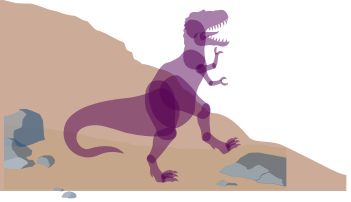
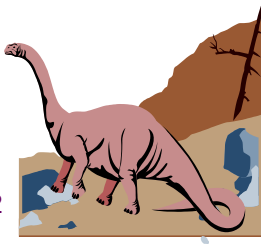
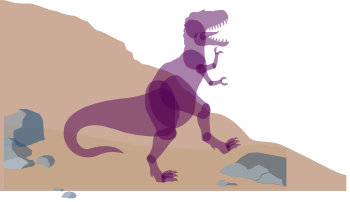


Capítulo 7 – Sincronização entre processos

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*



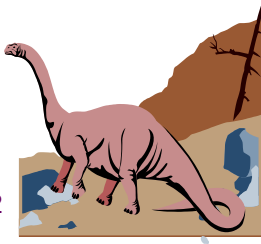


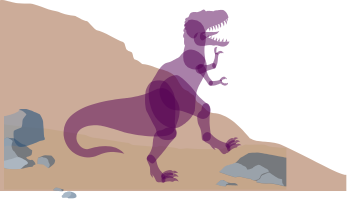
Problemas de gestão de recursos

- **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.

- **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.*

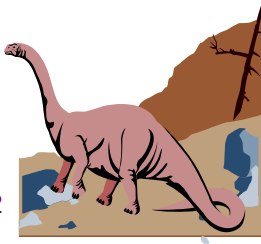
- **INCONSISTÊNCIA/CORRUPÇÃO DE DADOS:**
  Dois processos que tem acesso a uma mesma estrutura de dados não devem poder actualizá-la sem que haja algum processo de sincronização..
  - ☞ *Exemplo: a interrupção dum processo durante a actualização duma estrutura de dados pode deixá-la num estado de inconsistência*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

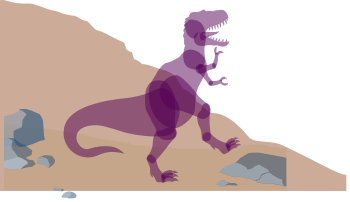




O porquê da sincronização?

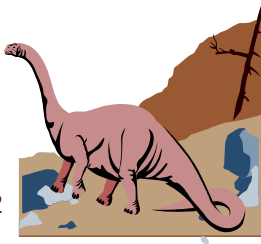
- O acesso concorrente a dados partilhados pode criar uma situação de inconsistência nestes dados.
- **Condição de corrida (*race condition*)**: Situação em que vários processos acedem e manipulam dados partilhados duma forma “simultânea”, deixando os dados num estado de (possível) inconsistência.
-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





Exemplo

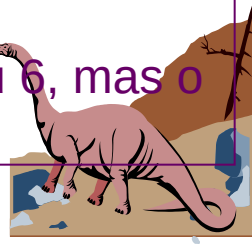
- Consider o código seguinte
- P1 – “Produtor” .. Execute a instrução counter ++
- P2 – “Consumidor” - Execute a instrução   counter --
- O valor inicial de counter é 5
- Os dois processo executem concorrentemente.
- Qual o valor final ?



Inconsistência na actualização de dados

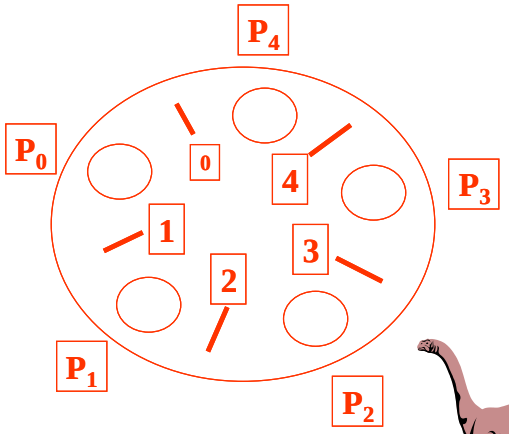
- As seguintes instruções devem ser executadas atomicamente:
  counter++;
  counter--;
- Uma operação atómica é instrução executada na totalidade sem interrupção.
- A instrução **counter++** pode ser implementada em linguagem assembly do seguinte modo:
  - **register1 = counter**
  - **register1 = register1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

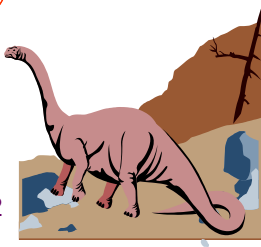
- Se os dois processos tentam actualizar a variável concorrentemente, as suas instruções em linguagem assembly podem entrelaçar-se.
- Assuma que **counter** tem inicialmente o valor 5. Um possível entrelaçamento de instruções dos dois processos é:
  producer: **register1 = counter**
  (*register1 = 5*)
  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.

