DEADLOCKS *

- O problema dos *deadlocks*
- Condições necessárias para a sua ocorrência
- Métodos de tratamento dos *deadlocks*

(*) *Deadlock* - impasse; bloqueio fatal; bloqueio permanente

O problema dos *deadlocks*

- Vários processos, executando concorrentemente, competem pelos mesmos recursos:
  - dispositivos físicos (ex: impressora, espaço de memória, ...)
  - dispositivos lógicos (ex: secção crítica, ficheiro, ...)
- Quando um processo detém um recurso, os outros têm de esperar.
- Em certas circunstâncias, o sistema pode encravar e nenhum processo pode avançar (*deadlock* ou bloqueio fatal)
- Os recursos alocados a processos encravados não são utilizáveis até que o *deadlock* seja resolvido.

Exemplos - possibilidade de *deadlock*

Exemplo 1

- Sistema com 1 disco + 1 *tape*
- 2 processos competindo pelo uso exclusivo destes recursos
- Acontece um *deadlock* se cada processo obtiver um recurso e requisitar o outro

```
Process P1;
 Begin
   ...
   Request(D);
   Request(T);
   ...
   ...
   Release(T);
   Release(D);
   ...
 End;
```

```
Process P2;
 Begin
   ...
   Request(T);
   Request(D);
   ...
   ...
   Release(D);
   Release(T);
   ...
 End;
```

Exemplo 2

- 2 processos utilizando 2 semáforos *Mutex*, S e Q
- Acontece um *deadlock* se a ordem de execução for, por ex.:

```
P1 - Wait(S)
P2 - Wait(Q)
P2 - Wait(S)
```

```
Process P1;
 Begin
   ...
   Wait(S);
   Wait(Q);
   ...
   ...
   Signal(Q);
   Signal(S);
   ...
 End;
```

```
Process P2;
 Begin
   ...
   Wait(Q);
   Wait(S);
   ...
   ...
   Signal(S);
   Signal(Q);
   ...
 End;
```

Exemplos

Exemplo 3

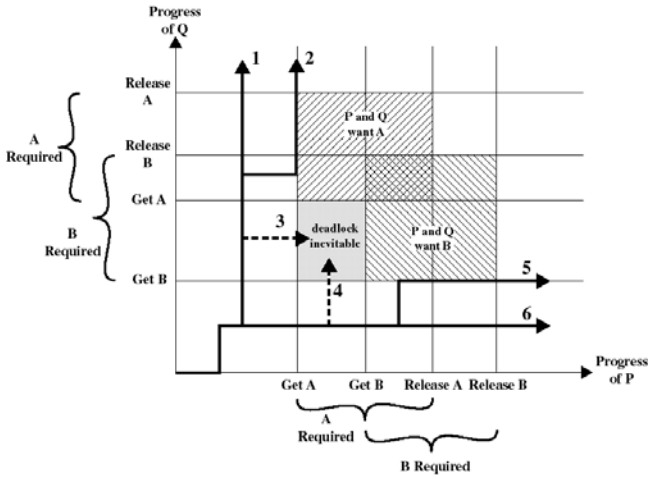
- 2 processos utilizando que comunicam entre si através de 2 filas de mensagens
- acontece *deadlock* se a operação Receive bloquear quando não há mensagens na fila

```
Process P1;
 Begin
   ...
   Receive(P2);
   ...
   ...
   Send(P1);
   ...
 End;
```

```
Process P2;
 Begin
   ...
   Receive(P1);
   ...
   ...
   Send(P2);
   ...
 End;
```

