

Sistemas Operativos

Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

Ano letivo 2023/2024

Nuno Lau (nunolau@ua.pt)

- A solução apresentada anteriormente não é segura!
- Se um produtor e um consumidor executarem `insert()` e `remove()` ao mesmo tempo, `count` pode não ser actualizado correctamente
- Para manter a consistência do buffer é necessário que estes métodos sejam sempre executados por apenas 1 *thread* de cada vez!

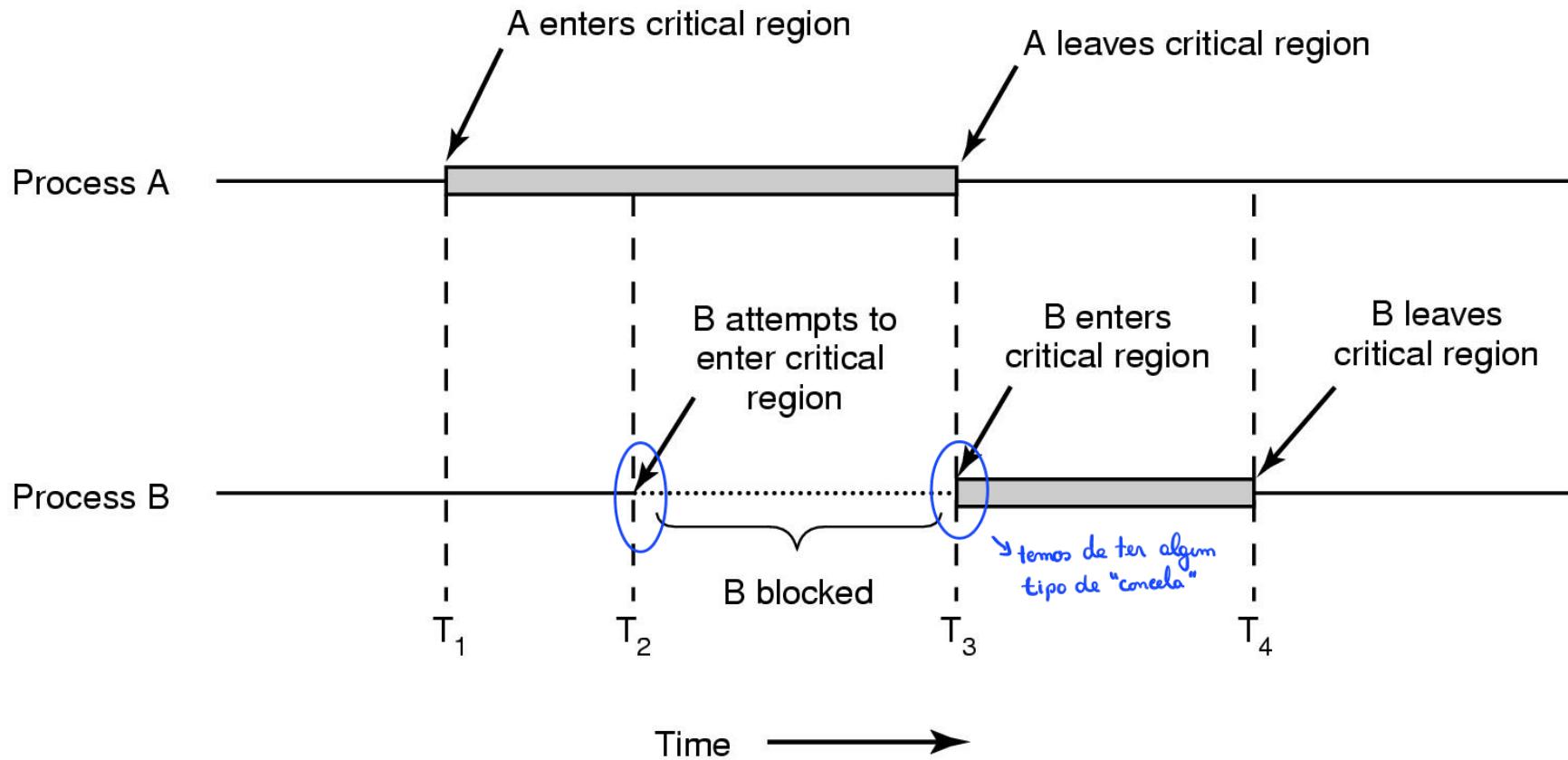
⇒ Exclusão Mútua no acesso a estes métodos