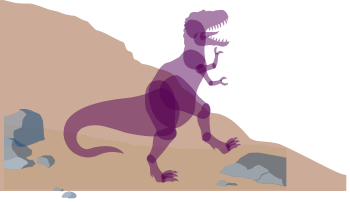


Problemas Classicos de Sincronização

- 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

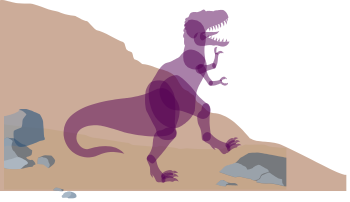
exit section

remainder section

until *false*;

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

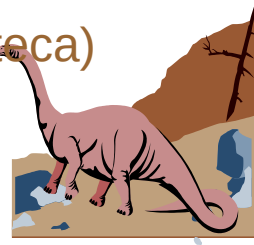
Nota que não podemos presumir nada sobre velocidade, prioridade ou numero de processos…

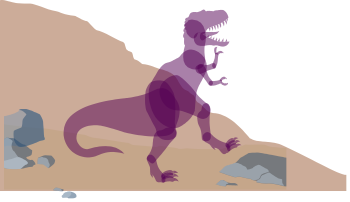


Soluções para o problema da inconsistência de dados

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**
  - **algoritmo de 2 – 2 processos**
  - **algoritmo de Lamport – n processos**
- **Hardware** mecanismos disponibilizados pela hardware
  - Disable interrupts
  - Test and Set
- **Recursos do sistema operativo**
  - semáforos, passagem de mensagens, etc
- **Monitores**
  - mecanismo específico duma linguagem de programação ou biblioteca)





Solução Alternância Estrita

Processo 1

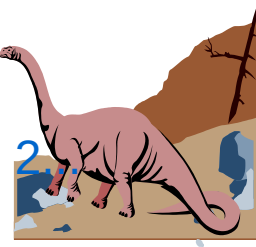
```
while vez == 2;                 {testa acesso}
   <seccao critica>
vez = 2;                        {autoriza acesso ao outro processo}
<resto do algoritmo>
```

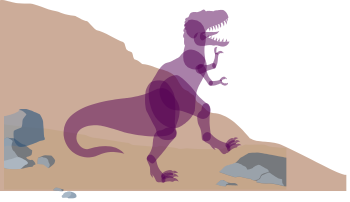
Processo 2

```
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

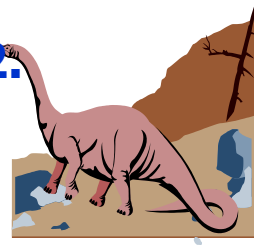
```
P1QuerEntrar = true;                    {P1 está pronto a entrar na SC}
while P2QuerEntrar do                   {mas dá a vez a P0, se ele precisar}
    if vez == 2 then
    {
          P1QuerEntrar = false;
          while vez == 2 ;              {testa acesso}
          P1QuerEntrar = true;
  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

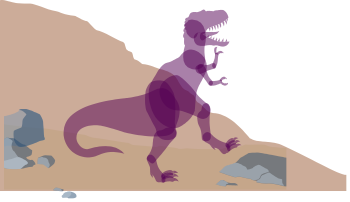
Processo 2

```
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:

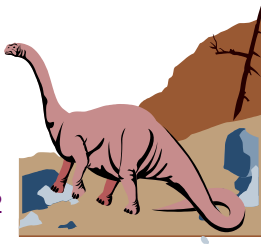
- Garante **exclusão mútua**, mas obriga a que o número de processos seja **2**.
- Garante a **progressão**. Não obriga à alternância estrita.

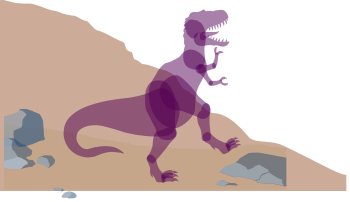




Results

- 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





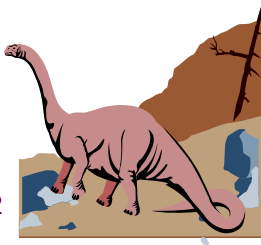
Algoritmo Para – 2 processos

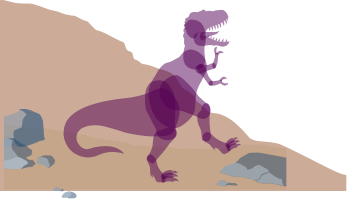
- Process P_i