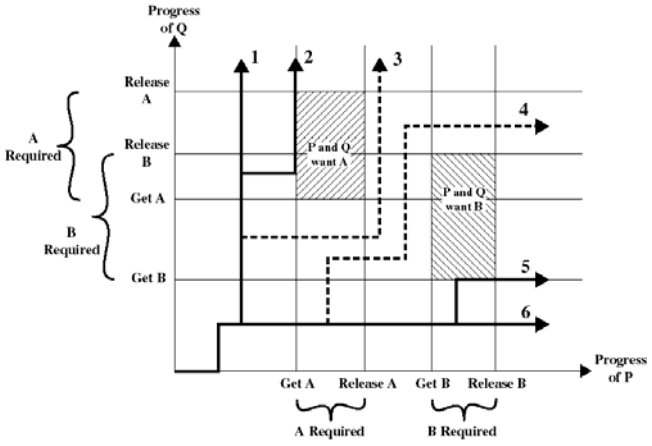
Exemplo: possibilidade de *deadlock*

```
Processo P;
…
Get(A);
…
Get(B);
…
Release(A);
…
Release(B);
…

Processo Q;
…
Get(B);
…
Get(A);
…
Release(B);
…
Release(A);
…
```



Exemplo: impossibilidade de *deadlock*

```
Processo P;
…
Get(A);
…
Release(A);
…
Get(B);
…
Release(B);
…

Processo Q;
…
Get(B);
…
Get(A);
…
Release(B);
…
Release(A);
…
```



Definição

- ***Deadlock*** **:**
  - **bloqueio permanente de um conjunto de processos que competem por recursos do sistema ou comunicam entre si.**

***Deadlock** versus **starvation*:**

- ***Deadlock*** **(bloqueio fatal)** ***:***
  - **esperar indefinidamente por alguma coisa que não pode acontecer.**
- ***Starvation*** **(inanição) :**
  - **esperar muito tempo por alguma coisa que pode nunca acontecer.**

Atribuição e utilização de recursos

- **O sistema operativo força uma utilização adequada dos recursos reutilizáveis fornecendo serviços para a sua requisição / utilização / libertação.**
  - ***Request***
    - **Geralmente, formas de abrir (*open*) ou alocar (*alloc*) um recurso.**
    - **Bloqueia o processo até que recurso seja concedido.**
      (Nem sempre ! Pode negar o recurso e informar o processo.
      ex: o *open* de um ficheiro retorna erro se o ficheiro não puder ser aberto)
  - ***Use***
    - **Serviços especiais para o uso do recurso (ex: *read* e *write* de ficheiros).**
  - ***Release***
    - **Formas de fechar (*close*) ou libertar (*free*) um recurso.**
    - **O S.O. pode então conceder o recurso a outro processo.**

Atribuição e utilização de recursos

- Outro tipo de recursos, os chamados <u>recursos consumíveis</u> (recursos que podem ser criados e destruídos), são criados pelos processos e partilhados por eles, geralmente em exclusão mútua:
  - Mensagens
  - Sinais
  - Semáforos
- Certas combinações de acontecimentos <u>podem</u> produzir *deadlocks*.
  - Ex:

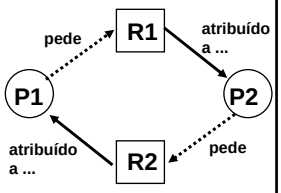
se o *Receive* de mensagens se fizer com bloqueio

```
Process P1;
 Begin
   ...
   Receive(P2);
   ...
   ...
   Send(P1);
   ...
 End;
```

```
Process P2;
 Begin
   ...
   Receive(P1);
   ...
   ...
   Send(P2);
   ...
 End;
```

Condições necessárias para a ocorrência de um *deadlock*

- <u>Exclusão mútua</u>
  - Só um processo pode usar um recurso de cada vez.
- <u>Retém e espera</u>
  - Um processo pode deter recursos enquanto está à espera que lhe sejam atribuídos outros recursos.
- <u>Não preempção dos recursos</u>
  - Quando um processo detém um recurso só ele o pode libertar.
- <u>Espera circular</u>
  - Deve existir um conjunto de processos $\{P_1, P_2, ..., P_n\}$ tal que P_1 está a espera de um recurso que P_2 detém, P_2 está a espera de um recurso que P_3 detém, ... , P_n está a espera de um recurso que P_1 detém.



