

# Introdução ao Sincronismo

- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, *Operating System Concepts, 6<sup>a</sup> Ed.*, John Wiley & Sons Inc., 2002 [cap. 7]

# Exercício de tarefas com dados partilhados

- Criar dois fios de execução concorrentes para incrementar o valor de uma variável.
- Cada fio de execução (tarefa) vai realizar 50 incrementos
- Com um funcionamento correcto será espectável que no fim da execução a variável partilhada tenha o valor 100.

```
int x;  
DWORD WINAPI IncFunc(LPVOID args)  
{  
    for(int i=0; i<50; i++)  
        x++;  
    return 0;  
}
```

```
mov    eax,dword ptr [x]  
add    eax,1  
mov    dword ptr [x],eax
```

```
DWORD WINAPI IncFunc(LPVOID args)  
{  
    int tmp;  
    for(int i=0; i<50; i++){  
        tmp = x;  
        tmp++;  
        x = tmp;  
    }  
    return 0;  
}
```



# Exercício

```
DWORD WINAPI IncFunc(LPVOID args)
{
    int tmp;
    for(int i=0; i<50; i++){
        tmp = x;
        tmp++;
        x = tmp;
    }
    return 0;
}
```

```
MyFirstThreads - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Community Help

MyFirstThreads.cpp
(Global Scope)

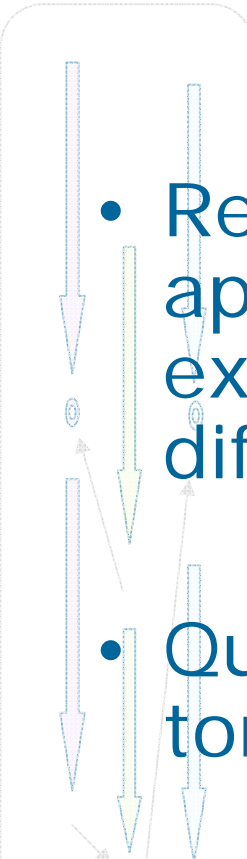
1 // MyFirstThreads.cpp : Defines the entry point for the console application.
2 //
3
4 #include "stdafx.h"
5
6 int x;
7
8 DWORD WINAPI IncFunc(LPVOID args)
9 {
10     int tmp;
11     for(int i=0; i<50; i++){
12         tmp = x;
13         tmp++;
14         x = tmp;
15     }
16     return 0;
17 }
18
19 int _tmain(int argc, _TCHAR* argv[])
20 {
21     HANDLE ht, ht2;
22     DWORD threadId, threadId2;
23     x = 0;
24
25     // Criar as duas tarefas
26     ht = CreateThread ( NULL, 0, IncFunc, NULL, NULL, &threadId);
27     ht2 = CreateThread ( NULL, 0, IncFunc, NULL, NULL, &threadId2);
28
29     //Esperar a terminação das tarefas
30     WaitForSingleObject(ht, INFINITE);
31     WaitForSingleObject(ht2, INFINITE);
32     _tprintf(TEXT("Terminei com o valor de x = %d\n"), x);
33     return 0;
34 }
35
36
```

Código das tarefas



# Exercício

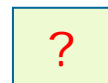
- Relativamente ao programa anterior, apresente alguns possíveis cenários de execução por forma a dar um resultado diferente do esperado ( $x = 100$ ).
- Qual o valor mínimo que a variável  $x$  pode tomar no final da execução do programa?



```
C:\WINDOWS\system32\cmd.exe
Terminei com o valor de x = 100
Press any key to continue . . .
```

```
C:\WINDOWS\system32\cmd.exe
Terminei com o valor de x = 50
Press any key to continue . . . _
```

...



# Acesso partilhado a um recurso

Várias *threads* a quererem aceder a um recurso

Controle de acesso ao recurso por uma *flag* de acesso

```
typedef enum {LIVRE, OCUPADO} TFLAG;  
TFLAG Token = LIVRE;
```

```
// esperar por token livre  
While (Token == OCUPADO)  
    // volta a testar  
    ;  
// token está livre  
// colocá-lo como ocupado  
Token=OCUPADO;  
// Aceder ao recurso partilhado  
Token=LIVRE; // fim de acesso
```

E se houver fim de *time slice* aqui ?



