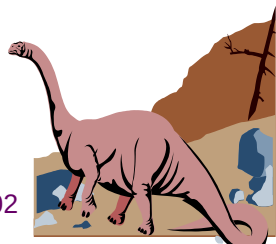
Capítulo 7 – Sincronização entre processos

SUMÁRIO:

- Problemas inerentes à gestão de recursos
- Inconsistência de dados versus sincronização de processos
- Os problemas clássicos de sincronização
- O problema das secções críticas
- Soluções para o problema da inconsistência de dados
  - Software generico
  - Hardware para sincronização
  - Sincronização usando mecanismos e recurso do SO
    - Semáforos
    - Mecansimos de IPC
  - *Monitores*
- *Sincronização em Linux , Solaris 2 & Windows 2000*

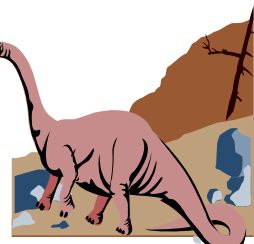


Problemas de gestão de recursos

- **INANIÇÃO (*starvation*):**
  Em consequência da política de escalonamento da CPU, um recurso passa alternadamente dum processo P1 para outro processo P2, deixando um terceiro indefinidamente bloqueado sem acesso ao recurso.

- **INCONSISTÊNCIA/CORRUPÇÃO DE DADOS:**
  Dois processos que tem acesso a uma mesma estrutura de dados não devem poder atualizá-la sem que haja algum processo de sincronização..
  - *Exemplo: a interrupção dum processo durante a atualização duma estrutura de dados pode deixá-la num estado de inconsistência*

- **ENCRAVAMENTO / IMPASSE / BLOQUEIO MÚTUO (*deadlock*):**
  Quando 2 processos se bloqueiam mutuamente.
  - *Exemplo: o processo P1 acede ao recurso R1, e o processo P2 acede ao recurso R2; a uma dada altura P1 necessita de R2 e P2 de R1.*



O porquê da sincronização?

- O acesso concorrente a dados partilhados pode criar uma situação de inconsistência nestes dados.

- **Condição de corrida (*race condition*)**: Situação em que vários processos acedem e manipulam dados partilhados duma forma “simultânea” (uma corrida), deixando os dados num estado de (possível) inconsistência.

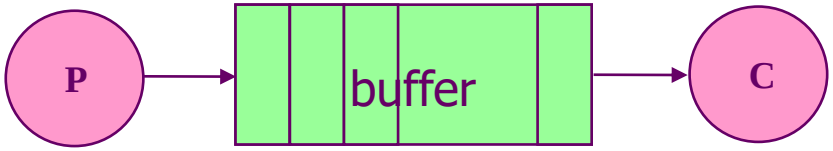
  - A manutenção da consistência de dados requer mecanismos que assegurem a executada ordenada e correcta dos processos cooperantes.

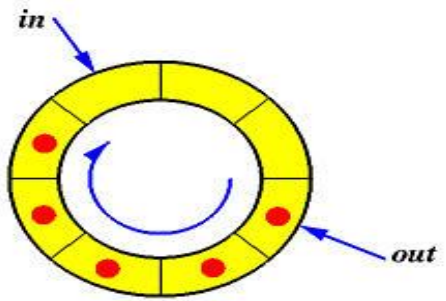
  - Os processos têm, pois, de ser **sincronizados** para impedir qualquer condição de corrida.

O problema do produtor-consumidor (bounded-buffer)

dados **partilhados**

```
#define BUFFER_SIZE 10

typedef struct {. . .} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```



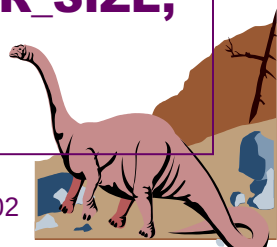


processo **produtor**

```
item nextProduced;
while (1) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