Condições para *deadlock*

- **O *deadlock* ocorre se e só se a condição de espera circular não tiver solução.**
- **A condição de espera circular não tem solução quando as 3 primeiras condições se verificam.**
- **As 3 primeiras condições são necessárias mas não suficientes para que ocorra uma situação de *deadlock*.**
- **Por isso, as 4 condições tomadas em conjunto constituem condições necessárias e suficientes para um *deadlock*.**

Métodos de tratamento dos *deadlocks*

- **Prevenir (*prevent*)**
  - **Assegurar que pelo menos 1 das 4 condições necessárias não se verifica.**
- **Evitar (*avoid*)**
  - **Não conceder recursos a um processo, se essa concessão for susceptível de conduzir a *deadlock*.**
- **Detectar e recuperar**
  - **Conceder sempre os recursos enquanto existirem disponíveis; periodicamente, verificar a existência de processos encravados e, se existirem, resolver a situação.**

Alternativa (por parte do S.O.): ignorar os *deadlocks*

Prevenir os *deadlocks*

Assegurar que pelo menos uma das 4 condições não se verifica.

- **Exclusão mútua**
  - **Solução: usar só recursos partilháveis …!**
  - **Problema:**
    - **certos recursos têm de ser usados com exclusão mútua.**
  - **A utilização de *spooling* (ex: da impressora) ajuda a prevenir esta condição**

- **Retém e espera**
  - **Solução: Garantir que quando um processo requisita um recurso não detém nenhum outro recurso ⇒**
    - **Requisitar todos os recursos antes de começar a executar ou**
    - **Requisitar os recursos incrementalmente, mas libertar os recursos que detém quando não conseguir requisitar os recursos de que precisa.**
  - **Problemas:**
    - **Sub-utilização dos recursos.**
    - **Necessidade de conhecimento prévio de todos os recursos necessários. (não faz sentido em sistemas interactivos)**
    - **Possibilidade de inanição.**

Prevenir os *deadlocks*

- **Não preempção de recursos**
  - **Solução: Permitir a preempção de recursos. Q.do a um processo é negado um recurso deverá libertar todos os outros, *ou* o processo que detém esse recurso deverá libertá-lo.**
  - **Problema: só é aplicável a recursos cujo estado actual pode ser guardado e restaurado facilmente (ex.: memória e registos da *CPU*)**

- **Espera circular**
  - **Solução: Protocolo para impedir espera circular; os vários tipos de recursos são ordenados e e os processos devem requisitá-los por essa ordem**
    - **Ex. 1-*tapes* ; 2-ficheiros ; 3-impressoras**
    - **O processo deve requisitar os recursos sempre pela mesma ordem.**
    - **Se já requisitou ficheiros, então, só pode requisitar a impressora.**
    - **Se o processo necessitar de várias instâncias do mesmo recurso deve requisitá-las de uma só vez.**

Prevenir os *deadlocks*

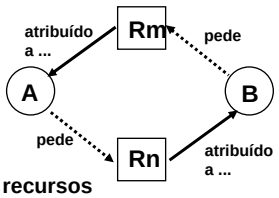
◆ **Espera circular** (cont.)

■ **Demonstração que o protocolo funciona (p/2 processos)**

- **Pressuposto: o recurso Ri precede Rj, se i<j**
- **Admitamos que 2 processos A e B estão encravados porque**

o processo A possui Rm e requisitou Rn ⇒ m<n
o processo B possui Rn e requisitou Rm ⇒ n<m

É impossivel que (m<n) e (n<m) !!!
(demonstração por contradição)



A já tem Rm **Se m>n, A não pode requisitar Rn**
B já tem Rn **Se n<m, B não pode requisitar Rm**
Logo, nunca se pode fechar o ciclo.

■ **Problemas**

- **Ineficiência devido à ordenação imposta aos recursos**
  - **os recursos têm de ser requisitados por uma certa ordem em vez de serem requisitados à medida que são precisos.**
  - **certos recursos são negados desnecessariamente**
- **Difícil encontrar uma ordenação que funcione.**

Evitar os *deadlocks*

◆ **Prevenir**

■ **Evitar os *deadlocks* indirectamente, impedindo que uma das 4 condições se verifique.**

◆ **Evitar**

■ **Permitir que aquelas condições se verifiquem, e decidir, perante cada pedido de recursos, se ele pode conduzir a um *deadlock*, caso os recursos sejam atribuídos. Se sim, negar a atribuição dos recursos pedidos.**

