

SINCRONIZAÇÃO DE PROCESSOS

- ◆ O problema das secções críticas
- ◆ Soluções baseadas em *software*
- ◆ Soluções baseadas em *hardware*
- ◆ O mecanismo de sincronização básico: semáforos
- ◆ Mecanismos de sincronização mais sofisticados: monitores, passagem de mensagens, regiões críticas
- ◆ Problemas clássicos de sincronização



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Execução concorrente

- ◆ Execução concorrente
 - execução cooperante ou não-cooperante logicamente ao mesmo tempo (ex: multiprogramação)
- ◆ Execução em paralelo
 - execução cooperante ou não-cooperante fisicamente ao mesmo tempo (ex.: multiprocessamento)
- ◆ A execução concorrente pode ocorrer em
 - sistemas uniprocessador
 - interlçamento de execução (multiprogramação)
 - sistemas multiprocessador
 - interlçamento e sobreposição de execução (multiprogramação c/ multiprocessamento)
- ◆ Os processos concorrentes precisam frequentemente de partilhar dados / recursos.
- ◆ O acesso concorrente a dados partilhados pode resultar em inconsistência desses dados, se o acesso não ocorrer de forma controlada.
- ◆ Execução concorrente ⇒
 - mecanismos de comunicação entre processos
 - sincronização entre as suas acções



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Necessidade de Sincronização - Exemplo 1

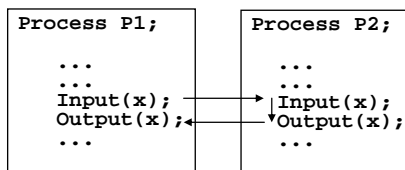
- ◆ Os processos P1 e P2 executam o seguinte código tendo acesso à mesma variável x.

```
Process P1;
Begin
  ...
  Input(x);
  Output(x);
  ...
End;
```

```
Process P2;
Begin
  ...
  Input(x);
  Output(x);
  ...
End;
```

- ◆ Os processos podem ser interrompidos em qualquer ponto.

- ◆ Se P1 for interrompido após a entrada de dados e P2 executar inteiramente então o carácter ecoado por P1 será o que foi lido por P2 !!!



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo 2

- ◆ Dois processos
 - um lê caracteres do teclado
 - outro ecoa os caracteres lidos para o écran
 - os caracteres são inseridos num *buffer* circular
 - uma variável partilhada, Count, indica o nº de caracteres contidos no *buffer*

```
Process Keyboard;
Begin
  ...
  Count:=Count+1;
  ...
End;
```

```
Process Display;
Begin
  ...
  Count:=Count-1;
  ...
End;
```

- ◆ O que parece uma operação única (actualização da variável count) pode ser traduzido numa série de instruções-máquina.
Ex.:

```

...
Count:=Count+1;  →  load  A, Count
...                add   A, 1
                   store A, Count
  
```



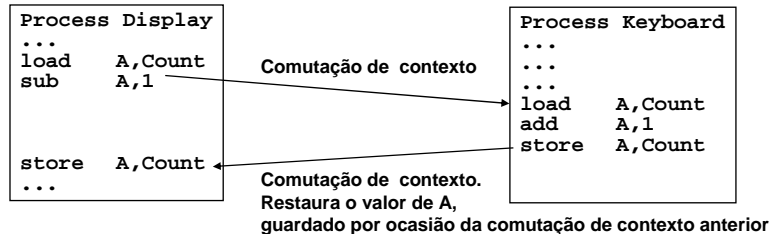
FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo 2 (cont.)

- ◆ A execução concorrente pode causar um problema (*race condition*):



- ◆ Admitindo que o valor inicial de Count era 5, qual será o valor final ?
E qual deveria ser ?

◆ Race condition

- situação em que vários processos acedem e manipulam os mesmos dados, concorrentemente, e o resultado da execução depende da ordem em que se dá o acesso a esses dados.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo 3

```

struct {
    // nome, B.I., etc...;
    int saldo; } Conta;

int levantar (Conta *conta, int valor) {
    if (conta->saldo >= valor)
        conta->saldo = conta->saldo - valor;
    else
        valor = -1; // ERRO
    return valor;
}

```

Qual é o problema ?



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo 3 (cont.)

```

struct {
    // nome, B.I., etc...;
    int saldo; } Conta;

int levantar (Conta *conta, int valor) {
    if (conta->saldo >= valor)
        conta->saldo = conta->saldo - valor;
    else
        valor = -1; // ERRO
    return valor;
}

```

Process P1;

```

... levantar (...) {
    if (...) {
        conta->saldo =
        conta->saldo - valor;
    }
    ...
}

```

Process P2;

```

... levantar (...) {
    if (...) {
        ...
        conta->saldo =
        conta->saldo - valor;
    }
    ...
}

```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo 3 (cont.)

```

struct {
    // nome, B.I., etc...;
    int saldo; } Conta;

int levantar (Conta *conta, int valor) {
    if (conta->saldo >= valor)
        conta->saldo = conta->saldo - valor;
    else
        valor = -1; // ERRO
    return valor;
}

```

```

... mov AX,saldo
   mv BX,valor
   sub AX,BX
   mov Saldo,AX
   ...