```
do {
        flag [i]:= true;
        turn = j;
        while (flag [j] and turn = j) ;
                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.

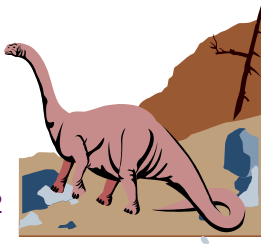


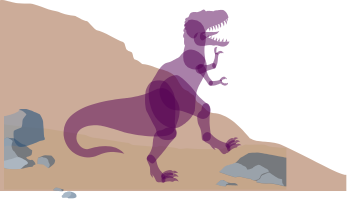


Pausa Start: Iniciar press return

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



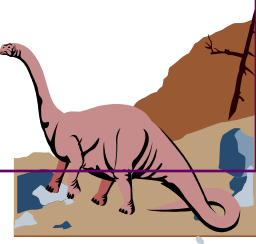


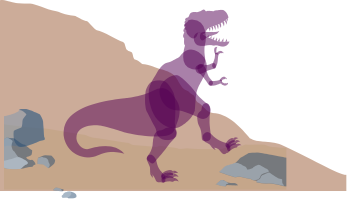
Algoritmo de Lamport – *n* processos

- 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.
- 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.

```
var senha : array[0..N] of integer;
    escolha : array[0..N] of boolean;

function compara(a, b:integer): boolean;
begin
   if (senha[a]<senha[b])
              or (senha[a]=senha[b] and a<b)
   then compara := true
   else compara := false;
end;

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

```
procedure PROC(i:integer);
var j:integer;
begin
   while true do
   begin
      escolha[i]:=true;        {atribuicao da senha}
      senha[i]:=max + 1;
      escolha[i]:=false;
      for j:=1 to N do         {espera ate senha menor}
      begin
            while escolha[i] do;
            while (senha[j]<>0) and compara(j,i) do;
      end;
      <executa seccao critica>
      senha[i]:=0;
      <executa resto do algoritmo>
    end
end;
```





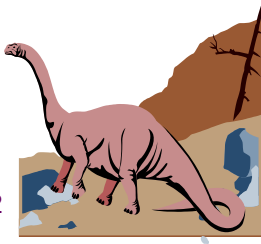
Implementação

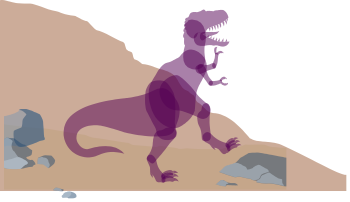
Exercicio :

Implementação com pthreads do algoritmo de Lamport (ling C)

Programas de testes

Relatorio dos testes





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.

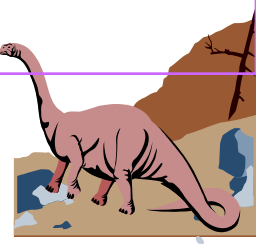
Implementação dum Trinco

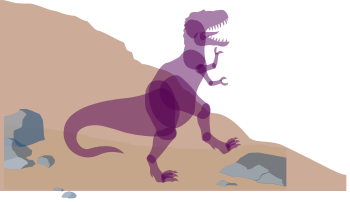
```
void lock(var trinco:boolean);
{
        while trinco ;
        trinco := true;
};
```

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

O código em cima deverá estar analisado com muito cuidado.

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

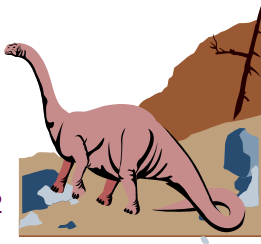


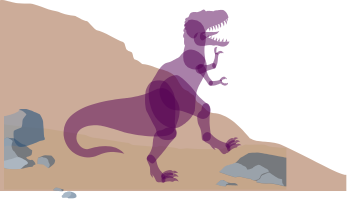


Inibição de interrupções

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 o computador nunca mais conseguir tratar duma interrupções .. crash
- Desvantagens e perigos são maiores que as vantagens



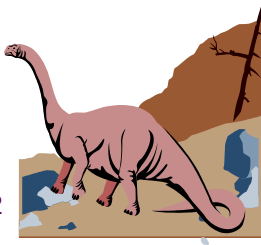


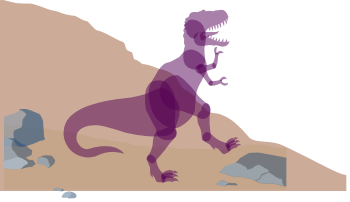
Trincos lógicos por hardware Instruçoes Especias

Soluções possíveis:

- Inibição de interrupções quando o processo se encontra na secção crítica. Mas, só é pratico para sistemas uniprocessador e mesmo assim j+a vimos tem muitos desvantagens .
- Instruções especiais (atómicas) do processador que se executam num ciclo de instrução:
  - ☐ *Test and Set*
  - ☐ *Compare and Exchange*

