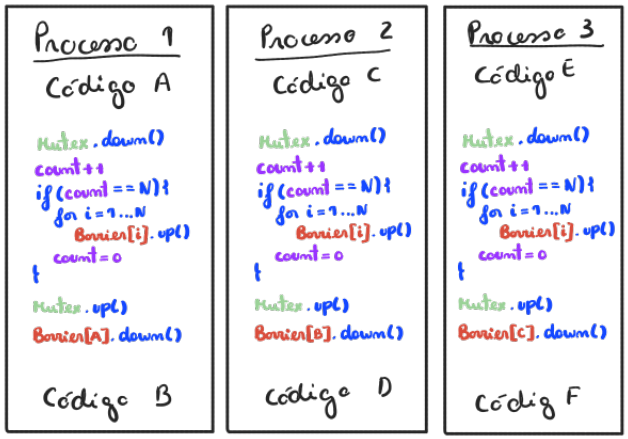
⇒ Sincronização entre processos/threads

↳ Normalmente existem bibliotecas que implementam isto

- Generalização de *Rendezvous* para mais do que 2 processos
- Solução anterior de *Rendezvous* não é generalizável
  - Porquê?
- Solução genérica
  - 1 semáforo para exclusão mútua (**`mutex`**), 1 semáforo para barreira (**`barrier`**), 1 inteiro partilhado (**`count`**)
  - **`mutex`** inicializado com valor 1, **`barrier`** com valor 0
  - **`count`** inicializado com valor 0

↳ Não pode ser! (array de N barreiras)

N = nº de processos →



⟸ mutex inicializado a 1 (Exclusão mútua)
Barrier[...] inicializados a 0 (Espera de uns pelos outros)
count (variável) inicializada a 0 (Quantos estão na barreira?)

Barreira

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

Processo j j ∈ 1..N

```
Código j.A
mutex.down()
bool localblk = false;
if(count == N-1) {
   for(i=1..N-1) barrier.up()
   count=0;
}
else {
   count++; localblk=true;
}
mutex.up ()
if(localblk) barrier.down()
Código j.B
```

Funciona se apenas 1 vez.
Pode dar problemas se barreira for cíclica.

Porquê?

→ Inia subir e fazer ou barrier.down() ...

Barreira

- Solução genérica mais simples
  - 1 semáforo para exclusão mútua (**`mutex`**), 1 semáforo para barreira (**`barrier`**), 1 inteiro partilhado (**`count`**)

Processo j j ∈ 1..N

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

Funciona se apenas 1 vez.
Pode dar problemas se barreira for cíclica.

Porquê?

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

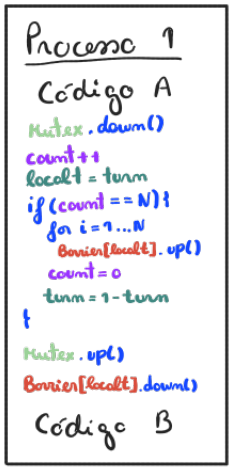
- Solução genérica
  - 1 semáforo para exclusão mútua (**`mutex`**), 2 semáforos para barreira (array **`barrier`**), 2 inteiros partilhados (**`count`** e **`turn`**)

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
```

variável local

⇒ Poupar semáforos!



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