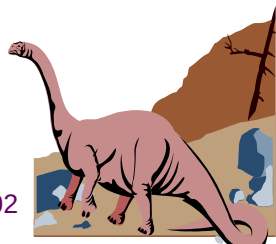
processo **consumidor**

```
item nextConsumed;
while (1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```



Exemplo

- Consider o código seguinte
- P1 – “Produtor” .. Execute a instrução counter ++
- P2 – “Consumidor” - Execute a instrução counter --
- O valor inicial de counter é 5
- Os dois processo executem concorrentemente.
- Qual o valor final ?



Inconsistência na actualização de dados

- As seguintes instruções devem ser executadas atomicamente:
  counter++;
  counter--;
- Uma operação atómica é uma instrução executada na totalidade sem interrupção.
- A instrução **counter++** pode ser implementada em linguagem assembly do seguinte modo:
  - **register1 = counter**
  - **register1 = register1 + 1**
  - **counter = register1**
- A instrução **counter--** pode ser implementada como:
  - **register2 = counter**
  - **register2 = register2 – 1**
  - **counter = register2**

- Se os dois processos tentam actualizar a variável concorrentemente, as suas instruções em linguagem assembly podem entrelaçar-se.
- Assuma que **counter** tem inicialmente o valor 5. Um possível entrelaçamento de instruções dos dois processos é:
  producer: **register1 = counter**
  (*register1 = 5*)
  producer: **register1 = register1 + 1**
  (*register1 = 6*)
  consumer: **register2 = counter**
  (*register2 = 5*)
  consumer: **register2 = register2 – 1**
  (*register2 = 4*)
  producer: **counter = register1**
  (*counter = 6*)
  consumer: **counter = register2**
  (*counter = 4*
- O valor de **count** pode ser 4 ou 6, mas o valor correcto devia ser 5.

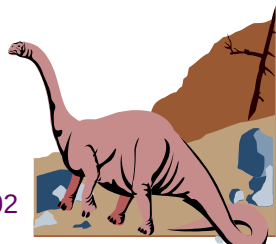
Exemplo

Exemplo do Mundo Real
    Spooler

EXEMPLO

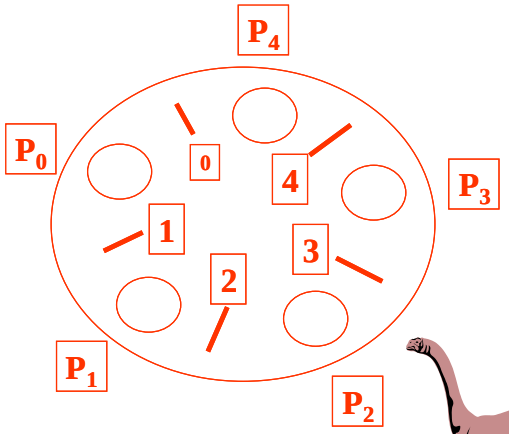
Outro
Exemplo do Mundo Real
    Data Base

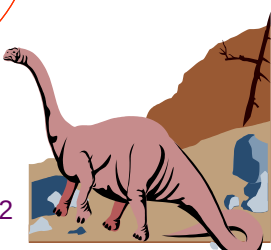
Registo é lido e alterado por dois processos / duas threads



Problemas Classicos de Sincronização

- 1 - Produtor-Consumidor (bounded-buffer)
  - O processo produtor produz itens e os insere numa fila, outro processo, o consumidor, retire os itens da fila. A fila é uma estrutura de dados manipulado e partilhada pelos dois processos.
- 2 - Readers/Writers Problem
  - Dados partilhados entre vários processos. Existem processos que apenas lerem os dados e outros que apenas escrevem - modificando os dados. Sincronização necessária para que dados a serem lidos não estão a ser alteradas
- 3 – Jantar de Filósofos
  - Partilha dum numero finito de recursos entre um numero de processo maior do que o numero de recursos





Requisitos de manutenção da consistência de dados partilhados

- Na partilha de dados por vários processos, cada processo tem um segmento de código, chamado **SECÇÃO CRÍTICA**, no qual os dados partilhados são acedidos. .
- Problema – assegurar que, enquanto um processo está a executar na secção crítica, nenhum outro processo é permitido executar na sua secção crítica.

repeat

entry section

critical section

exit section

remainder section

until *end*;

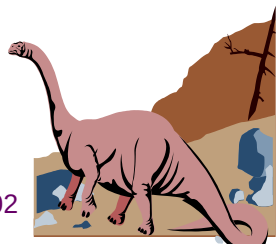
- **Exclusão Mútua**. Só um processo de cada vez pode entrar e executar na secção crítica.
- **Progressão**. Um processo a executar numa secção não-crítica não pode impedir outros processos de entrar na secção crítica.
- **Espera limitada**. Um processo que queira entrar numa secção crítica não deve ficar à espera indefinidamente.

Nota que : Não podemos presumir nada sobre velocidade de execução, prioridade ou numero de processos.

Soluções para o problema da inconsistência de dados

A consistência de dados pode ser garantida através de vários mecanismos de *sincronização:*

- **Software** mecanismos genéricos duma linguagem de programação
  - **algoritmo experimental – Alternância Estrita de 2 processos**
  - **algoritmo de Dekker – 2 processos 1962/63**
  - **algoritmo de Peterson – 2 processos 1981**
  - **algoritmo de Lamport – n processos 1974**
- **Hardware** mecanismos disponibilizados pela hardware
  - Disable Interrupts
  - Instruções Assembler Especiais (Test and Set etc.)
- **Recursos do sistema operativo**
  - semáforos, passagem de mensagens, etc
- **Monitores**
  - mecanismo de alto-nível - específico duma linguagem de programação ou biblioteca, por exemplo synchronized em Java



Solução Alternância Estrita

Processo 1

```
while ( vez == 2 ) ;          {testa acesso Entry protocol}
   <seccao critica>
vez = 2;                      {autoriza acesso ao outro processo: Exit protocol}
<resto do algoritmo>
```

Processo 2

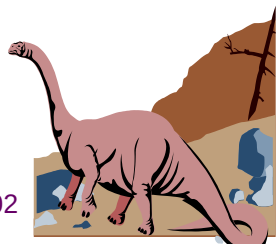
```
while ( vez == 1 ) ;          {testa acesso}
   <seccao critica>
vez = 1;                      {autoriza acesso ao outro processo}
<resto do algoritmo>
```

int vez =1; //shared variable

Isto é uma "pre-condition" – CONDIÇÃO PREVIA

Disto podemos deduzir que

Para sempre: vez=1 \/ vez=2



Solução Alternância Estrita

Processo 1

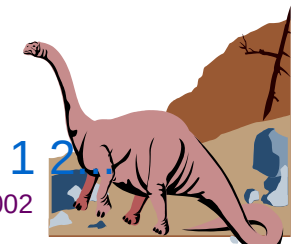
```
while ( vez == 2 ) ;          {testa acesso Entry protocol}
   <seccao critica>
vez = 2;                      {autoriza acesso ao outro processo: Exit protocol}
<resto do algoritmo>
```

Processo 2