```
boolean TestAndSet(var target: boolean)
{
        boolean TAS = target;
        target = true;
        return TAS;
}
```





Trincos lógicos por hardware Instruçoes Especias

A implementação da **exclusão mútua** poderá ser feita em varias formas. Aqui são duas :

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

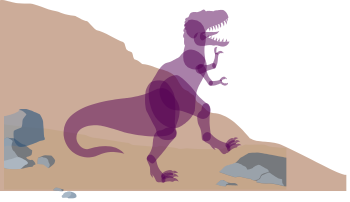
Não garante a espera limitada

```
boolean TestAndSet(var target: boolean)
{
        boolean TAS = target;
        target = true;
        return TAS;
}
```

```
waiting[i] = true;
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>
```

Garante a espera limitada

pois 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.

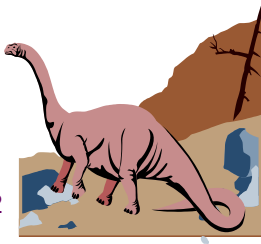


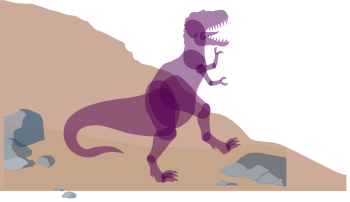
Resumo . So far so good

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, e os outros processos que vão passando sucessivamente pelo(s) processador(es) continuam a testar o trinco à espera da sua abertura – quer dizer que os outros processos estão a espera "activamente"





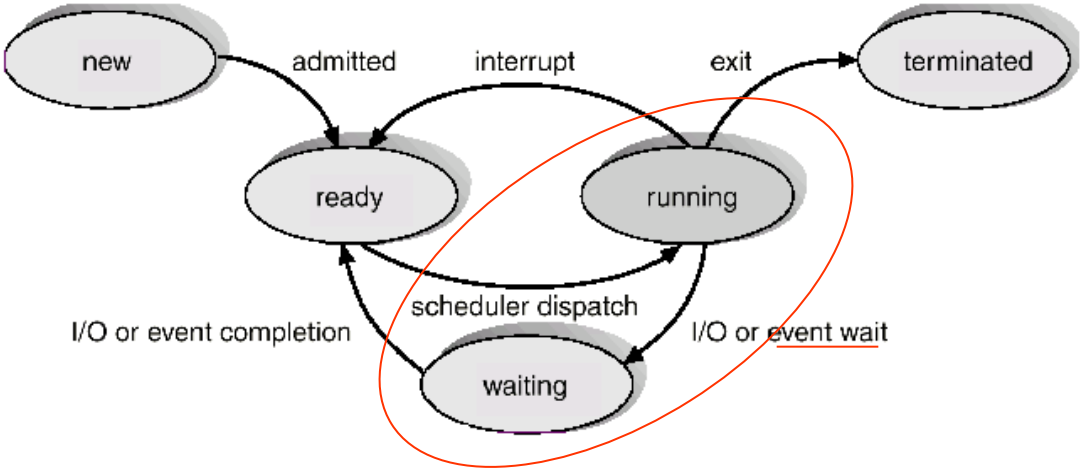
Semáforos

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.

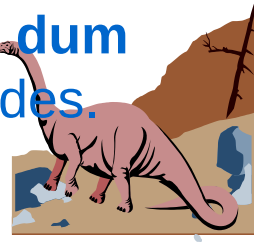
Um semáforo consiste numa **variável** e numa **fila de espera** associada a um **recurso**. Esta fila contém todos os descritores dos processos bloqueados no semáforo

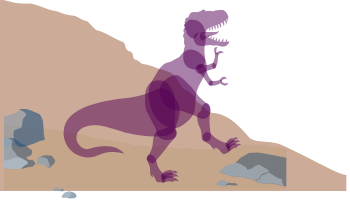
Bloquear um processo num semáforo significa retirá-lo do estado de execução (running state) e movê-lo para a fila de processos bloqueados (waiting state) do respectivo semáforo.

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
(manipulação de eventos)