```

Process P1;

```

... levantar (...) {
    if (...) {
        conta->saldo =
        conta->saldo - valor;
    }
    ...
}

```

Process P2;

```

... levantar (...) {
    if (...) {
        ...
        conta->saldo =
        conta->saldo - valor;
    }
    ...
}

```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

O problema das secções críticas

- ◆ Quando um processo executa código que manipula dados / recursos partilhados, diz-se que o processo está na sua secção crítica (para esses dados / recursos).
- ◆ A execução de secções críticas deve ser mutuamente exclusiva: em cada instante, apenas um processo poderá estar a executar na sua secção crítica (mesmo com múltiplos *CPUs*).
- ◆ Por isso, cada processo deve pedir autorização para entrar na sua secção crítica (SC).
- ◆ A secção de código que implementa este pedido é chamada a secção de entrada (SE).
- ◆ A SC é seguida por uma secção de saída (SS).
- ◆ O resto do código constitui a designada secção restante (SR).
- ◆ O problema das secções críticas é conceber um protocolo que os processos possam usar para que a sua acção não dependa da ordem pela qual a sua execução é interlaçada (mesmo com múltiplos *CPUs*).



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Pressupostos p/ a análise de soluções

- ◆ Cada processo executa com velocidade não nula mas não é feita qualquer suposição acerca da velocidade relativa dos processos.
- ◆ A estrutura geral de uma secção crítica é:

Secção de entrada;
Secção crítica;
Secção de saída;
Secção restante
- ◆ Podem existir vários *CPU's* mas o *hardware* de memória impede o acesso simultâneo à mesma posição de memória.
- ◆ Não são feitos pressupostos acerca da ordem de interlaçamento da execução.
- ◆ Nas soluções, é necessário especificar as secções de entrada e de saída.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Requisitos que uma solução do problema das SCs deve satisfazer

- ◆ Só um processo de cada vez pode entrar na secção crítica (exclusão mútua).
 - ◆ Um processo a executar numa secção não-crítica não pode impedir outros processos de entrar na secção crítica (progresso).
 - ◆ Um processo que peça para entrar numa secção crítica não deve ficar à espera indefinidamente (espera limitada).
-
- ◆ Não são feitos pressupostos acerca da velocidade relativa dos processadores (ou do seu número)
 - ◆ Supõe-se que um processo permanece numa secção crítica durante um tempo finito.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Tipos de soluções

- ◆ Soluções baseadas em software (do utilizador)
 - Código escrito pelo programador dos processos.
 - Baseadas em algoritmos cuja correcção não se baseia em nenhum pressuposto além dos anteriores.
- ◆ Soluções baseadas em hardware
 - Baseadas em instruções-máquina especiais.
- ◆ Soluções baseadas em serviços do Sistema Operativo / Linguagens
 - Baseadas em funções e estruturas de dados, fornecidas pelo S.O..
 - semáforos
 - monitores



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Soluções baseadas em software

- ◆ Vamos analisar a evolução de algumas tentativas para resolver o problema (escrita do código da SE e SS).
- ◆ Admite-se que
 - Os processos podem partilhar algumas variáveis globais para sincronizar as suas acções.
 - A operação de leitura (ou de escrita) da memória é atómica.
- ◆ Consideraremos primeiro o caso de 2 processos
 - O algoritmo 1 e o algoritmo 2 são incorrectos.
 - O algoritmo 3 (algoritmo de Peterson) é correcto.
- ◆ Apresentaremos uma solução mais geral, para n processos
 - O algoritmo da padaria (*bakery algorithm*)

Notação:

- Começamos com 2 processos, P0 e P1.
- Ao falar do processo Pi, Pj representa sempre o outro processo ($i \neq j$)



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Algoritmo 1

Variáveis partilhadas:

Turn: 0..1; { Turn = i significa que o proc. Pi pode entrar na SC
caso contrário não pode - Valor inicial: 0 ou 1 }

Process P0;

```
...
Repeat
  While Turn <> 0 do;
    { SecçãoCrítica }
    Turn:=1;
    { SecçãoRestante };
  Forever;
...
```

Process P1;

```
...
Repeat
  While Turn <> 1 do;
    { SecçãoCrítica }
    Turn:=0;
    { SecçãoRestante }
  Forever;
...
```

Análise do algoritmo:

- ◆ Garante a exclusão mútua das secções críticas:
 - Só um processo pode estar na sua secção crítica;
Pi está em espera activa (*busy waiting*) se Pj estiver na SC.
- ◆ Não garante o progresso:
 - A execução das secções críticas é feita de forma estritamente alternada
 - Se Pi quiser entrar 2 vezes consecutivas na secção crítica não pode.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Algoritmo 2

Variáveis partilhadas:

Flag: Array[0..1] of Boolean; { Flag[i]=True significa que
o proc. Pi está pronto a entrar na SC }

Inicialmente

Flag[0] = Flag[1] = false;

Process Pi;

...

```
Flag[i] = True;      { Pi está pronto a entrar na SC}
While Flag[j] do;    { Se Pj tb. estiver pronto, espera}
{ SecçãoCrítica }
Flag[i] = False;     { Permite que P0 entre}
...
```

Análise do algoritmo:

◆ Garante a exclusão mútua das secções críticas.