# Exemplo de acesso a uma variável partilhada

```
DWORD WINAPI IncFunc(LPVOID args)
{
    int tmp;
    for(int i=0; i<50; i++){
        while (recursoOcupado)
            ;
        recursoOcupado = true;
        /* Inicio da região critica */
        tmp = x;
        tmp++;
        x = tmp;
        /* Fim da região critica */
        recursoOcupado = false;
    }
    return 0;
}
```

Continua com problemas se houver fim de *time slice* aqui

C:\WINDOWS\system32\cmd.exe

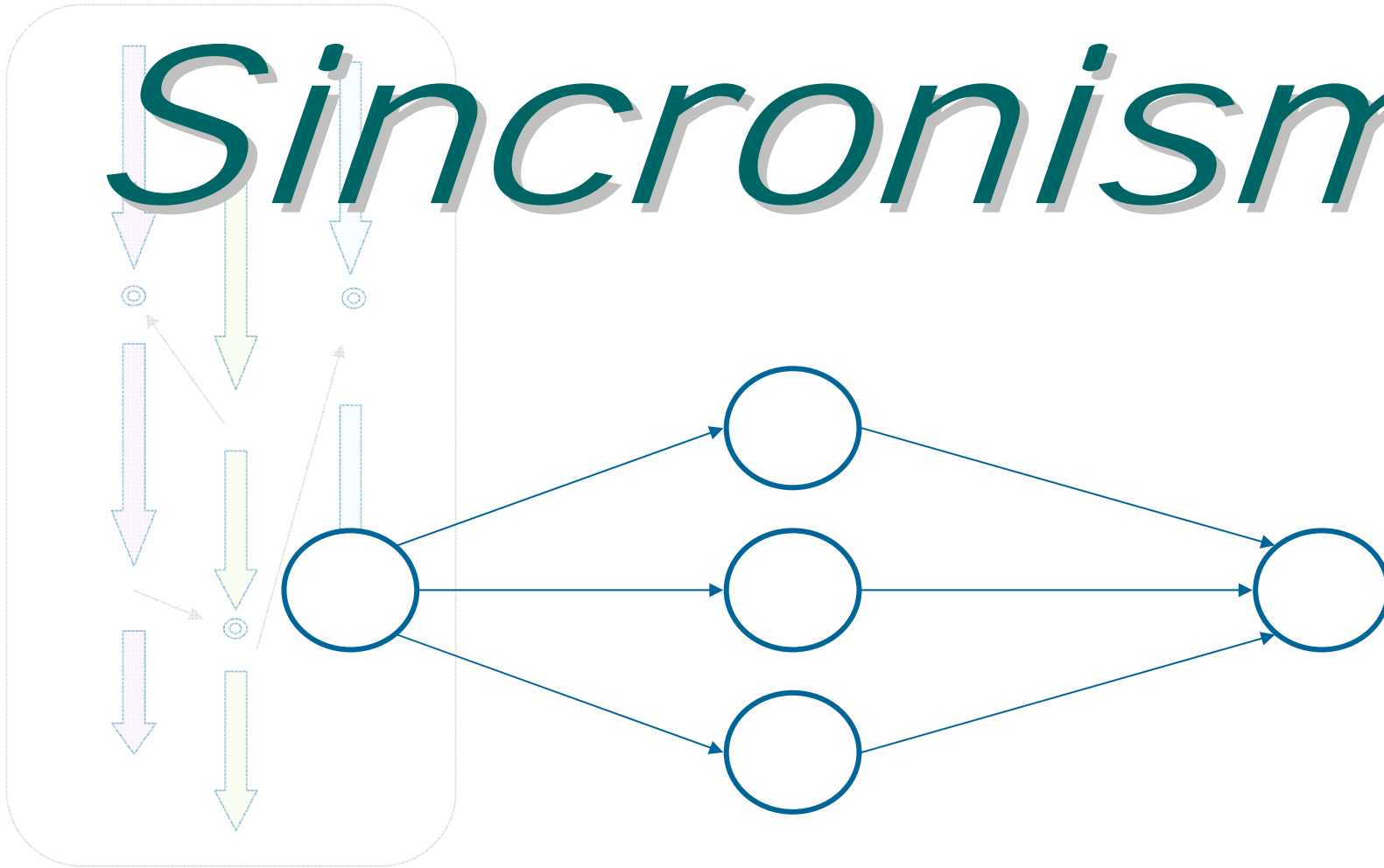
Terminei com o valor de x = 100  
Press any key to continue . . .