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

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

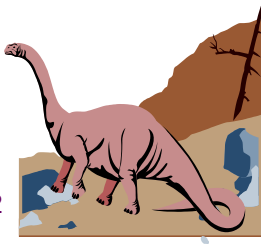
wait: para receber um sinal, um processo executa *wait*, e bloqueia no caso do semáforo impedir a continuação da sua execução;

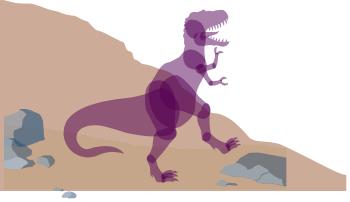
Funcionamento:

- 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***(*flag*)
signal(*flag*)	*B*





Semáforos Implementação

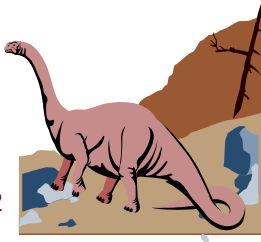
(semáforo binário, 0≤s≤1)

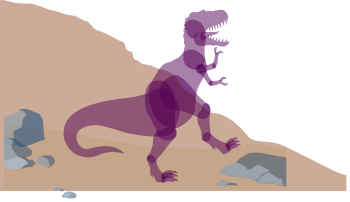
wait(s)

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

signal(s)

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





Semáforos Implementação

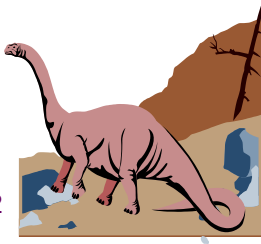
(semáforo genérico, s≥0)

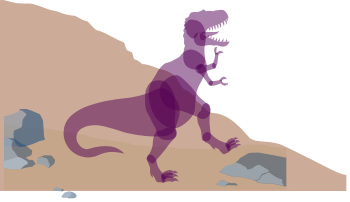
wait(s)

```
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;
}
```

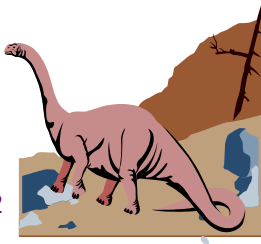


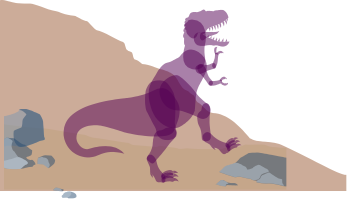


Semáforos
(aplicações possíveis)

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





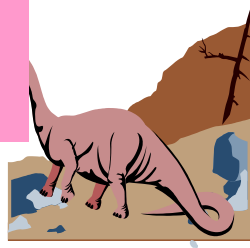
Exclusão mútua através de semáforos

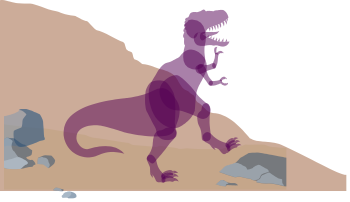
Processos P1,P2 ..Pn executem concorrentemente/paralelmente uma secção critiça com exclusão mutua, satisfendo as regras da progessão e espera limitida

```
const n = ...; (* number of processes *)

Semaphore  s = 1;
```

```
void P (int i)
{
   do
            wait(s);
            <executa seccao critica>
            signal(s);

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

```
void main()
{
   (* Inicia n tarefas em paralelo *)
   parbegin (P(1), ... , P(n))
}
```