◆ Não garante o progresso:

■ Se for executada a sequência

- t0 : Flag[0] = true;
- t1 : Flag[1] = true;

ambos os processos ficarão eternamente à espera de entrar na SC
(situação de *deadlock*)



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Algoritmo 3 (algoritmo de Peterson)

Variáveis partilhadas:

Turn: 0..1;

Flag: Array[0..1] of Boolean; // Flag[i]=True significa que
// o proc. Pi está pronto a entrar na SC

Inicialmente

Turn = qualquer valor, 0 ou 1;
Flag[0] = Flag[1] = False;

Process Pi;

...

```
Flag[i] := True;      // Pi está pronto a entrar na SC
Turn := j;            // mas dá a vez a Pj, se ele precisar
While Flag[j] And (Turn=j) do;
{ SecçãoCrítica }
Flag[i] = False;      //Permite que Pj entre
...
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Algoritmo 3

(algoritmo de Peterson)

Análise do algoritmo:

- ◆ Garante a exclusão mútua das secções críticas
 - Um processo (ex: P1) só entra na SC se
 - o outro não quiser entrar ($Flag[0]=False$)
 - ou se for a sua vez ($Turn=1$)
 - Mesmo que os processos executem a sequência
 - $t0 : Flag[0] := True;$
 - $t1 : Flag[1] := True;$
 só um deles poderá entrar porque $turn$ só pode tomar um valor, 0 ou 1.
- ◆ Garante o progresso e uma espera limitada
 - P1 entrará na sua secção crítica (progresso) depois de, no máximo, uma entrada de P0 (espera limitada)

A implementação deste algoritmo para mais de 2 processos é complicada.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Algoritmo da padaria

(*bakery algorithm, Lamport*)

- ◆ Solução para n processos.
- ◆ Antes de entrar na secção crítica, cada processo recebe um *ticket* com um número (como nas padarias, ...)
- ◆ Entra na SC o processo que tiver o número mais pequeno.
- ◆ Se vários processos receberem o mesmo número usa-se o identificador do processo para desempatar.

Notação:

- $n = n^o$ de processos
- $(a,b) < (c,d)$ se $(a < c)$ ou $((a=c) \text{ e } (b < d))$
 - $a, c = n^o$ do *ticket*
 - $b, d =$ identificador do processo
- $\max(a_0, \dots, a_{n-1})$ é um número k (n^o do *ticket*), tal que $k \geq a_i$ ($a_i = n^o$ do *ticket* de qq. um dos outros), para $i = 0 \dots n-1$



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Algoritmo da padaria

Variáveis partilhadas:

choosing: Array[0..n-1] of Boolean;
number : Array[0..n-1] of Integer;

Inicialmente

choosing[i] = false, para i=0..n-1;
number[i] = 0, para i=0..n-1;

Bloco de entrada na SC

```
choosing[i]:=true;
number[i]:= max(number[0],..., number[n-1])+1;
choosing[i]:=false;
For j:=0 to n-1 do
  Begin
    While choosing[j] do;
    While (number[j]<>0) And
      ((number[j],j)<(number[i],i)) do;
  End;
```

- Anuncia intenção de tirar *ticket*
- Tira o *ticket*
- Anuncia que já tirou o *ticket*
- Espera que outros acabem de tirar o *ticket*
- Espera, se alguém está a executar a SC. Esse alguém deve
 - ter *ticket* → number[j]<>0
 - e ter direito a acesso antes de Pi → (number[j],j)<(number[i],i)

Bloco de saída da SC

number[i]:=0; • Deita fora o *ticket*



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Limitações das soluções por *software*

Características das soluções algorítmicas apresentadas:

- São da competência do programador.
- São complexas (principalmente, para mais do que 2 processos).
- Requerem espera activa (*busy waiting*)
 - os processos que estão a pedir para entrar na sua secção crítica estão a consumir tempo do processador, desnecessariamente.
 - Se as SCs forem demoradas seria mais eficiente bloquear os processos que estão à espera.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Soluções baseadas em *hardware*

◆ Sistemas uniprocessador

- Os processos concorrentes não podem ter execução sobreposta no tempo, mas apenas interlaçada.
- Para garantir a exclusão mútua bastaria inibir as interrupções, impedindo assim qualquer processo de ser interrompido.
 - perigoso permitir que os processos do utilizador o façam

```
Process Pi;
...
Disable_Interrupts;
SecçãoCrítica;
Enable_Interrupts;
...
```

◆ Sistemas multiprocessador

- A inibição de interrupções não garante a exclusão mútua.
- São necessárias instruções especiais que permitam testar e modificar uma posição de memória num único passo (sem interrupção), mesmo com vários *CPUs*.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

A instrução *Test-and-Set*

◆ Equivalente numa linguagem de alto nível

```
Function Test_and_Set(Var Target: Boolean): Boolean;
Begin
  Test_and_Set := Target;
  Target := True;
End;
```

◆ Executada atomicamente (sem interrupção)

◆ Implementação da exclusão mútua usando *Test_and_Set*:

Variáveis partilhadas:

```
Lock: Boolean; // Valor inicial: false
}
```

```
Process Pi;
...
While Test_and_Set(Lock) do;
  SecçãoCrítica;
  Lock:=false;
...
```

Não satisfaz a condição de espera limitada. Quando um processo deixa a sua SC e há mais do que um processo à espera, a selecção do processo que entra a seguir é arbitrária. Por isso, um processo pode ficar indefinidamente à espera (inanição).



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

A instrução Swap

- ◆ Permite trocar entre si o valor de 2 variáveis atómicamente.
- ◆ Implementação da exclusão mútua usando *Swap*:

Variáveis partilhadas:

Lock: Boolean; { Valor inicial: False }

```
Process Pi;  
...  
Key:=True;  
Repeat Swap(Lock,Key) Until Key=False;  
  {SecçãoCrítica };  
Lock:=False;  
...
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Spinlocks

- ◆ É um mecanismo de sincronização em que um processo / *thread* espera num ciclo (*spins*), testando se o *lock* (ferrolho) está disponível.
- ◆ É uma solução do tipo *busy waiting* para o problema da exclusão mútua.
- ◆ As operações sobre *spinlocks* são: InitLock, Lock e UnLock.