C:\WINDOWS\system32\cmd.exe

Terminei com o valor de x = 50  
Press any key to continue . . .



# *Sincronismo*



# Concorrência

Na terminologia de sistemas operativos, são designados de **sistemas concorrentes**, os sistemas com múltiplas actividades (por exemplo processos) paralelas.

**Um Sistema Operativo multiprocesso é, então, um ambiente de processos concorrentes.**

Os processos concorrem no acesso a:

- recursos internos ao SO  
exº: processador, blocos de memória
- recursos externos ao SO  
exº: impressoras, ficheiros





# Concorrência requer controle de acesso

- Supondo que 3 processos enviam os seus dados para: uma impressora, ou um canal série ou o écran:
  - P1: "O João tem febre"
  - P2: "A sopa tem letras"
  - P3: "Hoje está sol"
- Caso não haja qualquer controle no acesso ao dispositivo em questão, poder-se-ia receber:
  - "Hoje O João tem A febre sopa tem letras está sol"



# Controle de acesso por exclusão mútua

- Para que os anteriores processos possam enviar os seus dados dentro do esperado, eles têm que ter acesso exclusivo ao dispositivo. Então ter-se-ia:
  - P1: [Inic-excl] "O João tem febre" [fim-excl]
  - P2: [Inic-excl] "A sopa tem letras" [fim-excl]
  - P3: [Inic-excl] "Hoje está sol" [fim-excl]
- Apesar de não garantido qual seria a ordem de envio (poderia ser  $p1-p2-p3$ , ou  $p2-p3-p1$ , ou outra qualquer), o resultado não iria conter frases deturpadas.