Definições

- **Condição de corrida** (*Race Condition*)
 - Quando vários processos/*threads* acedem a dados partilhados e o resultado final depende de forma inesperada da ordem de execução
- **Região crítica** • Dentro da região crítica vamos ter exclusão mútua
 - **Zona de código que manipula dados partilhados** e que não pode ser executada concorrentemente por mais do que um processo/*thread*

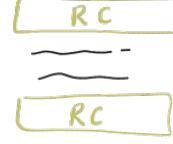
Região Crítica



Condições para Região Crítica

- **Exclusão Mútua**
 - Se um processo P_i está a executar na sua região crítica então nenhum dos outros processos pode estar em execução nas suas regiões críticas
- **Progresso**
 - Se nenhum processo está em execução em regiões críticas e pelo menos um processo pretende o acesso à região crítica então a seleção do processo que deverá ter acesso a esta região não pode ser adiada indefinidamente
⇒ Temos de deixar entrar ...
- **Espera limitada**
 - Deve existir um limite ao número de vezes que é concedido o acesso a outros processos à região crítica, após um determinado processo ter pedido esse acesso e até que esse pedido seja satisfeito
- **Não há nenhum pressuposto sobre a velocidade ou número de CPUs**
⇒ Condições para qualquer tipo de Hardware! //

Definições

- **Condição de corrida**
 - Quando vários processos/*threads* acedem a dados partilhados e o resultado final depende de forma inesperada da ordem de execução
 - **Região crítica** *Região em que pode acontecer...* → Dependem da ordem de execução! //
 - **Zona de código que manipula dados partilhados** e que não pode ser executada concorrentemente por mais do que um processo/*thread*
 - **Região de entrada** → Pedido de entrada (tempo limitado)
 - **Região de saída** → Liberta a região crítica!
- Pode ser uma região não contínua !!!*
- 

Estrutura típica

```
while (true) {
```

entry section

→ Pedido de acesso
à região crítica

critical section

→ Região crítica

exit section

→ Liberta a
região crítica

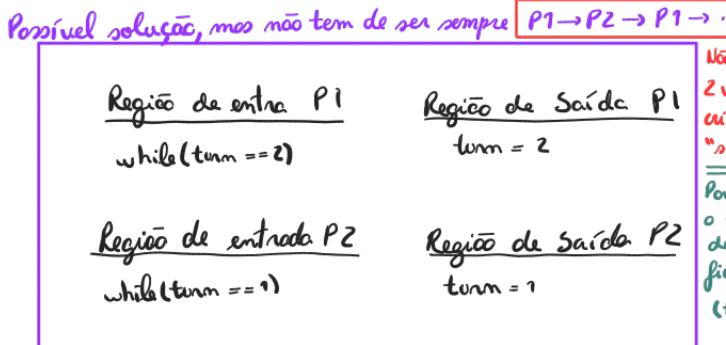
remainder section

→ O resultado não depende
da ordem de
execução

```
}
```

Soluções por software

- Variável partilhada de lock
 - Valor=0 se Região Crítica não está a ser usada
 - Valor=1 se Região Crítica está a ser usada
 - Não funciona se implementado apenas em software
- Alternância estrita
- Algoritmo de Dekker (1964)
- Algoritmo de Peterson (1981)



Região de entrada X Região de saída
 $\text{while}(\text{valor} == 1);$
 $\text{valor} = 0$

Não faz Exclusão Mútua:

P1:
 $\text{while}(\text{valor} == 1);$
{ Escalonadas alternadamente ...
 $\text{valor} = 1$
Entrou RC

Saiu e entrou (valor = 0) while (valor == 1)
valor = 1 Entrou RC

Não existe solução perfeita a nível de Software

→ implica sempre um "busy waiting"
→ E como os processadores são optimizados o CPU podem mudar a ordem das instruções isso pode criar problemas ...