```
while ( vez == 1 ) ;          {testa acesso}
   <seccao critica>
vez = 1;                      {autoriza acesso ao outro processo}
<resto do algoritmo>
```

Análise do algoritmo:

- Garante a **exclusão mútua**, mas só é aplicável a 2 processos.
- Não garante a **progressão**. Se um dos processos quiser entrar 2 vezes consecutivas na SC, não pode. Só há a garantia da **alternância** dos processos na execução da secção crítica.
- Não garante a **espera limitada**. Se um processo falha o outro ficará permanentemente bloqueado (deadlock).
- Só funciona se os dois processos forem absolutamente alternativos: 1 2 1 2 ...



Algoritmo de Dekker – 2 processos

Processo 1

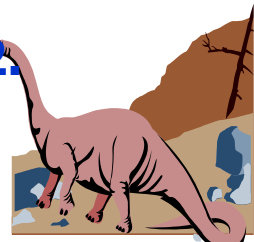
```
P1QuerEntrar = true;                    {P1 está pronto a entrar na SC}
while P2QuerEntrar do                   {mas dá a vez a P0, se ele precisar}
    if vez == 2 then
    {
          P1QuerEntrar = false;
          while vez == 2 ;              {testa acesso}
          P1QuerEntrar = true;
    };
<executa seccao critica>
vez = 2;                                {autoriza acesso ao outro processo}
P1QuerEntrar = false;
<executa resto do algoritmo>
```

Processo 2

```
P2QuerEntrar = true;
while P1QuerEntrar do
    if vez == 1 then
    {
          P2QuerEntrar = false;
          while vez == 1 ;              {testa acesso}
          P2QuerEntrar = true;
    };
<executa seccao critica>
vez = 1;                                {autoriza acesso ao outro processo}
P2QuerEntrar = false;
<executa resto do algoritmo>
```

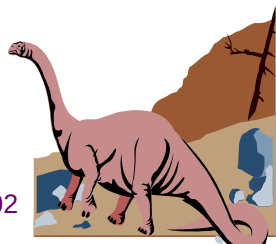
Análise do algoritmo:

- Garante **exclusão mútua**, mas obriga a que o número de processos seja **2**.
- Garante a **progressão**. Não obriga à alternância estrita.



Results

- Pausa Start: Iniciar press return
- process 1 .. 0 vez=1 .. 2180
- process 1 .. 1 vez=2 .. 2180
- process 1 .. 2 vez=2 .. 2180
- process 1 .. 3 vez=2 .. 2180
- process 1 .. 4 vez=2 .. 2180
- process 1 .. 5 vez=2 .. 2180
- process 1 .. 6 vez=2 .. 2180
- process 1 .. 7 vez=2 .. 2180
- process 2 .. 0 vez=2 ..3984
- process 2 .. 1 vez=1 ..3984
- process 2 .. 2 vez=1 ..3984
- process 2 .. 3 vez=1 ..3984
- process 1 .. 8 vez=2 .. 2180
- process 1 .. 9 vez=2 .. 2180
- process 1 .. 10 vez=2 .. 2180
- process 1 .. 11 vez=2 .. 2180
- process 1 .. 12 vez=2 .. 2180
- process 1 .. 13 vez=2 .. 2180
- process 1 .. 14 vez=2 .. 2180
- process 1 .. 15 vez=2 .. 2180
- process 2 .. 4 vez=1 ..3984
- process 2 .. 5 vez=1 ..3984
- process 2 .. 6 vez=1 ..3984
- process 2 .. 7 vez=1 ..3984
- process 2 .. 8 vez=1 ..3984
- process 2 .. 9 vez=1 ..3984
- process 1 .. 16 vez=1 .. 2180
- process 1 .. 17 vez=2 .. 2180
- process 1 .. 18 vez=2 .. 2180
- process 1 .. 19 vez=2 .. 2180
- error = 0.000000
- Pausa Fim : press return



Algoritmo (Peterson) Para – 2 processos

- Global Shared Variables int flag[2]={false,false], turn;
- Process P_i

```
do {
        flag [i] = true;
        turn = j;
        while ( flag [j]  && turn == j )  /*do nothing*/ ;
              critical section
        flag [i] = false;
              remainder section
} while (1);
```

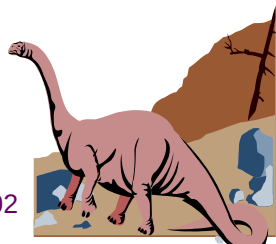
Análise do algoritmo:

- Garante **exclusão mútua**, mas obriga a que o número de processos seja 2.
- Garante a **progressão**. Não obriga à alternância estrita.

Pausa Start: Iniciar press return

process 1 .. 0 vez=2 .. 3356
process 1 .. 1 vez=2 .. 3356
process 1 .. 2 vez=2 .. 3356
process 1 .. 3 vez=2 .. 3356
process 1 .. 4 vez=2 .. 3356
process 1 .. 5 vez=2 .. 3356
process 1 .. 6 vez=2 .. 3356
process 1 .. 7 vez=2 .. 3356
process 1 .. 8 vez=2 .. 3356
process 1 .. 9 vez=2 .. 3356
process 1 .. 10 vez=2 .. 3356
process 1 .. 11 vez=2 .. 3356
process 1 .. 12 vez=2 .. 3356
process 1 .. 13 vez=2 .. 3356
process 1 .. 14 vez=2 .. 3356
process 1 .. 15 vez=2 .. 3356
process 2 .. 0 vez=2 ..3212
process 1 .. 16 vez=1 .. 3356
process 1 .. 17 vez=1 .. 3356
process 2 .. 1 vez=2 ..3212
process 2 .. 2 vez=2 ..3212
process 1 .. 18 vez=1 .. 3356
process 2 .. 3 vez=2 ..3212
process 1 .. 19 vez=1 .. 3356
process 2 .. 4 vez=1 ..3212
process 2 .. 5 vez=1 ..3212
process 2 .. 6 vez=1 ..3212
process 2 .. 7 vez=1 ..3212
process 2 .. 8 vez=1 ..3212
process 2 .. 9 vez=1 ..3212
error = 0.000000
Pausa Fim : press return

Results

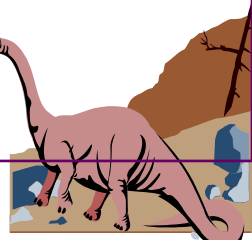


Algoritmo de Lamport – *n* processos

- A cada processo é atribuída uma senha de entrada na secção crítica.
- Antes de aceder à secção crítica, o processo executa a função max que lhe atribui uma senha superior a todas as outras.
  - Não está garantido. Ver função *compare*
- Depois de atribuída a senha, o processo efectua um ciclo para determinar se o seu número de ordem é o menor de todos.
- Quando o número de ordem (senha) for inferior ao de todos os outros, o processo entra na secção crítica.