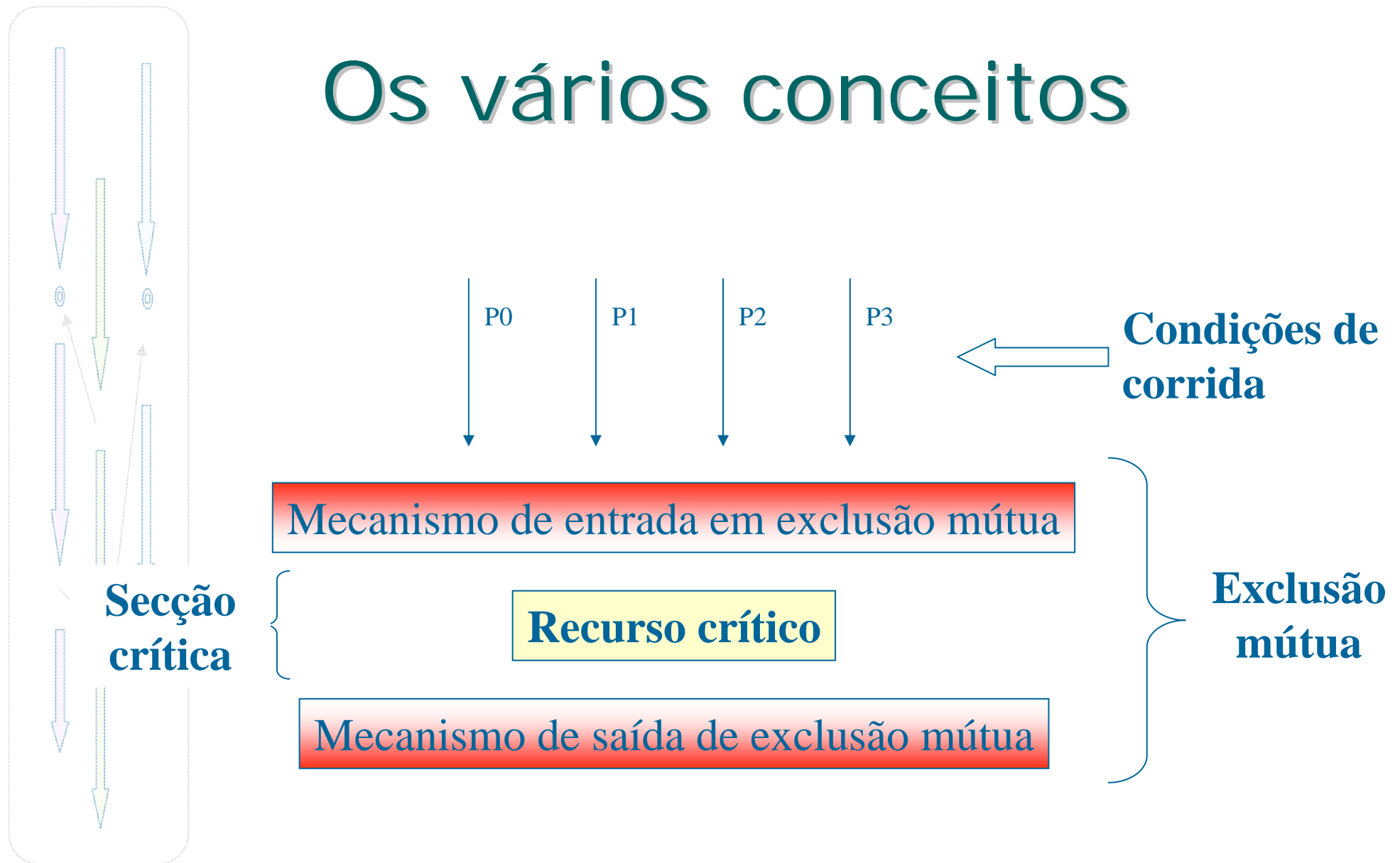
# Definições associadas à concorrência

- **Recurso crítico (*critical resource*)**
  - Um recurso crítico é um recurso que só tem um comportamento coerente quando é acedido (em modo exclusivo) por um só processo\*.
- **Condições de corrida (*race conditions*)**
  - Existem "*race conditions*" quando o resultado final do uso de um recurso partilhado, por dois ou mais processos\*, depende da ordem de execução
- **Exclusão mútua (*Mutual Exclusion*)**
  - Diz-se que uma zona de código é executada em exclusão mútua **quando é garantido o seu acesso exclusivo**, ou seja, não é permitido o acesso simultâneo por mais de um processo
- **Secção crítica (*critical section*)**
  - Uma secção crítica é uma zona de código que altera o estado de um recurso partilhado (e.g. variáveis). Esta secção deve ser executada em Exclusão Mútua

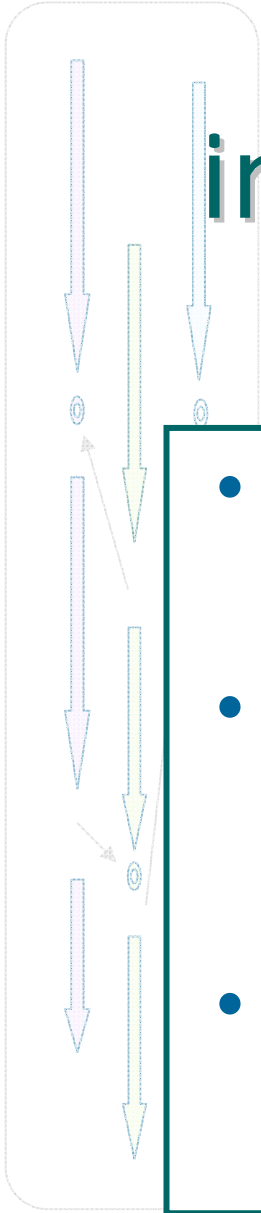
\* Processo ou *thread*



# Os vários conceitos



# Requisitos na implementação de exclusão mútua

- 
- **Exclusão mútua** : não podem aceder simultaneamente dois processos\* à secção crítica
  - **Decisão de Acesso** : a decisão de qual o processo\* que acede à secção crítica só deve ter em conta os processos\* que o tentam fazer
  - **Espera limitada** : o número de vezes que um processo\* é preterido no acesso à secção crítica deve ser limitado (impede o *starvation*)

\* Processo ou *thread*



# Exclusão mútua (*software*): v1

A título de exemplo considere-se que o recurso exclusivo é uma impressora

vez = 0;

```
/* Processo 0 */  
...  
while(vez != 0) ;  
/* Acesso ao recurso exclusivo */  
vez = 1;  
...  
}
```

```
/* Processo 1 */  
...  
while(vez != 1) ;  
/* Acesso ao recurso exclusivo */  
vez = 0;  
...  
}
```

Como os processos têm de repetidamente testar o valor das variáveis de controle, este tipo de espera designa-se de **espera activa**.