■ **⇒ Examinar dinamicamente o estado de alocação de recursos para assegurar que não vai ocorrer uma situação de espera circular.**

◆ **Duas estratégias p/ evitar *deadlocks*:**

■ **Não começar a executar um processo se as suas necessidades, juntamente c/ as necessidades dos que já estão a correr, forem susceptíveis de conduzir a um *deadlock*.**

■ **Não conceder um recurso adicional a um processo se essa concessão for susceptível de conduzir a um *deadlock*.**

Evitar os *deadlocks*

Notação:

- `n` - nº de processos
- `m` - nº de classes de recursos
- `Available[1..m]` - quantidade de recursos de cada classe disponíveis num determinado instante
- `Max[1..n,1..m]` - necessidades máximas de cada processo relativamente aos diferentes recursos
- `Allocation[1..n,1..m]` - número de recursos de cada classe atribuídos a cada processo
- `Need[1..n,1..m]` - necessidades que falta satisfazer
  `Need[i,j]= Max[i,j]-Allocation[i,j]`

- Se `X` e `Y` são vectores de comprimento n
  - Diz-se que `X<=Y` se `X[i]<=Y[i]`, para `i=1..n`
  - Diz-se que `X<Y` se `X<=Y` e `X<>Y`
- Se `M` é uma matriz
  - M_i representa a linha `i` da matriz

Evitar os *deadlocks*

- Os principais algoritmos para evitar *deadlocks* são baseados no conceito de estado seguro.
- Um estado diz-se seguro se o sistema conseguir alocar recursos a cada processo, por uma certa ordem, de modo a evitar *deadlocks*.
- Evitar ⇒ assegurar que o sistema nunca entra num estado inseguro (estado que pode conduzir a *deadlock*).

- **1ª estratégia**
  - O início de execução de um novo processo é negado se as máximas necessidades de todos os processos em execução mais as necessidades deste novo processo excederem a quantidade de qualquer classe de recurso
  - Esta estratégia é demasiado restritiva.

Evitar os *deadlocks*

- **2ª estratégia**
  - **Não conceder um recurso adicional se essa concessão for susceptível de conduzir a um *deadlock*.**
- **Algoritmo do banqueiro / teste de estado seguro (*Dijkstra*)**
  - **1. Vectores**
    - **Work[1..m]** - quant. de recursos disponíveis em cada instante da simulação
    - **Finish[1..n]** - indica se o proc. i tem possibilidade de obter os recursos que precisa

    Inicializar:
    ```
    Work := Available;    {recursos disponíveis no instante da chamada }
    Finish[i] := False, para i=1..n;
    ```
  - **2. Encontrar um processo i, tal que**
    (Finish[i]=False) and ($Need_i$<=Work);
    Se não existir tal i, saltar para 4. {não existe se condição ant. = False}
  - **3. Work := Work + $Allocation_i$** {se entrar aqui significa que encontrou um processo}
    Finish[i] := True; { que tem possibilidade de terminar.}
    Saltar para 2; { Após terminar, os seus recursos são libertados; daí o sinal + }
  - **4. Se Finish[i]=True, para i=1..n**
    então o sistema está num estado seguro.

ou, de outra forma:

```
C = {conjunto de todos os processos};
While (C != CONJUNTO_VAZIO) {
  Procurar um P, elemento de C, que possa terminar;
  Se não existir nenhum P {
      o estado é INSEGURO;
      terminar; }
  senão {
      remover P de C;
      adicionar os recursos de P aos rec.s disponíveis; }
}
O estado é SEGURO;
```

Evitar os *deadlocks*

- **Algoritmo de requisição de recursos**
  - $Request_i$ - vector que representa as necessidades do processo P_i
  - 1. Se $Request_i <= Need_i$, saltar para 2,
    senão assinalar erro(P_i excedeu os limites que tinha declarado)
  - 2. Se $Request_i <= Available$, saltar para 3
    senão P_i tem de esperar, dado que os recursos não estão disponíveis.
  - 3. Simular a alocação de recursos ao processo P_i

    $Available := Available - Request_i$;
    $Allocation_i := Allocation_i + Request_i$;
    $Need_i := Need_i - Request_i$;

    Se o estado resultante for seguro {--> ALGORITMO DO BANQUEIRO}
    a transacção é completada e o processo P_i recebe os recursos.