```
Shared senha : array[0..N-1] of integer;
        escolha : array[0..N-1] of boolean;

function compare(a, b:integer): boolean;
begin
   if (senha[a]<senha[b])
          or (senha[a]==senha[b] and a<b)
          //desempatar pelo nome/id
      then return true;  else return false;
end;

function max: integer;
integer maximo:=0, i;
begin
   for i:=1 to N do
          if ( senha[i]>maximo )
              then maximo := senha[i];
   return maximo;
end;
```

```
procedure PROC(id:integer);
integer j;
begin
   while true do  {protocolo tem que fuuncionar em ciclo }
   begin
      escolha[id]:=true;        {atribuicao da senha}
      senha[id]:=max + 1;
      escolha[id]:=false;
      for j:=0 to N-1 do        {espera ate senha menor}
      begin
         while escolha[j] do;
         while (senha[j]<>0) and compare(j,id) do;
      end;
      <executa seccao critica>
      senha[id]:=0;
      <executa resto do algoritmo>
   end
end;
```



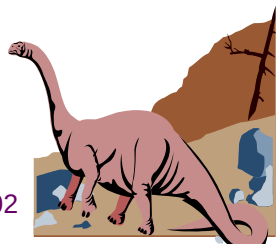
Implementação

Exercício para Casa:

Implementação com pthreads do algoritmo de Lamport (ling C)

Programas de testes

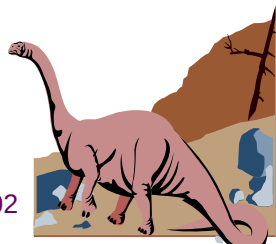
Relatório dos testes



```
#define true  1
#define false 0
//macros or functions
cmp(a, b,c, d) => (a < c) or ((a == c) and (b < d))
max( int[] )  => maximum( int[] )
//Shared Global Arrays
int Entering [NUM_THREADS]  =  {false};
int Number  [NUM_THREADS]   = {0};
```

```
void lock_entry(int id) {
        Entering[id] = true;
        Number[id] = 1 + max(Number);
        Entering[id] = false;
        for (int j = 0; j < NUM_THREADS; j++) {
                // Espera a minha vez dexiando os outros entrar
                while (Entering[j]) { ; /* spin */ }
                // Espera que as threads com MENOR senha ou maior "numero" acabam a sua secção critica
                while ((Number[j] != 0) && (cmp(Number[j], j, Number[id], id))) { ; /* spin */ }
        }
}
void unlock_exit(int id) {
        Number[id] = 0;
}
```

```
void * Thread(void  *idArg) {
   int id =  * (int *) idArg;
    while (true) {
          lock_entry(id);
          // The critical section goes here...
          unlock_exit(id);
          // non-critical section...
          }
}
```



Trincos lógicos por Software

As soluções anteriores são complexas para o programador que deseja escrever aplicações concorrentes e garantir a exclusão mútua.

Solução Desejável:

- Uma solução desejável é usar variáveis que funcionam como trincos lógicos de uma estrutura de dados.
- O trinco é fechado quando se acede à estrutura de dados e aberto à saída.
- O trinco lógico é referido como **trinco** e manipulado por duas primitivas: ***lock*** e ***unlock***.
- Um processo que tenta aceder a uma secção crítica protegida por um trinco ficará em ciclo no teste até o trinco ser libertado.

Implementação ?? dum Trinco

```
void lock(var trinco:boolean);
{
        while trinco ;    //busy wait
        trinco := true;
};
```

```
void unlock(var trinco:boolean);
{
        trinco := false;
};
```

•código em cima deverá estar analisado com muito cuidado.

•Ex. Mostre que não seja correto com um exemplo (program trace)

Para funcionar corretamente ainda temos de aliar este código com um dos algoritmos previamente visto ou com algum mecanismo de hardware (ver seguinte)

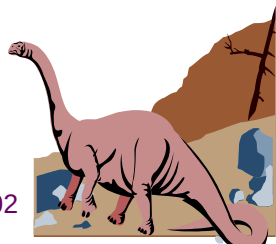
Trincos lógicos por Software

Implementação dum Trinco em C - pthreads

```
boolean trinco = false;  //global variables
int x = 0;               //
main() {
    create 2 threads
    join 2 thread
    print x
}
 void * thread(void *  args)
 {
1   lock ( &trinco );
2       x = x +1;          //secção critica
3   unock ( &trinco );
4   Return NULL;
   }
```

```
5 void lock (boolean *  trinco) {
6       while ( *trinco )
7                 ;    //spin lock-Busy Wait
8       *trinco = true;
   }
```

