Sistemas Operativos

Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

Ano letivo 2024/2025

Nuno Lau (nunolau@ua.pt)

Problema do produtor-consumidor
Solução Java com memória partilhada

- 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**

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
- **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**

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**
  - **Zona de código que manipula dados partilhados** e que não pode ser executada concorrentemente por mais do que um processo/*thread*
- **Região de entrada**
  - Código que realiza o pedido de acesso à região crítica
- **Região de saída**
  - Código executado após a saída da região crítica

Estrutura típica

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

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)

Alternância estrita

- Variável **turn** controla acesso à região crítica

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
- 2 processos: **`i`** e **`j`**
- Código para processo **`i`**:

```
while (true) {

        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);

            critical section

        flag[i] = FALSE;

            remainder section

}
```

Soluções Hardware

- Muitos sistemas tem suporte de hardware para facilitar a implementação de regiões críticas
- Sistemas uniprocessador – podem desactivar interrupções
  - 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

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);
    }
}
```

testa e altera de forma atómica

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();

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

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

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”

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**

Semáforos

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

```
acquire() {
    while value <= 0
       ; // no-op
    value--;
}
```

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

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*

```
Semaphore S = new Semaphore();

S.acquire();

   // critical section

S.release();

   // remainder section
```

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

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

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

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**)

Processo 1

```
Código A

Região crítica

Código B
```

Processo 2

```
Código C

Região crítica

Código D
```

- Implementação usando semáforos?

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`**

Processo 1	Processo 2
`Código A`	**`Código C`**
`Código B`	**`Código D`**

  - 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

Processo 1	Processo 2
`Código A`	**`Código C`**
`Código B`	**`Código D`**

  - 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

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

Processo 2

```
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

- 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

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?

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
```