Problemas: **viola a decisão de acesso**, pois requer acesso estritamente alternado



# Exclusão mútua (*software*): v2

De modo a permitir que cada um entre quando o recurso estiver disponível

*“flags”* = 0;

```
/* Processo 0 */  
...  
while(flag[1]) ;  
flag[0] = true;  
/* Acesso ao recurso crítico */  
flag[0] = false;  
...  
}
```

```
/* Processo 1 */  
...  
while(flag[0]) ;  
flag[1] = true;  
/* Acesso ao recurso crítico */  
flag[1] = false;  
...  
}
```

Problemas: viola a exclusão mútua, pois ambos os processos podem testar as *flags* antes de as alterarem



# Exclusão mútua (*software*): v3

De modo a inibir o avanço do outro processo quando um processo testa já, anteriormente, indicou que quer entrar

```
/* Processo 0 */  
...  
    flag[0] = true;  
    while(flag[1]) ;  
    /* Acesso ao recurso crítico */  
    flag[0] = false;  
...  
}
```

```
/* Processo 1 */  
...  
    flag[1] = true;  
    while(flag[0]) ;  
    /* Acesso ao recurso crítico */  
    flag[1] = false;  
...  
}
```

Problemas: gera *deadlock*, pois ambos os processos podem indicar que querem entrar e depois testar as *flags*





# Exclusão mútua (*software*): v4

De modo a inibir o *deadlock*, cada processo recua no pedido até conseguir entrar

```
/* Processo 0 */
...
flag[0] = true;
while(flag[1]) {
    flag[0] = false;
    delay;
    flag[0] = true;
};
/* Acesso ao recurso crítico */
flag[0] = false;
...
}
```

```
/* Processo 1 */
...
flag[1] = true;
while(flag[0]) {
    flag[1] = false;
    delay;
    flag[1] = true;
};
/* Acesso ao recurso crítico */
flag[1] = false;
...
}
```

Problemas: não é garantido que os processos após recuarem que não se venham a encontrar na mesma situação. Nesta situação de aparente *deadlock*, designa-se de *livelock*. Pode-se atenuar com *delay(random)*



# Exclusão mútua (*software*): algoritmo de Dekker

```
flag[0] = 0;  
flag[1] = 0;  
vez = 1;
```

De modo a evitar o livelock, este algoritmo, nas condições de ambos quererem entrar, atribuí o acesso de forma alternada

```
/* Processo 0 */  
...  
flag[0] = true;  
while(flag[1])  
    if (vez == 1) {  
        flag[0] = false;  
        while (vez == 1) ;  
        flag[0] = true;  
    };  
/* Acesso ao recurso crítico */  
vez = 1;  
flag[0] = false;  
...  
}
```

```
/* Processo 1 */  
...  
flag[1] = true;  
while(flag[0])  
    if (vez == 0) {  
        flag[1] = false;  
        while (vez == 0) ;  
        flag[1] = true;  
    };  
/* Acesso ao recurso crítico */  
vez = 0;  
flag[1] = false;  
...  
}
```



# Exclusão mútua (*hardware*): *disabling interrupts*

Por inibição das interrupções

```
/* Processo */  
...  
/* Disable Interrupts */  
/* Acesso ao recurso exclusivo */  
/* Enable Interrupts */  
...  
}
```

Problemas: não permite o uso de *interrupts* pelo recurso, interfere com o SO, inclusive retirando a possibilidade de activar o *scheduler*



# Exclusão mútua (*hardware*): Variáveis de bloqueio por funções *Test-and-Set*

Instrução do  
processador

```
Enter_csection(int &val)
{
    while(!testandset(val));
}
```

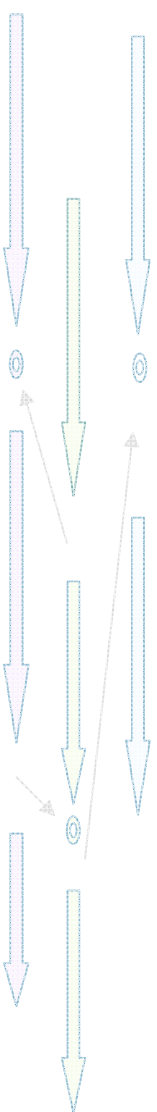
```
Leave_csection(int &val)
{
    val = 0;
}
```

```
testandset(int &val)
{
    if (val == 0){
        val=1;
        return 1;
    }
    else return 0;
}
```

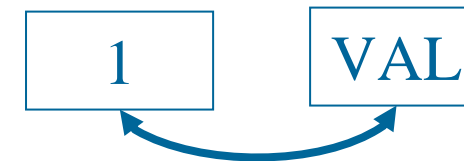
Problemas: a espera não é limitada, podendo levar à situação de **starvation**; e tem espera activa.