```
9  void unlock(boolean * trinco) {
10      * trinco = false;
   }
```

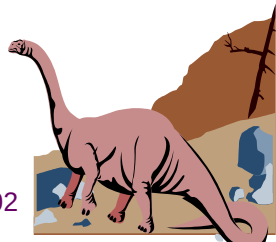


```
 int x = 0;
 pthread_mutex_t mutex1;
 void *addx (void *argumentos)
 {
   pthread_mutex_lock (&mutex1);
   x = x + 1; // seção crítica
   pthread_mutex_unlock (&mutex1);
 }

int main ()
{
 pthread_t thread[4];
 pthread_mutex_init (&mutex1, NULL);
 for (int i = 0; i < 4; i++)
   pthread_create (&thread[i], NULL, addx, NULL);
 for (int i = 0; i < 4; i++)
   pthread_join (thread[i], NULL);
 printf ("x = %d\n", x);
}
```

Inibição de interrupções

Permitir um processo desligar interrupções, Inibição de interrupções, antes de entrar uma secção critica e ligar as interrupções quando sair da secção critica:

- Assim CPU não pode trocar processos ! → Garante que um processo pode utilizar uma variável partilhada sem a interferência de nenhum outro processo
- Mas desligando interrupções vai dar muito trabalho…
  - Assim o computador não vai poder atender interrupções durante muito tempo.
  - Pode ser que um processo mal escrito ou com erro implica que a seguir o ***OS*** nunca mais conseguir retomar o controlo !
  - MultiCore/Multi-Processador
    - Desligando *interrupts* em apenas um processador não garante exclusão mutua duma variável !!!!
  - Problemas com as Caches (write backs etc)
  - Desvantagens e perigos são maiores que as vantagens



Trincos lógicos por Software / Hardware

Implementação dum Trinco

Exercício:

Usando as instruções de assembler tenta modificar o pseudocódigo a direita para criar um trinco de exclusão mutua.

```
void lock(var trinco:boolean);
{
        while trinco ;    //busy wait
        trinco := true;
};
```

```
void unlock(var trinco:boolean);
{
        trinco := false;
};
```

IF (Interrupt Flag) is a system flag bit in the x86 architecture's FLAGS register, which determines whether or not the CPU will handle maskable hardware interrupts.

The flag may be set or cleared using the CLI (Clear Interrupts), STI (Set Interrupts)

CLI clears IF (sets to 0), while STI sets IF to 1.

MAS.. CLI and STI are privileged instructions, which trigger a general protection fault if an unprivileged application attempts to execute it

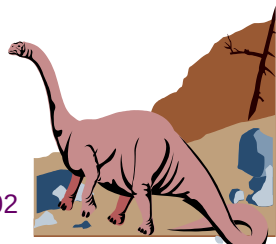
Linux Kernel

spin_lock_irqsave disables interrupts unconditionally.

It preserves the old state so when exiting the critical region the processor can be safely put back the way it was.

Using spin_lock_irqsave() and spin_lock_irqrestore().

```
spinlock_t mLock = SPIN_LOCK_UNLOCK;
unsigned long flags;

spin_lock_irqsave(&mLock, flags); // save the state, if locked already it is saved in flags

// Critical section

spin_unlock_irqrestore(&mLock, flags); // return to the formally state specified in flags
```



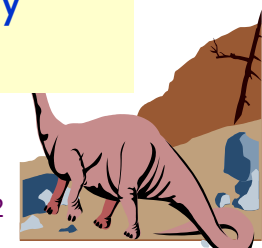
Trincos lógicos por hardware
Instruções Especias

Soluções possíveis:

1. Inibição de interrupções quando o processo se encontra na secção crítica. Mas, só é pratico para sistemas uniprocessador e mesmo assim já vimos que tem muitos desvantagens .
2. Instruções especiais (atómicas) do processador que se executam num ciclo de instrução:
   - *Test and Set* - Modificar o conteúdo duma variável em memória, e devolve o seu valor velho de forma atómica, permitindo assim a invocadora testar se houve mudança
   - *Compare and Exchange*

```
int TestAndSet( int * target, int value )
{
        int TAS = target;
        *target = value;
        return TAS;
}
```

- High level view of TAS as a função where all (3) instructions are executed atomicaly and indivisibly



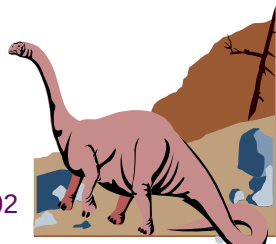
Exclusão Mutua Usando TSL Assembler

Code Block
  entry_section
    secção critica
  exit_section

NOTE flag= 1(lock set) 0 (free)
"flag" is Global
"register" is Local
entry_section ≈ while

```
entry_section:
 tsl    register, flag    ;copy flag to register and set flag to 1
 cmp    register, #0      ;was flag zero?
 jnz    entry_section     ;if flag was non zero, lock was already set,
                           so jump ..in other words loop
 ret                      ;return (and enter critical region)
```

```
exit_section:
  mov   flag, #0          ; store zero in flag
  ret                     ;return
```



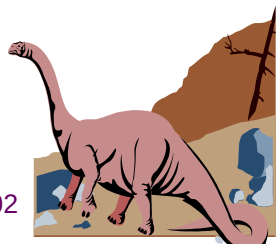
Specification of TSL Intel

BTS - Bit Test and Set (386+)

Usage: BTS dest,src
Modifies flags: CF

The destination bit indexed by the source value is copied into the Carry Flag and then set in the destination.

	Clocks				Size
Operands	808x	286	386	486	Bytes
reg16,immed8	-	-	6	6	4-8
mem16,immed8	-	-	8	8	4-8
reg16,reg16	-	-	6	6	3-7
mem16,reg16	-	-	13	13	3-7



Trincos lógicos por hardware Instruçoes Especias

A implementação da **exclusão mútua** poderá ser feita em varias formas.

Aqui são duas :

1- Não garante a espera limitada