Type Lock = Boolean;

InitLock(L: Lock)

Lock := False;

Lock(L: Lock)

While TestAndSet(L) do;

UnLock(L: Lock)

Lock := False;

Secção crítica:

```
...  
Lock(Mutex);  
  SecçãoCrítica;  
Unlock(Mutex);  
...
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Características das soluções por *hardware*

- + Podem ser usadas com múltiplas secções críticas, cada secção crítica controlada por uma variável.
- + São aplicáveis a qualquer número de processos em sistemas uniprocessador ou multiprocessador
- Requerem suporte de hardware (instruções-máquina especiais).
- Usam espera activa (*busy waiting*)
 - Um processo à espera de entrar numa SC consome tempo de *CPU*.
- Se não forem tomadas precauções (v. pág. seguinte) é possível a inanição dos processos
 - Quando um processo deixa uma SC e há mais do que um processo à espera a selecção do processo que entra a seguir é arbitrária



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Solução baseada em *hardware* com espera limitada

```
Type Lock      = Boolean;
Waiting = Array[N] of Boolean;
```

InitLock(L: Lock)

```
Lock := False;
Waiting[1..N] := False;
```

Lock(L: Lock)

```
Waiting[i] := True;
Key := True;
while (Waiting[i] And Key)
  Key := TestAndSet(Lock);
Waiting[i] := False;
```

UnLock(L: Lock)

```
j := (i+1) Mod N;
while ((j <> i) And (Not Waiting[j]))
  j := (j+1) Mod N;
if (j=i)
  Lock := False;
else
  Waiting[j] := False;
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Semáforos (Dijkstra,1965)

◆ Semáforo

- mecanismo de sincronização (fornecido pelo S.O.) que não requer espera activa.

◆ Semáforo S

- Variável inteira S inicializada com um valor não-negativo (≥ 0)
- Depois de inicializada só pode ser actualizada através de duas operações atómicas:
 - `wait(S)` ou `P(S) : S:=S-1;`
If $S < 0$ then `Block(S)`;
 - `signal(S)` ou `V(S) : S:=S+1;`
If $S \leq 0$ Then `WakeUp(S)`;
- `Block(S)` - o processo que a invocou é bloqueado
- `WakeUp(S)` - um processo que invocou anteriormente `Block(S)` fica pronto a correr
- Para evitar espera activa, quando um processo tem de esperar é colocado numa fila de processos bloqueados no semáforo.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Semáforos

◆ De facto, um semáforo é um registo (*record* ou *structure*):

```

Type Semaphore = Record
    Count: Integer;
    Queue: List_of_Process
End;

```

```
Var S: Semaphore;
```

◆ E as operações sobre semáforos são:

- `Wait(S) :`
`S.Count:=S.Count-1;`
 If $S.Count < 0$ then
 Begin
 Colocar este processo em S.Queue;
 Bloquear este processo
 End;
- `Signal(S):`
`S.Count:=S.Count+1;`
 If $S.Count \leq 0$ then
 Begin
 Remover um processo, P, de S.Queue;
 Colocar P na fila de proc.s prontos
 End;
- S.Count pode ser inicializada com um valor não negativo.
- Quando $S.Count \geq 0$, S.Count representa o nº de proc.s que podem executar `Wait(S)` sem bloquear.
- Quando $S.Count < 0$, $|S.Count|$ representa o nº de proc.s que estão à espera no semáforo.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Implementação dos semáforos

- ◆ A implementação dos semáforos introduz implicitamente uma secção crítica:
 - O incremento e decremento da variável introduz uma secção crítica, bem como,
 - a alteração do valor da variável seguida de um teste do seu valor na instrução seguinte.
- ◆ `Wait()` e `Signal()` têm de ser operações atómicas.
- ◆ Como implementar a secção crítica interna ?
 - Sistema uniprocessador
 - Inibir as interrupções durante a execução de `Wait()` e `Signal()` ...
 - Sistemas multiprocessador
 - Usar uma das "soluções por *software*", anteriormente analisadas.
 - Notar que não nos livramos da espera activa, mas ela fica limitada às operações `Wait()` e `Signal()` que são curtas.
 - Usar uma das "soluções por *hardware*", anteriormente analisadas, se disponíveis (ex: `Test_And_Set` ou `Swap`)



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Utilização dos semáforos para resolver problemas de secções críticas

- ◆ Variável partilhada pelos N processos:

```
Var Mutex: Semaphore;
```
- ◆ Valor inicial

```
Mutex.Count:= 1;
```