Alternância estrita

- Variável **turn** controla acesso à região crítica
→Termos o problema da alternância estrita! //

Processo 0

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

Processo 1

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

Algoritmo de Peterson

- 2 processos (pode ser generalizado para mais)
- 2 variáveis partilhadas
 - `int turn` - Usada para indicar de quem é a vez de entrar (em caso de conflito)
 - `boolean flag[2]` - Usada para indicar que processo pretende acesso à região crítica ↳ Array de flags
- 2 processos: i e j
- Código para processo i :

Partilhados
 $Turn = j$
 $Flag[i] = \text{TRUE}$
 $Flag[j] = \text{FALSE}$

```
while (true) {
```

```
    Flag[i] = \text{TRUE}  
    turn = j  
    while (Flag[j] && turn == j);  
        RC  
    Flag[i] = \text{FALSE}
```

```
    flag[i] = \text{TRUE};  
    turn = j;  
    while (flag[j] && turn == j);
```

critical section

```
    flag[i] = \text{FALSE};
```

remainder section

Processo i
 $Flag[i] = \text{TRUE}$
 $Turn = j$

 $\text{while}(\text{Flag}[j] \&\& Turn == j)$
Entra na RC

Processo j
 $Flag[j] = \text{TRUE}$
 $Turn = i$

 $\text{while}(\text{Flag}[i] \&\& Turn == i)$
F T

Funciona! Mas continua a existir o "busy waiting"

}

Soluções Hardware

- Muitos sistemas tem suporte de hardware para facilitar a implementação de regiões críticas
- Sistemas uniprocessador – podem desactivar interrupções \Rightarrow *Interrupt disable, desativa a utilização da CPU!*
 - Código é executado sem preempção
 - Ineficiente em sistemas multiprocessador
- Instruções atómicas (não interrompíveis) especiais
 - Testam valor na memória e alteram esse valor
 - Ou Trocam o valor de 2 posições de memória

Interrupt disable
Entrar na RC
Interrupt enable

Problemas:
• Estamos a assumir que o utilizador tem controlo do sistema
• Para vários processadores

↓
Instruções atómicas!
(vai até ao fim)

Soluções Hardware

```

public class HardwareData {
    private boolean value = false;

    public HardwareData(boolean value) {
        this.value = value;
    }

    public boolean get() {
        return value;
    }

    public void set(boolean newValue) {
        value = newValue;
    }

    public boolean getAndSet(boolean newValue) {
        boolean oldValue = this.get();
        this.set(newValue);

        return oldValue;
    }

    public void swap(HardwareData other) {
        boolean temp = this.get();

        this.set(other.get());
        other.set(temp);
    }
}
  
```

Isto é Software!
(não funcionava...)

← **testa e altera de forma atómica**

Uma única instrução Assembly...

← **troca de forma atómica**

Solução usando getAndSet

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
    while (lock.getAndSet(true))
        Thread.yield(); → Deixa de utilizar o CPU
    criticalSection();
    lock.set(false);
    remainderSection();
}
```

Com TesteAnd Set

Região de entrada

do{

TesteAndSet r1, shared
{while (r1 == 1)
 ← Faz isso
 de forma
 atómica!//}

Região de saída

shared = 0



Só sai do ciclo se valor de retorno de getAndSet for false, ou seja, se o lock estava “livre”

Com Swap

Região de entrada

local = 1

do{
 swap local, shared
 † while (local == 1)

Região de saída

shared = 0

← shared = 1
local vai ser 1
ou 0, dependendo
do Shared

Solução usando swap

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

// each thread has a local copy of key
HardwareData key = new HardwareData(true);

while (true) {
    key.set(true);

    do {
        lock.swap(key);
    }
    while (key.get() == true);

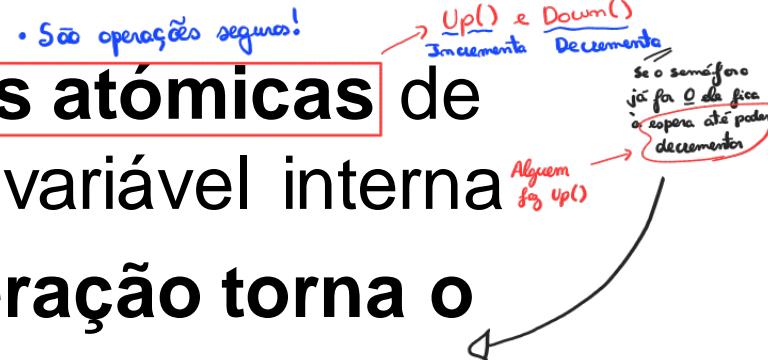
    criticalSection();
    lock.set(false);
    remainderSection();
}
```