```
while TestAndSet(lock) ;
    <seccao critica>
lock = false;
    <resto do algoritmo>
```

2 Garante a espera limitada

```
waiting[i] = true;      //process i
key = true;

while (waiting[i] and key)
        key = TestAndSet(lock) ;
waiting[i] = false;

    <seccao critica>

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;

    <resto do algoritmo>
```

o processo que sai da secção crítica determina qual o próximo processo a entrar (percorre a tabela waiting para colocar a variável waiting[j] do próximo processo a false). Se não existirem processos à espera de entrar, o trinco é libertado.

Resumo . So far so good

As soluções anteriores têm dois aspectos em comum:

- Todo o processo que não consegue entrar na secção crítica FICA À ESPERA de entrar.
- Não há nenhum mecanismo que evite a preempção dum processo quando está a executar a zona crítica ⇒ **Existencia de "ESPERA ACTIVA"**
- ESPERA ACTIVA: acontece quando um processo que está na secção crítica é retirado do processador :
  - O trinco mantém-se fechado
  - Os outros processos que vão passando sucessivamente pelo(s) processador(es) continuam a testar o trinco à espera da sua abertura e a gastar CPU.
    - quer dizer que os outros processos estão a espera "activamente" e a gastar CPU

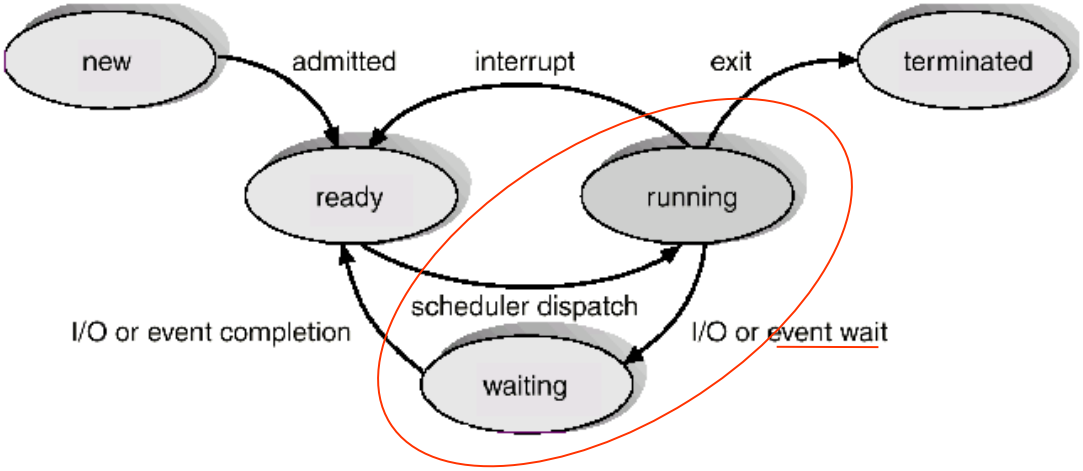
Semáforos

Semáforo: mecanismo de sincronização sem espera activa. Para evitar a espera activa, um processo que espera a libertação dum recurso deve ser bloqueado, devendo a razão que o levou a bloquear ficar memorizada.

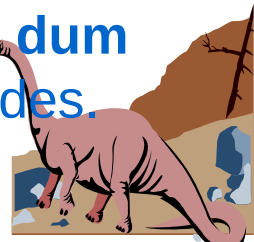
Um semáforo consiste numa **variável** e numa **fila de espera** associada a um **recurso**. Esta fila contém todos os descritores dos processos bloqueados no semáforo

Bloquear um processo num semáforo significa retirá-lo do estado de execução (running state) e movê-lo para a fila de processos bloqueados (waiting state) do respectivo semáforo.

Desbloquear um processo é o mesmo que retirá-lo da fila de processos bloqueados (waiting state) e inseri-lo na fila de processos executáveis (ready state).



Gestão da fila de processos bloqueados dum semáforo: normalmente, FCFS ou prioridades.



Semáforos

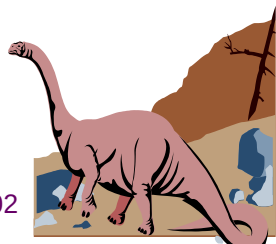
Definir um semáforo como um registo (record)

typedef struct { int value; processQueue queue;} semaphore;

Um semáforo é manipulado através de **duas** primitivas atómicas:

wait: para receber um sinal, um processo executa *wait*, e bloqueia no caso do semáforo impedir a continuação da sua execução;

signal: para enviar um sinal, um processo executa *signal*; caso um ou mais processos estejam bloquedos no semaforo então um dos processos será desbloqueado.

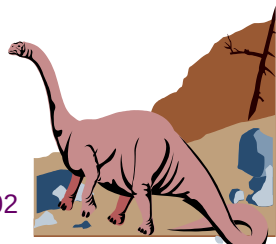


Syntax

POSIX	JAVA
sem_wait() sem_post()	aquire() release()

Originalmente

P() – probern (to test → wait)
V() – verhogen (to increment → signal)



Semáforos Implementação

(semáforo binário, 0≤s≤1)

Define a semaphore as a record

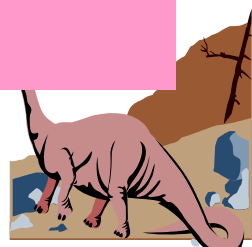
typedef struct { int value; processQueue queue;} semaphore;

wait(s)