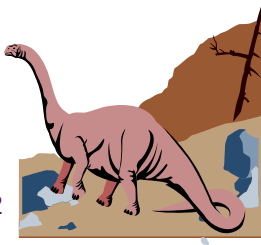


Impasse e Inanição
(deadlock e starvation)

- **Deadlock** – dois ou mais processos estão à espera indefinidamente por um evento que só pode ser causado por um dos processos à espera.
- 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);

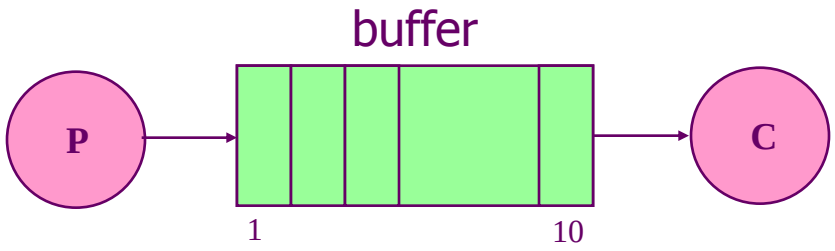
- **Starvation** – bloqueio indefinido. Um processo corre o risco de nunca ser removido da fila do semáforo, na qual ele está suspenso
- Semáforos não evitem, por si próprio, estes problemas !.



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;
```

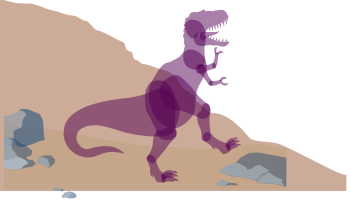


processo **produtor**

```
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--;
}
```

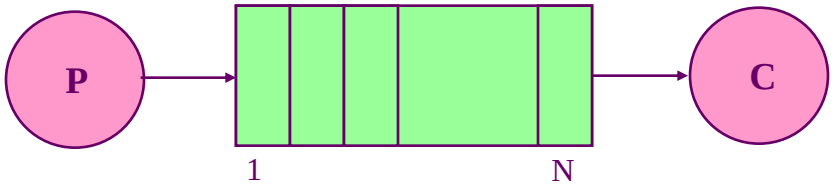


Semáforos

(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;
```

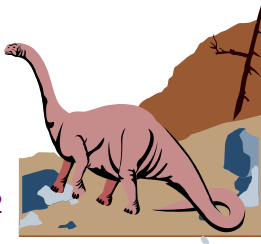
P

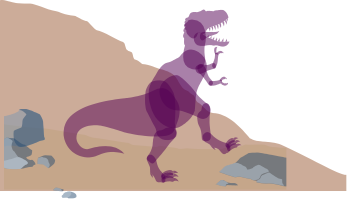
```
void Produtor ()
{
   int dado;

   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)
}
```

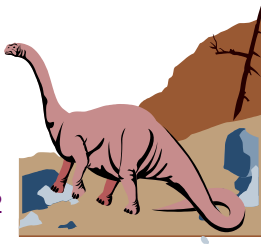


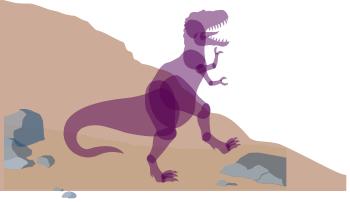


Producer Consumer

- 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





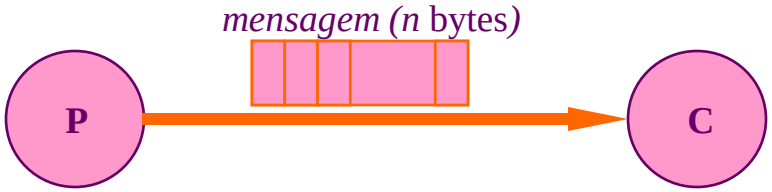
Passagem de mensagens (endereçamento directo)

O mecanismo de passagem de mensagens permite implementar simultaneamente:

- **sincronização** de processos
- **comunicação** entre processos

O **comunicação** entre processos pode ser sempre reduzido à interacção entre um produtor e um consumidor de informação.

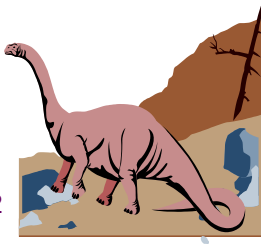
No caso de a **comunicação** entre processos ser feita através de mensagens, o produtor é o remetente ou **emissor** da mensagem e o consumidor é o destinatário or **receptor** da mensagem.