    Se o estado resultante for inseguro,
    P_i tem de esperar por $Request_i$ e
    o estado de alocação anterior é restaurado.
- **Algoritmo de libertação de recursos**
  - Quando um recurso é libertado
    actualizar o vector Available e
    reconsiderar os pedidos pendentes para esse recurso, se os houver.

Evitar os *deadlocks*

- **Vantagens (algoritmo do banqueiro):**
  - **Menos restritivo do que a prevenção.**
  - **Não requer a requisição simultânea de todos os recursos necessários.**
  - **Não obriga à preempção dos recursos.**
- **Dificuldades**
  - **Necessidade de conhecimento antecipado de todos os recursos necessários**
    ⇒ utilidade prática limitada.
  - ***Overhead* necessário para detectar os estados seguros.**

Exemplo

- 5 processos (P0..P4)
- 3 tipos de recursos
  - A (10 instâncias)
  - B (5 instâncias)
  - C (7 instâncias)

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

- O sistema está actualmente num estado seguro porque a sequência P1, P3, P4, P2, P0 satisfaz o critério de segurança

Exemplo

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

3 0 2 → simulação da alocação de recursos

0 2 0 → simulação da alocação de recursos

2 3 0

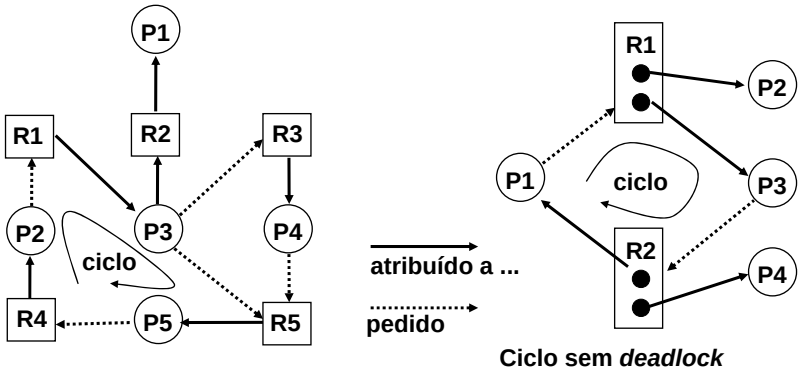
- Quando P1 faz o pedido → $Request_1=(1,0,2)$
  - Verificar que $Request_1$ <= Available → (1,0,2) <= (3,3,2) ? → True
  - A execução do teste de segurança mostra que a sequência P1, P3, P4, P0, P2 também satisfaz o critério de segurança
  - Por isso, o pedido pode ser atendido.

Detecção e Recuperação

- **Os recursos são concedidos se estiverem disponíveis. Periodicamente detecta-se a ocorrência de *deadlocks*. Se existir *deadlock*, aplica-se uma estratégia de recuperação.**
- **Quando fazer a detecção ?**
  - **Sempre que é concedido um novo recurso ⇒ *overhead* elevado.**
  - **Com um período fixo.**
  - **Quando a utilização do processador é baixa.**
- **Como proceder à recuperação ?**
  - **Avisar o operador e deixar que seja ele a tratar do assunto.**
  - **O sistema recupera automaticamente**
    - **Abortando alguns processos envolvidos numa espera circular ou**
    - **Fazendo a preempção de alguns recursos.**

Detecção

- **<u>Método 1</u>: quando há uma única instância de cada tipo de recurso**
  - **Manter um grafo de processos e recursos *(wait for graph).***
  - **Periodicamente, invocar um algoritmo de detecção de ciclos em grafos.**



Ciclo com *deadlock*

Ciclo sem *deadlock*
(há mais do que uma instância de cada recurso)