```
if ( s.value == 1)
   s.value =  0;
else
{ //suspend process
   place this calling process in s.queue;
   block this calling process;
}
```

signal(s)

```
if ( s.queue is empty)
   s.value =  1;
else
{ //wakeup process
   remove a process P from s.queue;
   place process P on ready queue;
}
```



Semáforos
(exclusão mutua)

Manipulação: um semáforo é manipulado através de **duas** primitivas atómicas:

signal: para enviar um sinal, um processo executa *signal*;

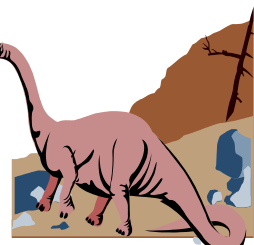
wait: para receber um sinal, um processo executa *wait*, e bloqueia no caso do semáforo impedir a continuação da sua execução;

Funcionamento:

- 2 ou mais processos podem executar uma secção critica com exclusão mutua, progressão e espera limitada
- O protocolo é efetuado usando um semáforo binário.

- SC uma secção critica
- Usa semáforo s inicializada a 1
- Código:

P_1	P_2
wait(s)	***wait(s)***
SC	SC
signal(*s*)	***signal(s)***



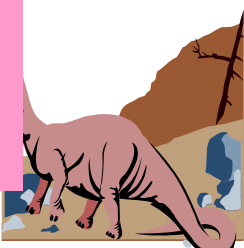
Exclusão mútua através de semáforos

Processos P1,P2 ..Pn executem concorrentemente uma secção critiça com exclusão mutua, satisfendo as regras da progessão e espera limitida

```
#define  n  10 ;  /* number of processes */

sem_t  s;
```

```
void * P (void * id)
{
  do
        sem_wait(&s);
        <executa seccao critica>
        sem_post(&s);

        <executa resto do algoritmo>
  while (1);
}
```

```
int main()
{
   sem_init(&s,0,1); /* Inicialize semaphore to 1 */
   (* Inicia n tarefas concorrentemente *)
   for (i=0;i<n;i++) pthread_create( null,null, P, null );
}
```



Semáforos
(manipulação de eventos)

Manipulação: um semáforo é manipulado através de **duas** primitivas atómicas:

signal: para enviar um sinal, um processo executa *signal*;

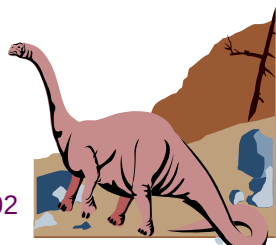
wait: para receber um sinal, um processo executa *wait*, e bloqueia no caso do semáforo impedir a continuação da sua execução;

Funcionamento:

- 2 ou mais processos podem cooperar através de sinais, em que um processo é obrigado a parar num local especificado até receber um sinal específico.
- A sinalização é feita através dum semáforo.

- Executa *B* em P_1 **só após**
  - *A* ser executada em P_2
- Usa *flag* do semáforo inicializada a 0
- Código:

P_2	P_1
A	***wait***(*s*)
signal(*s*)	*B*



Semáforos Implementação

(semáforo genérico, s (inicial)≥0)

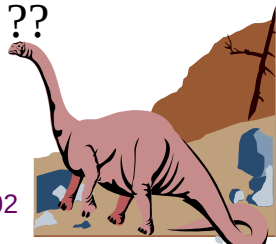
wait(s)

```
s.count--;
if (s.count < 0)
{
   place this process in s.queue;
   block this process;
}
```

signal(s)

```
s.count++
if (s.count <= 0)
{
   remove a process P from s.queue;
   place process P on ready queue;
}
```

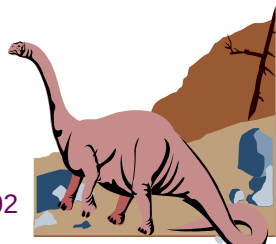
Qual processo será removido da fila do semaforo e desbloqueado ??



Counting Semaphore // Linux Src

kernel/locking/semaphore.c

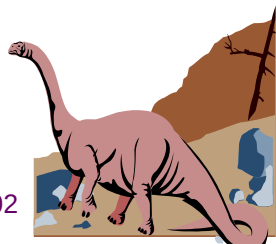
```
struct semaphore {
    raw_spinlock_t        lock;
    unsigned int          count;
    struct list_head      wait_list;
};

void down(struct semaphore *sem)
{
    unsigned long flags;
    raw_spin_lock_irqsave( &sem->lock, flags );
    if (likely(sem->count > 0))
      sem->count--;
    else
       __down(sem);
    raw_spin_unlock_irqrestore( &sem->lock, flags );
}