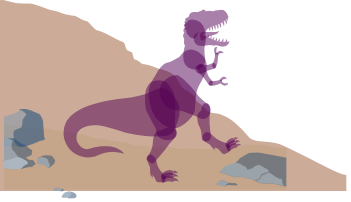


Primitivas de endereçamento DIRECTO:

send (receptor, mensagem)

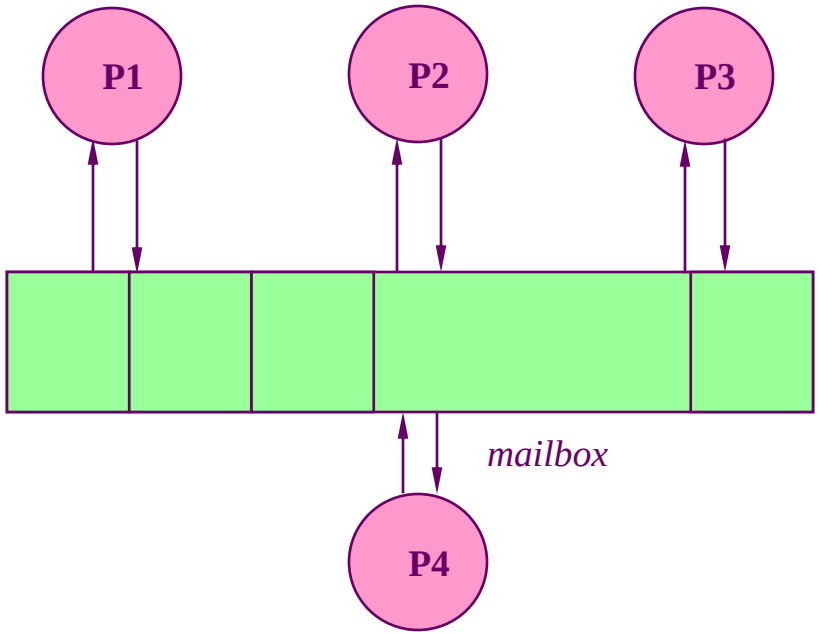
receive(emissor, mensagem)





Passagem de mensagens (endereçamento indirecto)

Através deste método os processos não comunicam directamente, mas sim através de **mailboxes** residentes no núcleo do S.O., i.e. **filas de mensagens** residentes no núcleo. As mailboxes permitem a existência de vários receptores.



Ordenação da mailbox:

FIFO, mas se algumas mensagens forem urgentes torna-se necessário definir prioridades.

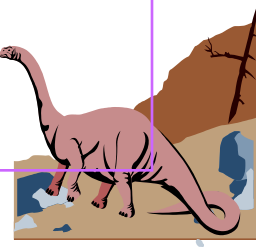
Formato das mensagens:

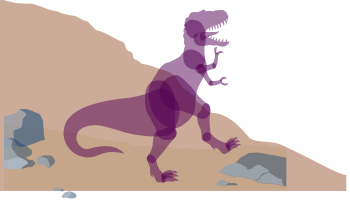
- tipo
- comprimento
- dados

Primitivas de endereçamento INDIRECTO:

send (mailbox, mensagem)

receive(mailbox, mensagem)



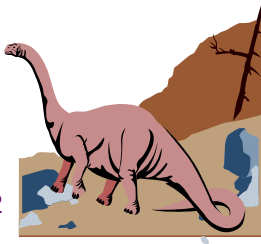


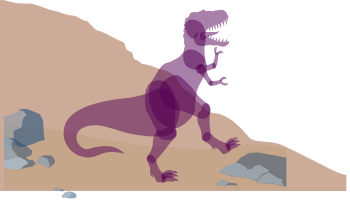
Passagem de mensagens

(exclusão mútua através de mailbox)

Esboço do algoritmo:

- Um processo pretende entrar numa secção crítica, tentando receber uma mensagem;
- Se a mailbox estiver vazia, então o processo fica bloqueado.
- Logo que o processo obtém a mensagem, então executa a sua secção crítica;
- Reenvia a mensagem de volta para a mailbox.