(só 1 processo consegue entrar na SC - exclusão mútua)
- ◆ Um processo que não consiga entrar imediatamente na SC bloqueia e cede o processador.
- ◆ Os processos bloqueados retomam a sua execução, à medida que os outros processos forem saindo das SC's correspondentes.
- ◆ Um semáforo que é inicializado com o valor 1 e é usado por 2 ou mais processos para assegurar que só um deles consegue executar uma SC ao mesmo tempo é conhecido por semáforo binário ou *mutex*.

```
Process Pi;  
...  
Wait(Mutex);  
Secção_Crítica;  
Signal(Mutex);  
...
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Problemas comuns c/ a utilização de semáforos *Mutex*

- ◆ Inicializar o semáforo com o valor 0, em vez de 1
 - ⇒ nenhum processo consegue executar a secção crítica
- ◆ Trocar as operações P e V (Wait e Signal não se trocam tão facilmente)
 - ⇒ não há exclusão mútua
- ◆ "Aninhamento" inadequado de semáforos *Mutex*

■ Ex:
O que acontece
se a ordem
de execução
for A, C, B, D ?

R: *deadlock*

A
B
Process P1;
...
Wait (Mutex1);
...
Wait (Mutex2);
Secção_Crítica;
Signal (Mutex1);
...
Signal (Mutex2);
...

C
D
Process P2;
...
Wait (Mutex2);
...
Wait (Mutex1);
Secção_Crítica;
Signal (Mutex2);
...
Signal (Mutex1);
...



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Utilização dos semáforos para resolver problemas de sincronização

- ◆ Os semáforos que são inicializados com `S.Count=0` fornecem um mecanismo p/ sincronizar a execução de 2 processos.

Ex:
A instrução S1 do proc. P1
tem de ser executada
antes da instrução S2 do proc. P2

Sync.Count:=0; // Inicialmente

Process P1;
...
S1;
Signal (Sync);
...

Process P2;
...
Wait (Sync);
S2;
...



Ex:
Dois processos necessitam de
esperar um pelo outro
num determinado ponto

Sync1.Count:=0; //Inicialmente
Sync2.Count:=0;

Process P1;
...
Signal (Sync1);
Wait (Sync2);
...

Process P2;
...
Signal (Sync2);
Wait (Sync1);
...



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Problemas com os semáforos

- ◆ Os semáforos constituem um mecanismo poderoso para garantir a exclusão mútua e a sincronização de processos.
- ◆ Contudo, é fácil cometer erros na sua utilização.
Quando as operações `wait()` e `signal()` estão espalhadas por vários processos, pode ser difícil compreender os seus efeitos.
- ◆ A sua utilização tem de ser correcta em todos os processos.
- ◆ Um processo mal escrito (ou "mal intencionado") pode contribuir para a falha de um conjunto de processos.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Problemas clássicos de sincronização

Problema do Produtor/Consumidor

Enunciado

- ◆ Um processo produtor produz informação que é consumida por um processo consumidor.
 - ex: um programa de impressão produz caracteres que são consumidos por um *driver* de impressora
- ◆ Existe um *buffer* de itens que pode ser preenchido pelo produtor e esvaziado pelo consumidor.
- ◆ O processo produtor pode produzir um item enquanto o processo consumidor consome outro item.
- ◆ O produtor e o consumidor devem ser sincronizados de modo a que o consumidor não tente consumir um item ainda não produzido.
- ◆ Sendo o tamanho do *buffer* limitado, o produtor não pode acrescentar novos itens se o *buffer* estiver cheio.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Problema do Produtor/Consumidor

```

Var
  Buffer = ...
  Full, Empty, Mutex: Semaphore;

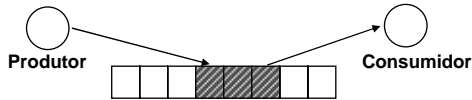
Inicialização:
  Full.Count:= 0;
  Empty.Count:= N;
  Mutex.Count:=1;
  
```

```

Process Producer;
...
Repeat
...
Produce(Item);
Wait(Empty);
Wait(Mutex);
Append(Item);
Signal(Mutex);
Signal(Full);
...
Until ...;
  
```

```

Process Consumer;
...
Repeat
...
Wait(Full);
Wait(Mutex);
Item=Take();
Signal(Mutex);
Signal(Empty);
Consume(Item);
...
Until ...;
  
```



- ◆ Full
 - p/ sincronizar os 2 processos;
 - não significa *buffer* cheio mas que tem pelo menos 1 item.
- ◆ Empty
 - p/ sincronizar os 2 processos;
 - não significa *buffer* vazio mas que se esvaziou um elem.to
- ◆ Mutex
 - p/implementar a exclusão mútua.



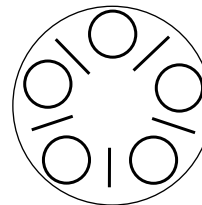
FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Problema dos filósofos a jantar

◆ Enunciado:

- 5 filósofos estão sentados a uma mesa;
- os filósofos passam a vida a pensar e a comer (arroz ...?!);
- cada um precisa de 2 "pausinhos" para comer;
- só há 5 "pausinhos" ...!!!
- só pode pegar num pausinho de cada vez (e não pode roubar um do vizinho !)



- ◆ Problema clássico de sincronização.
- ◆ Ilustra a dificuldade de alocar recursos entre processos sem provocar impasse / bloqueamento fatal (*deadlock*) ou inanição (*starvation*).



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Problema dos filósofos a jantar

1ª tentativa de solução

- ◆ Cada filósofo é um processo.
- ◆ Um semáforo por "pausinho":
 - Fork: Array[0..4] of Semaphore;
- ◆ Conduz a *deadlock* se, por exemplo, cada filósofo começar por pegar no "pausinho" à sua esquerda (/ direita).

Inicialização:

```
For i:=0 to 4 do  
  Fork[i].Count:=1;
```

```
Process Pi;  
Repeat  
  Think;  
  Wait(Fork[i]);  
  Wait(Fork[(i+1) Mod 5]);  
  Eat;  
  Signal(Fork[(i+1) Mod 5]);  
  Signal(Fork[i]);  
Until ...;
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Problema dos filósofos a jantar

2ª tentativa de solução

- ◆ Depois de pegar no "pausinho" à sua esquerda, por exemplo, vê se o "pausinho" da direita está livre. Se estiver pausa o da esquerda.
- ◆ Problema:
 - Todos pegam no "pausinho" da esquerda simultaneamente.
 - Ao verem que o "pausinho" da direita está ocupado pousam todos os da esquerda !!!
 - Conduz a inanição (*starvation*).