Só sai do ciclo se valor anterior
de lock for false,
ou seja, se o lock estava “livre”

- Tipo de dados abstrato que permite sincronização de *threads/processos* sem *busy waiting* → Permite a sincronização, e.g.: para a exclusão mútua!
- 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**

Se o valor for 0, esse down() fica bloqueado até o valor ser maior que zero // Esperar por um up() de outro processo!



Semáforos

- Ferramenta de sincronização que não necessita de *busy waiting*
- Semáforo S
 - variável inteira
 - 2 métodos de acesso:
 - release(); up(); P()
 - acquire(); down(); V()

Up()

```
acquire() {
    while value <= 0
        ; // no-op
    value--;
    Isto não é
    feito assim!!
}
```

„É Executado de forma atómica!“

incremento
decremento

Down()

```
release() {
    value++;
}
```

Estas instruções são
SEGURAS

Semáforos

- Semáforos podem considerar que:
 - Variável interna pode tomar qualquer valor inteiro
 - Variável interna é binária
 - Por vezes estes semáforos são designados de mutex ou lock

O segundo processo fica bloqueado

Valor inicial a 1

Mutual Exclusion

```
Semaphore S = new Semaphore();  
S.acquire();      ← Se fizermos isto bloqueamos o semáforo ( semáforo = 0 )  
                ↓  
// critical section !  
                ↑  
O valor do semáforo → aqui está a 0,  
Mais ninguém entra  
S.release();      ← Desbloqueia o semáforo  
        .up()  
// remainder section
```

Processo P1

→ Down()
→ Entra R.C
 ↓
 → Up()
 ← R.C

Processo P2

→ Down()

(->) espera (->)
Entra na R.C
 ↓
 → Up()
 ← R.C

Implementação de Semáforos

- Quando não é possível terminar acquire imediatamente, o semáforo, em geral, bloqueia o processo numa fila de espera própria
 - Block* – bloqueia o processo que tem de esperar pelo semáforo
 - Wakeup* – acorda um/vários processos da fila de espera
- Deve garantir que não existem 2 processos a executar *acquire* ou *release* simultaneamente
 - Estas funções constituem regiões críticas

SC

```
acquire(){  
    value--;  
    if (value < 0) {  
        add this process to list  
        block; ← Bloqueados!  
    }  
}
```

Do ponto de vista do CPU, os processos bloqueados não ficam a ocupar o CPU, estão apenas na lista de espera

```
release(){  
    value++;  
    if (value <= 0) {  
        remove a process P from list  
        wakeup(P);  
    }  
}
```



Deadlock e Adiamento indefinido

• **Deadlock** → Acontece muito ?

- 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
S.acquire()
Q.acquire()
...
Q.release()
S.release()

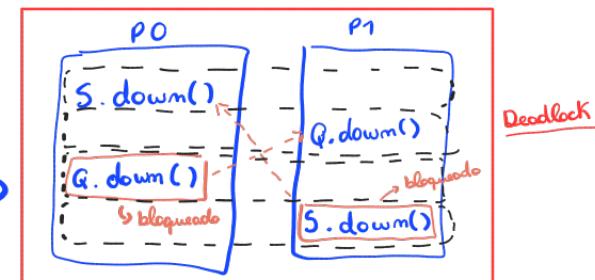
P1
Q.acquire()
S.acquire()
...
S. release()
Q. release()

Para eu te "abrir" preciso de
me "abrir", só que tu para me
"abrires" preciso que eu
te "abra". } Impossível de sair !

Processo P0:
S.down()
Q.down()
S.down()

Processo P1:
Q.down()
S.down()

Dead Lock



• Adiamento indefinido (starvation)

- Um processo pode nunca ser removido da fila de espera de um semáforo → Tentou fazer um down(), ficou na fila, e você entrou noutro processo à frente dele.