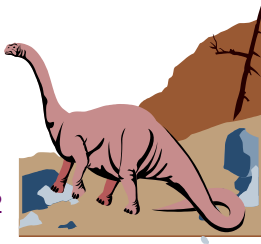


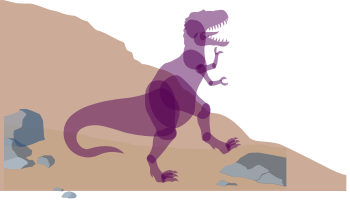
Passagem de mensagens

(capacidade da mailbox)

Uma **mailbox** (canal de comunicação) tem uma **capacidade** de armazenamento que determina o número de mensagens que pode comportar temporariamente:

- **Capacidade zero**
  - não pode haver mensagens armazenadas
  - o emissor tem de esperar pela disponibilidade do receptor
  - os processos têm de sincronizar em qualquer troca de mensagens
- **Capacidade limitada**
  - tamanho finito
  - se a fila não estiver cheia, não há necessidade de bloquear o emissor
  - o sucesso do envio não implica o sucesso da recepção
- **Capacidade ilimitada**
  - tamanho infinito
  - o emissor nunca é bloqueado
  - o sucesso do envio não implica o sucesso da recepção

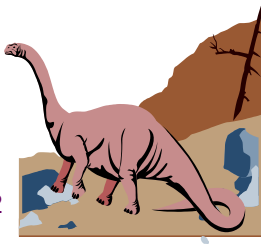


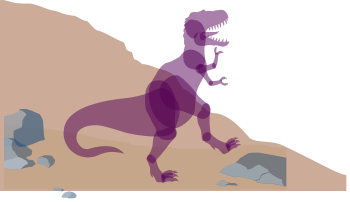


Passagem de mensagens
(sincronização)

A primitiva de envio de mensagem pode ter vários modos de funcionamento:

- **Assíncrona** (a mais vulgar)
  - o emissor envia um pedido e continua a execução
- **Síncrona**
  - o emissor fica bloqueado até recepção da mensagem
- **Cliente/Servidor**
  - o emissor fica bloqueado até o receptor lhe responder





Passagem de mensagens
(condições de excepção)

- Um sistema de mensagens é particularmente útil num ambiente distribuído, onde os processos podem residir em computadores distintos.
- Uma falha num processo não implica necessariamente a falha de todo o sistema.
- Quando ocorre uma avaria, algum processo de recuperação de erro deve ser executado.

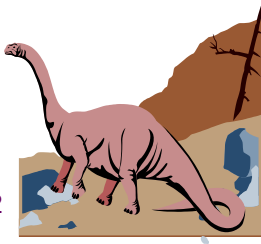
Terminação anormal dum processo. Dois casos:

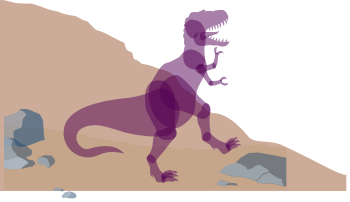
- O processo P poderá bloquear indefinidamente à espera duma mensagem do processo Q que já terminou.

  Solução: o sistema termina também o processo P ou notifica-o da terminação de Q.

- O processo P envia uma mensagem ao processo Q que já terminou.

  Solução: Se a mailbox tiver capacidade zero, P bloqueia indefinidamente, caindo no caso anterior; caso contrário, P poderá continuar a sua execução.





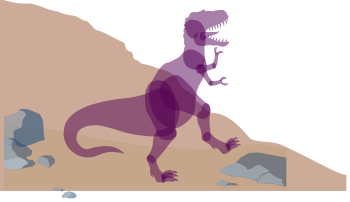
Passagem de mensagens
(condições de excepção)

Mensagem perdida de P para Q.

Soluções:

- cabe ao sistema operativo detectar a situação e reenviar a mensagem.
- cabe ao processo emissor P detectar a situação e reenviar a mensagem.
- cabe ao sistema operativo detectar a situação, notificando o processo emissor P, ao que P actuará conforme entender (reenviando ou não a mensagem).

Fim de Capítulo



Outros Mecanismos de IPC

- Outros mecanismos de comunicação entre processos com ou sem **sincronização e comunicação** entre os processos
- Pipeta (pipes)
  - Sincronização e Comunicação usando o Kernel
- Shared Memory (Memoria Partilhada)
  - Comunicação (rapida !) de dados mas sem sincronização
- Signals
  - Permitem sincronização - Comunicação do acontecimento dum evento Mas não comunicam dados/inormação