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Problema dos filósofos a jantar

Uma solução

- ◆ Admitir que só 4 filósofos tentam comer simultaneamente.
- ◆ Usar um outro semáforo M que limita a 4 o número de filósofos que podem tentar comer.

Inicialização:

$M.Count := 4;$

- ◆ Então 1 filósofo pode sempre estar a comer enquanto os outros 3 seguram 1 "pausinho".

Quando aquele terminar, um dos outros pode comer.

Inicialização:

$M.Count := 4;$

```
Process Pi;
Repeat
  Think;
  Wait(M);
  Wait(Fork[i]);
  Wait(Fork[(i+1) Mod 5]);
  Eat;
  Signal(Fork[(i+1) Mod 5]);
  Signal(Fork[i]);
  Signal(M);
Until ...;
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Problema dos filósofos a jantar

Outra solução ("melhor que a anterior, pois garante máximo paralelismo", Tanenbaum)

```
#define N 5 /* Número de filósofos */
#define RIGHT(i) (((i)+1) % N)
#define LEFT(i) (((i)==0) ? (N-1) : (i)-1)
#define THINKING 0
#define HUNGRY 1
#define EATING 2

semaphore mutex = 1;
semaphore s[N]; /* inicializados com zero */
```

```
void take_forks(int i) {
  wait(&mutex);
  state[i] = HUNGRY;
  test(i);
  signal(&mutex);
  wait(&s[i]);
}
```

2

```
void put_forks(int i) {
  wait(&mutex);
  state[i] = THINKING;
  test(LEFT(i));
  test(RIGHT(i));
  signal(&mutex);
}
```

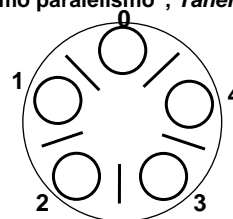
3

```
void test(int i) {
  if ( state[i] == HUNGRY &&
      state[LEFT(i)] != EATING &&
      state[RIGHT(i)] != EATING ) {
    state[i] = EATING;
    signal(&s[i]);
  }
}
```

4

```
void philosopher(int i) {
  while(.....) {
    think();
    take_forks(i); /*obtem 2 pausinhos ou bloqueia*/
    eat();
    put_forks(i);
  }
}
```

1



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Problema dos Leitores/Escritores

◆ O problema

- leitores e escritores acedem a informação comum
- leitores - apenas leem a informação
- escritores - modificam a informação

◆ Solução 1 - os leitores têm prioridade (mais simples)

- Enquanto um escritor estiver a aceder à informação nenhum outro escritor ou leitor pode aceder.
- Quando um leitor estiver a aceder à informação outros leitores que entretanto cheguem podem aceder livremente.

◆ Solução 2 - os escritores têm prioridade

- Impedir qualquer leitor de aceder à informação sempre que haja algum escritor à espera de a actualizar.
- Quando o leitor/escritor actual terminar o acesso um escritor que esteja à espera tem prioridade sobre outros leitores.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Solução 1 - Prioridade aos leitores

```

Program ReadersWriters;
Var ReadCount: Integer;
    X, WSem: Semaphore (:=1);

Procedure Reader;
Begin
  Repeat
    Wait(X);
    ReadCount:=ReadCount+1;
    If ReadCount=1 Then Wait(WSem);
    Signal(X);
    READUNIT;
    Wait(X);
    ReadCount:=ReadCount-1;
    If ReadCount=0 Then Signal(WSem);
    Signal(X)
  Forever
End;

Begin
  ReadCount:=0;
  ParBegin
    Reader;
    Writer;
  ParEnd
End.

```

```

Procedure Writer;
Begin
  Repeat
    Wait(WSem);
    WRITEUNIT;
    Signal(WSem);
  Forever
End;

```

WSem - garante a exclusão mútua no acesso à informação partilhada; desde que um escritor esteja a aceder aos dados nenhum outro escritor ou leitor pode aceder; leitores ou escritores que cheguem entretanto têm de esperar em WSem.

X - garante que a actualização de ReadCount é feita correctamente

ReadCount - para tomar nota do número de leitores; desde que haja pelo menos um leitor os leitores que cheguem entretanto não têm de esperar.



FEUP

Faculdade de Engenharia da Universidade do Porto

Solução 2 - Prioridade aos escritores

```
Program ReadersWriters;
Var ReadCount, WriteCount: Integer;
    X, Y, Z, WSem, RSem: Semaphore (:=1);
```

```
Procedure Reader;
Begin
  Repeat
    Wait(Z);
    Wait(RSem);
    Wait(X);
    ReadCount:=ReadCount+1;
    If ReadCount = 1 Then Wait(WSem);
    Signal(X);
    Signal(RSem);
    Signal(Z);
    READUNIT;
    Wait(X);
    ReadCount:=ReadCount-1;
    If ReadCount=0 Then Signal(WSem);
    Signal(X);
  ForEver
End;
```