void __sched __down(struct semaphore *sem)
{
  __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
```



Semáforos
(aplicações possíveis)

Um semáforo **s** é criado com um valor inicial. Os casos mais comuns são:

- **s=0** Sincronização entre processos (por ocorrência de eventos)
- **s=1** Exclusão mútua
- **s≥2** Controlo de acesso a um grupo finito de recursos

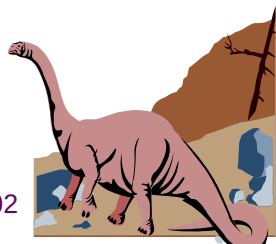


Impasse e Inanição
(deadlock e starvation)

- **Deadlock** – dois ou mais processos estão à espera indefinidamente por um evento que só pode ser causado por um dos processos à espera.
- Exemplo: Sejam P_0 e P_1 dois processos e S e Q dois semáforos inicializados a 1; depois, P_0 passa a aceder ao recurso sinalizado por S e P_1 ao recurso sinalizado por Q:

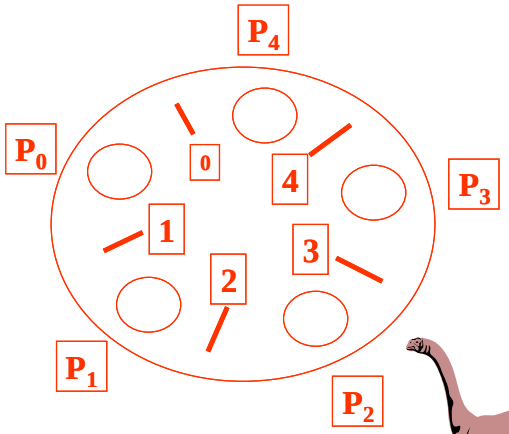
P_0	P_1
***wait* (*S*);**	***wait* (*Q*);**
***wait* (*Q*);**	***wait* (*S*);**
……	**……..**
***signal* (*S*);**	***signal* (*Q*);**
***signal* (*Q*);**	***signal* (*S*);**

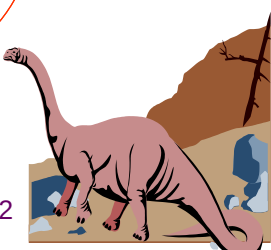
- **Starvation** – bloqueio indefinido. Um processo corre o risco de nunca ser removido da fila do semáforo, na qual ele está suspenso
- Semáforos não evitem, por si próprio, estes problemas !.



Problemas Classicos de Sincronização

- 1 - Produtor-Consumidor (bounded-buffer)
  - O processo produtor produz itens e os insere numa fila, outro processo, o consumidor, retire os itens da fila. A fila é uma estrutura de dados manipulado e partilhada pelos dois processos.
- 2 - Readers/Writers Problem
  - Dados partilhados entre vários processos. Existem processos que apenas lerem os dados e outros que apenas escrevem - modificando os dados. Sincronização necessária para que dados a serem lidos não estão a ser alteradas
- 3 – Jantar de Filósofos
  - Partilha dum numero finito de recursos entre um numero de processo maior do que o numero de recursos

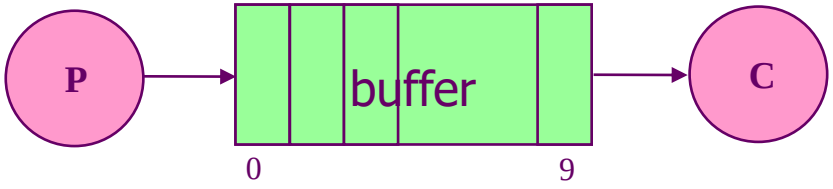


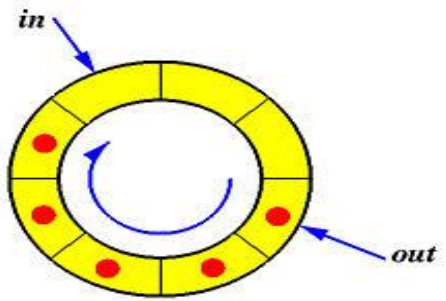


O problema do produtor-consumidor c/ entrepósito limitado (bounded-buffer) em memória partilhada

dados **partilhados**

```
#define BUFFER_SIZE 10

typedef struct {. . .} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```





processo **produtor**

```
item nextProduced;
while (1) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

processo **consumidor**

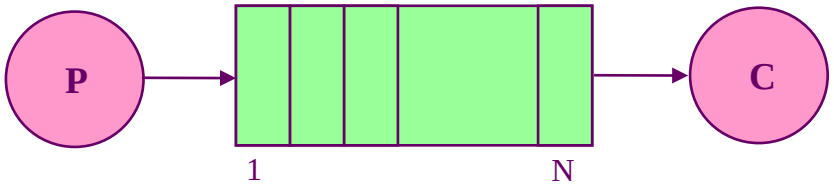
```
item nextConsumed;
while (1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

Semáforos

(Controlo de acesso a recursos com capacidade limitada)

Esta solução utilize 3 semaforos !

Problema do produtor-consumidor




```
semaphore  excMutua = 1;
semaphore  vazio = N;
semaphore  cheio = 0;
```

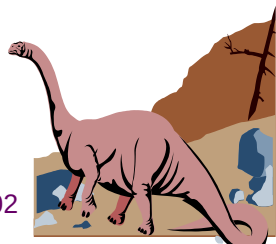
P

```
void Produtor ()
{
   int dado;

   wait(vazio);
     wait(excMutua);
       <adiciona dado ao buffer>
     signal(excMutua);
   signal(cheio);
}
```

C

```
void Consumer ()
{
   int dado;

   wait(cheio);
     wait(excMutua);
       <retire dado do buffer>
     signal(excMutua);
   signal(vazio);
}
```

```
void main()
{
   cria semáforos;
   (* Inicia 2 tarefas em paralelo *)
   parbegin (Produtor, Consumidor)
}
```



Producer Consumer

- 1: Utilizando busy-wait - pc-bw.c
- 2: Utilizando um semaphore pc-s1.c
- O programa em cima é um bom exemplo das "races conditions". Variando o "wait" varia os resultados. Eventualmente chegará a conclusão que para que o consumer consome tudo o que o producer produz apenas um semaphore não chegue !
- 3: Utilizando dois semaphores pc-s2.c a solução é facil
- Implementações com pthreads.

- Exemplo – unix semaforos