# Função *Test and Set*, por instrução *Exchange*



```
int TestAndSet(int &val)
{
    int x = 1;
    __asm{
        // x <-> *val
        mov    eax, dword ptr [val]
        mov    ecx, dword ptr [x]
        xchg   ecx, dword ptr [eax]
        mov    dword ptr [x], ecx
    }
    return x == 0 ? 1 : 0;
}
```



Exchange

Se Val = 0  
val ← 1  
ret 1

Se Val = 1  
val ← 1  
ret 0



# Exemplo com garantia de exclusão mútua

```
DWORD WINAPI IncFunc(LPVOID args)
```

```
{
```

```
    int tmp;
```

```
    for(int i=0; i<50; i++){
```

```
        Enter_csection(token);
```

```
        /* Inicio da região crítica */
```

```
        tmp = x;
```

```
        tmp++;
```

```
        x = tmp;
```

```
        /* Fim da região crítica */
```

```
        Leave_csection(token);
```

```
    }
```

```
    return 0;
```

```
}
```

```
Enter_csection(int &val) {  
    while(!testandset(val));  
}
```

C:\WINDOWS\system32\cmd.exe

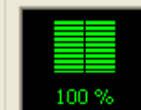
Terminei com o valor de x = 100  
Press any key to continue . . .

Windows Task Manager

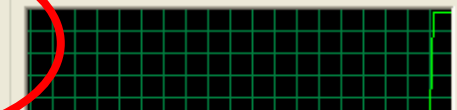
File Options View Help

Applications Processes Performance Networking

CPU Usage



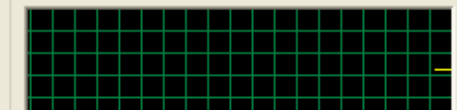
CPU Usage History



PF Usage



Page File Usage History



Totals

Handles	13027
Threads	541
Processes	47

Physical Memory (K)

Total	490992
Available	61160
System Cache	120472

Commit Charge (K)

Total	533220
Limit	1148916
Peak	533948

Kernel Memory (K)

Total	46524
Paged	36772
Nonpaged	9752

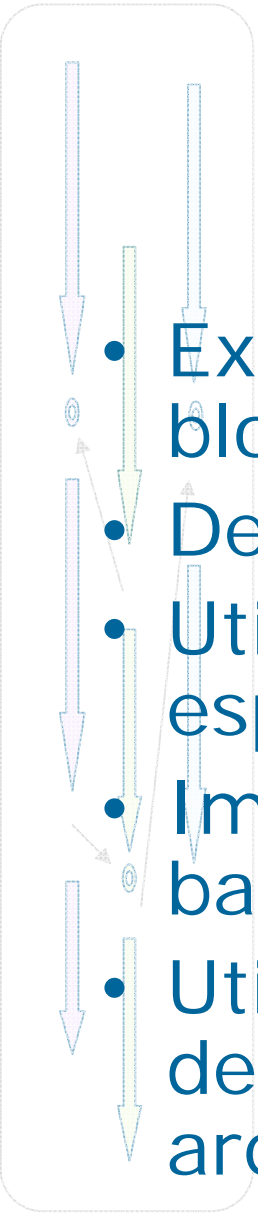
Processes: 47

CPU Usage: 100%

Commit Charge: 520M / 1121M



# *Spin lock*

- 
- Exclusão mútua baseada numa variável de bloqueio (*lock*) com espera activa
  - Deve ser evitado pois gasta tempo de CPU
  - Utilizado apenas quando é expectável uma espera muito breve
  - Implementados em muitas arquitecturas com base numa operação de *Test and Set*
  - Utilizado pelos núcleos dos SO (*kernel*) para definirem exclusão mútua em sistemas com arquitecturas de multi-processadores