```
Procedure Writer;
Begin
  Repeat
    Wait(Y);
    WriteCount:=WriteCount+1;
    If WriteCount=1 Then Wait(RSem);
    Signal(Y);
    Wait(WSem);
    WRITEUNIT;
    Signal(WSem);
    Wait(Y);
    WriteCount:=WriteCount-1;
    If WriteCount=0 Then Signal(RSem);
    Signal(Y);
  ForEver
End;
```

```
Begin
  ReadCount:=0; WriteCount:=0;
  ParBegin
    Reader;
    Writer;
  ParEnd
End.
```

Além dos semáforos e variáveis anteriores temos:

- RSem - impede o acesso dos leitores enquanto houver pelo menos um escritor a querer aceder à informação partilhada
- Y - garante que a actualização de WriteCount é feita correctamente
- WriteCount - controla o Signal a Rsem
- Z - só um leitor pode fazer fila em RSem os outros fazem fila em Z

Faculdade de Engenharia da Universidade do Porto

```
Procedure Reader;
...
  Wait(Z);
  Wait(RSem);
  Wait(X);
  ReadCount:=ReadCount+1;
  If ReadCount = 1 Then Wait(WSem);
  Signal(X);
  Signal(RSem);
  Signal(Z);
  READUNIT;
  Wait(X);
  ReadCount:=ReadCount-1;
  If ReadCount=0 Then Signal(WSem);
  Signal(X);
...
```

```
Procedure Writer;
...
  Wait(Y);
  WriteCount:=WriteCount+1;
  If WriteCount=1 Then Wait(RSem);
  Signal(Y);
  Wait(WSem);
  WRITEUNIT;
  Signal(WSem);
  Wait(Y);
  WriteCount:=WriteCount-1;
  If WriteCount=0 Then Signal(RSem);
  Signal(Y);
...
```

Estado das filas dos semáforos:

Só leitores no sistema WSem activado
Não existem filas

Só escritores no sistema WSem e RSem activados
Os escritores fazem fila em WSem

Leitores e escritores, com leitor a aceder em 1º lugar WSem activado pelo leitor
RSem activado pelo escritor
Todos os escritores fazem fila em WSem
Um leitor faz fila em RSem
Outros leitores fazem fila em Z

Leitores e escritores, com escritor a aceder em 1º lugar WSem activado pelo escritor
RSem activado pelo escritor
Os escritores fazem fila em WSem
Um leitor faz fila em RSem
Outros leitores fazem fila em Z

Faculdade de Engenharia da Universidade do Porto

Construções de alto nível p/ exclusão mútua e sincronização

- ◆ Monitores
- ◆ Regiões críticas
- ◆ Passagem de mensagens



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Monitores

- ◆ Monitor
 - módulo de *software* constituído por
 - 1 ou mais procedimentos
 - 1 secção de inicialização
 - dados locais (escondidos)
- ◆ O "mundo exterior" só "vê" os procedimentos.
- ◆ Os dados locais só podem ser manipulados no interior dos procedimentos.
- ◆ A entrada no monitor faz-se através de uma chamada a um procedimento.
- ◆ Só um processo pode estar a executar no monitor de cada vez.
- ◆ Deste modo os monitores permitem implementar facilmente a exclusão mútua.
- ◆ As variáveis de tipo condição (*condition variables*) permitem a sincronização.



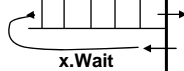
FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

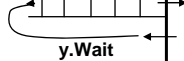
Monitores

x, y: condition variables
(podem ser usadas c/ 2 operações
pré-definidas: wait e signal)

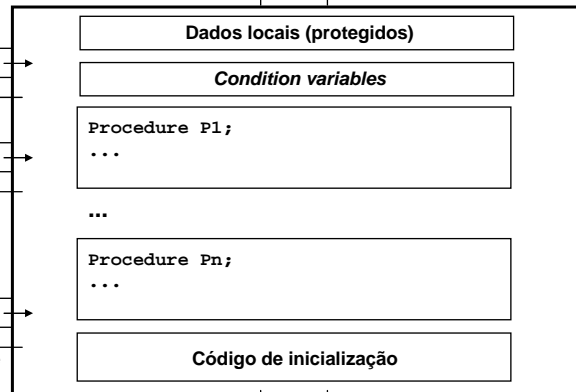
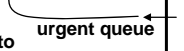
processos que
executaram
x.Wait



processos que
executaram
y.Wait



processos que
executaram
Signal
a meio de
um procedimento



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo (Problema do produtor/consumidor)

```

Monitor Bounded_Buffer;
Var
  Buffer: Array[0..N-1] of Char;
  NextIn, NextOut, Count: Integer;
  NotFull, NotEmpty: Condition;

Procedure Append (X: Char);
Begin
  If Count=N then NotFull.Wait;
  Buffer[NextIn]:=X;
  NextIn:=(NextIn+1) Mod N;
  Count:=Count+1;
  NotEmpty.Signal;
End;

Procedure Take (X: Char);
Begin
  If Count=0 then NotEmpty.Wait;
  X:=Buffer[NextOut];
  NextOut:=(NextOut+1) Mod N;
  Count:=Count-1;
  NotFull.Signal;
End;

Begin {Monitor initialization}
  NextIn:=0; NextOut:=0; Count:=0;
End;

```

```

(* Programa que usa o monitor *)
...
Procedure Producer;
Var
  X: Char;
Begin
  Repeat
    Produce(X);
    Append(X);
  Until ...
End;