Detecção

- **Método 2: quando há várias instâncias de cada tipo de recurso.**
- **Estruturas de dados**
  - `Available [1..m]` `Finish[1..n]`
    `Allocation [1..n,1..m]` `Work[1..m]`
    `Request[1..n,1..m]`
- **Algoritmo**
  - `1. Inicializar Work := Available;`
    `Para i:=1 até n`
    `Se Allocation`$_i$ `<> 0 então Finish[i] := False {Pi pode estar encravado}`
    `senão Finish[i] := True;`
  - `2. Encontrar um i tal que`
    `(Finish[i] = False) e (Request`$_i$`<=Work);` `{Pi pode terminar}`
    `Se não existir tal i, saltar para 4.`
  - `3. Work := Work + Allocation`$_i$`;` `{Quando terminar,`
    `Finish[i] := True;` `libertará os recursos}`
    `Saltar para 2;`
  - `4. Se Finish[i]=False para qualquer i, 1 <= i<= n,`
    `o sistema está num estado de` ***deadlock***`.`
    `Além disso, se Finish[i]=False, o processo Pi está encravado.`

Recuperação

Alternativas

- **Terminação de processos**
- **Preempção de recursos**

- **Terminação de processos**
  - **Abortar todos os processos encravados.**
  - **Abortar sucessivamente um processo até eliminar o *deadlock*.**
    - **Por que ordem ?**
      Factores a ter em conta
      - **prioridade dos processos**
      - **tempo de computação passado (e futuro ...? → estimado)**
      - **recursos usados**
      - **recursos necessários para acabar**
      - **tipo de processo (interactivo ou *batch*)**
      - **...**
  - **Correr o algoritmo de detecção após cada terminação de um processo.**

Recuperação

- **Preempção de recursos**
  - **Retirar sucessivam. recursos aos processos até desfazer o *deadlock* .**
  - **Questões:**
    - **Que recursos e que processo seleccionar ?**
      - **Factores de custo:**
        - nº de recursos detidos pelos processos encravados;
        - tempo de computação que os processos já usaram.
    - **Que fazer com o processo a quem foram retirados os recursos ?**
      - **Fazer o *rollback***
        - Retornar o processo a um estado seguro e continuar a partir daí (difícil!)
        - Abortar o processo e recomeçar de início.
    - **Como evitar a inanição de um processo, isto é, que seja sempre o mesmo processo a ser seleccionado como "vítima" ?**
      - **Tomar nota do nº de *rollbacks* no factor de custo.**

Estratégia integrada

- **Nenhum dos métodos analisados anteriormente é adequado para todos os tipos de problemas de alocação de recursos.**
- **Solução: combinar os 3 métodos, partindo os recursos em classes, e seleccionar o método mais adequado para cada classe.**
- **Exemplo:**
  - **Espaço de *swap*, em disco (*swappable space*)**
    - ***Prevenir* a condição de retém e espera:**
      todo o espaço de *swap* em disco deve ser requisitado de uma única vez.
  - **Memória de dados ou código**
    - ***Prevenir* a condição de não preempção:**
      quando não há memória suficiente no sistema para a próxima alocação, um ou mais processos são *swapped* para disco libertando assim a memória.
  - **Recursos internos do SO**
    - ***Prevenir* a condição de espera circular**
      através da ordenação dos recursos e da requisição e alocação por essa ordem.
  - **Recursos dos processos (dispositivos de *I/O*, ficheiros, ...)**
    - **Evitar o *deadlock*:**
      o processo indica à partida os recursos que necessita ...

Ignorar os *deadlocks*

- **Aproximação usada em muitos sistema operativos, incluindo o UNIX.**
- **Considera-se que, é preferível que ocorra um *deadlock*, de vez em quando, do que estar sujeito ao *overhead* necessário para os evitar/detectar.**
- **O UNIX limita-se a negar os pedidos se não tiver os recursos disponíveis.**
- **Alguns sistemas (ex: VMS) iniciam um temporizador sempre que um processo bloqueia à espera de um recurso. Se o pedido continuar bloqueado ao fim de um certo tempo, é então executado um algoritmo de detecção de *deadlocks*.**
- **Os *deadlocks* ocorrem essencialmente nos processos do utilizador, não nos processos do sistema.**