Procedure Consumer;
Var
  X: Char;
Begin
  Repeat
    Take(X);
    Consume(X);
  Until ...
End;

Begin
  ParBegin
    Producer; Consumer;
  ParEnd;
End.

```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Monitores

- ◆ Tal como acontece com os semáforos
é possível cometer erros de sincronização com os monitores.
 - Ex: omitir `NotFull.Signal`, no exemplo anterior
- ◆ A vantagem que os monitores têm sobre os semáforos é que
todas as funções de sincronização ficam confinadas ao interior do monitor
 - mais fácil detectar e corrigir os erros de sincronização
- ◆ Os monitores podem ser implementados recorrendo a semáforos
e vice-versa.
- ◆ Algumas linguagens de programação suportam monitores
 - ex: Java (<http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/monitors.html>)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Regiões críticas

- ◆ Uma região crítica protege uma estrutura de dados partilhada.
O compilador encarrega-se de gerar código que
garante a exclusão mútua no acesso aos dados.
- ◆ Requer uma variável `v`, de tipo `T`, declarada como segue:
 - `var V: Shared T; {ex: var I: Shared Integer;}`
- ◆ A variável `v` só pode ser acedida dentro de uma instrução do tipo:
 - `Region V When B do S;`
onde `B` é uma expressão booleana e
`S` é uma instrução (simples ou composta);
- ◆ Enquanto `S` estiver a ser executada, nenhum outro processo
pode executar esta ou outra região "guardada" pela variável `v`.
- ◆ Quando um processo executar a instrução `Region`, a expressão Booleana `B` é avaliada.
 - Se `B` for `True`, a instrução `S` é executada.
 - Se `B` for `False`, o processo é retardado até que
(`B` seja `True`) e (nenhum outro processo esteja a executar numa região associada a `V`).



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo (Problema do produtor/consumidor)

```

Var
  Buffer = Shared Record
    Pool: Array[0..N-1] of Item;
    Count, In, Out: Integer;
End;

```

```

Process Producer;
...
{Insere ItemP no buffer partilhado}
Region Buffer When Count<N do
  Begin
    Pool[In] := ItemP;
    In := (In+1) Mod N;
    Count := Count+1;
  End;
...

```

```

Process Consumer;
...
{Remove ItemC no buffer partilhado}
Region Buffer When Count>0 do
  Begin
    ItemC := Pool[Out];
    Out := (Out+1) Mod N;
    Count := Count-1;
  End;
...

```



Pascal-FC (linguagem p/o ensino de programação concorrente)
suporta *conditional critical regions*

MIEIC
Faculdade de Engenharia da Universidade do Porto

Passagem de mensagens

◆ Semáforos e monitores

- resolvem o problema da exclusão mútua em sistemas com 1 ou mais CPUs que tenham acesso a uma memória comum
- não podem ser usados em sistemas distribuídos

◆ Semáforos

- são construções de mais baixo nível

◆ Monitores

- só estão disponíveis em algumas linguagens

◆ Passagem de mensagens

- pode ser usada em sistemas c/ memória partilhada (uniprocessador ou multiprocessador), bem como em sistemas distribuídos



MIEIC
Faculdade de Engenharia da Universidade do Porto

Passagem de mensagens

- ◆ Os sistemas operativos implementam geralmente um sistema de mensagens que permite que os processos
 - comuniquem
 - sincronizem as suas acções
- ◆ Há pelo menos 2 operações que devem ser suportadas:
 - `send(destination,message)`
 - `receive(source,message)`
- ◆ Depois de executar `Send()`/`Receive()` os processos podem bloquear ou não.
- ◆ **Sender (transmissor)**
o mais natural é não bloquear após executar `Send()`.
- ◆ **Receiver (receptor)**
o mais natural é bloquear após executar `Receive()`.
- ◆ Por vezes, existem outras possibilidades
 - Ex: `Send()` c/bloqueio e `Receive()` c/bloqueio



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo (resolução de problemas de exclusão mútua)

- ◆ Criar uma *mailbox* (ex. *Mutex*) partilhada por N processos.
- ◆ `Send()` não bloqueia.
- ◆ `Receive()` bloqueia quando *Mutex* estiver vazia.
- ◆ Inicialização:
`Send(Mutex,Anything)`
- ◆ O 1º processo que executar `Receive()` entra na secção crítica.
Os outros ficam bloqueados até que ele reenvie a mensagem.

```
Process Pi;  
  
Var  
  Msg: Message;  
  ...  
  Repeat  
    Receive(Mutex,Msg);  
    {Secção Crítica}  
    Send(Mutex,Msg);  
    ...  
  Until ...  
  ...
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo (Problema do produtor/consumidor)

Usa-se 2 *mailboxes*
capacidade igual a Capacity

MayConsume

- contém os itens

MayProduce

- contém mensagens nulas

```
...
Begin
  Create_Mailbox(MayProduce);
  Create_Mailbox(MayConsume);
  For I:=1 to Capacity do
    Send(MayProduce,Null);
  ParBegin
    Producer; Consumer
  ParEnd
End.
```

```
...
Procedure Producer;
Var
  PMsg: Message;
Begin
  Repeat
    Receive(MayProduce,PMsg);
    PMsg:=produce();
    Send(MayConsume,PMsg);
  Until ...
End;
```

```
Procedure Consumer;
Var
  CMsg: Message;
Begin
  Repeat
    Receive(MayConsume,CMsg);
    Consume(CMsg);
    Send(MayProduce,Null);
  Until ...
End;
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto