Sistemas Operativos: *Deadlocks*

Pedro F. Souto (pfs@fe.up.pt)

March 30, 2012

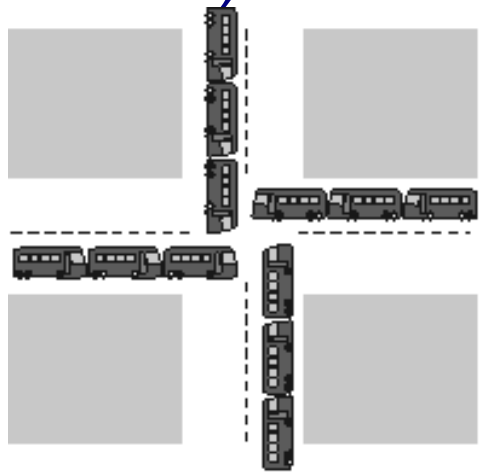
Deadlocks

- Um ***deadlock*** é uma situação em que 2 ou mais processos ficam bloqueados indefinidamente
  - pode ser uma *race condition*
- Um exemplo é a seguinte solução do *bounded buffer* com semáforos:

```
public void remove(Object item) {
    mutex.down();       // keep other threads out
    empty.down();       // wait for some item
```

  que difere da solução correcta na ordem de execução das operações `down()`:
  - Se o *buffer* estiver vazio, o consumidor bloqueia sem *libertar* o *mutex*, impedindo o produtor de eventualmente introduzir a mensagem no *buffer*.
- Outro exemplo é o problema dos *dining philosophers*
  - Processo tem que aceder a mais de 1 recurso partilhado cujo acesso deve ser feito em exclusão mútua

Bloqueio Mútuo (Deadlock)



Definição um conjunto de processos está **mutuamente bloqueado (*deadlocked*)**, se cada processo nesse conjunto está à espera dum evento que apenas outro processo nesse conjunto pode causar.

- Normalmente, o evento pelo qual um processo espera é a libertação dum recurso na posse de outro processo nesse conjunto
  - Numa situação de *deadlock*, nenhum processo pode:
    - executar;
    - libertar os recursos que detém;
    - ser activado.

Recursos

- O estudo do problema de *deadlock* tem origem em SOs, no contexto da gestão de *recursos* dum computador.
- Um *recurso* é um "objecto" usado por um processo. P. ex.:
  - físico: disco, memória, CPU, ...;
  - lógico: um *socket*, um ficheiro, uma secção crítica, ...;
- Num sistema, pode haver mais do que um recurso do mesmo tipo, p.ex. disco.
- Propriedades importantes de recursos:

  tipo de acesso certos recursos, p.ex. uma impressora, têm que ser acedidos em exclusão mútua;

  preemptibilidade recursos podem ser ou não preemptíveis: isto é, podem ser ou não retirados a um processo sem "problemas de maior". Por exemplo, uma impressora não é preemptível, mas o CPU é-o.

Uso de Recursos por Processos

- Recursos são geridos por *gestores*, os quais podem ser ou não componentes do SO.
- O uso de um recurso por um processo tipicamente envolve a sequência:
  - pedido do recurso;
  - uso do recurso;
  - libertação do recurso.
- Se o recurso pedido por um processo não estiver disponível, o processo não o pode usar. Neste caso, o processo:
  - ou bloqueia, à espera que o recurso fique disponível;
  - ou prossegue, mas não terá acesso ao recurso pedido.

Condições *Necessárias* para *Deadlock*

Bloqueio á espera que o recurso fique disponível (*exclusão mútua*);

Espera com retenção processos não libertam os recursos na sua posse quando bloqueiam à espera de outros recursos;

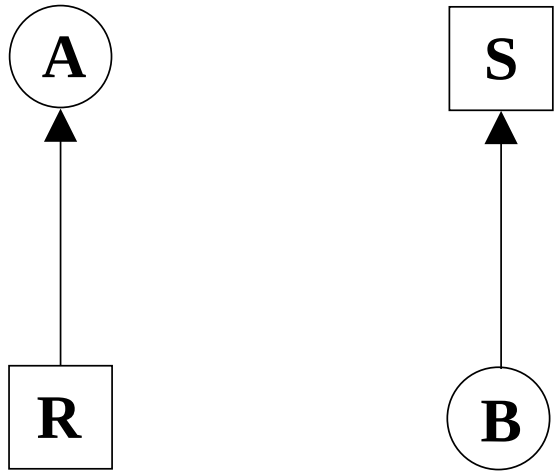
Não-preempção recursos na posse dum processo não lhe podem ser retirados;

Espera-circular tem que haver uma cadeia de 2 ou mais processos, cada um dos quais à espera dum recurso na posse do processo seguinte.

Sumário

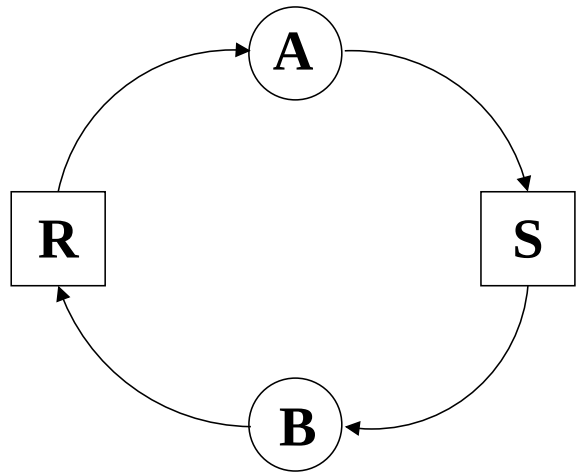
Modelação: Grafos de Alocação de Recursos

- Um *grafo* consiste num conjunto de nós interligados por arcos.
- Um grafo de alocação de recursos tem 2 tipos de nós:
  - processos (círculos);
  - recursos (quadrados).
- Um arco **dum** recurso **para um** processo, indica que o recurso está na posse do processo;
- Um arco **dum** processo **para um** recurso, indica que o processo está bloqueado à espera dum recurso.



Grafos de Alocação e *Deadlocks*

- **Em sistemas com apenas 1 exemplar de cada tipo de recurso**, se um grafo de alocação tiver um ciclo, então há *deadlock*:



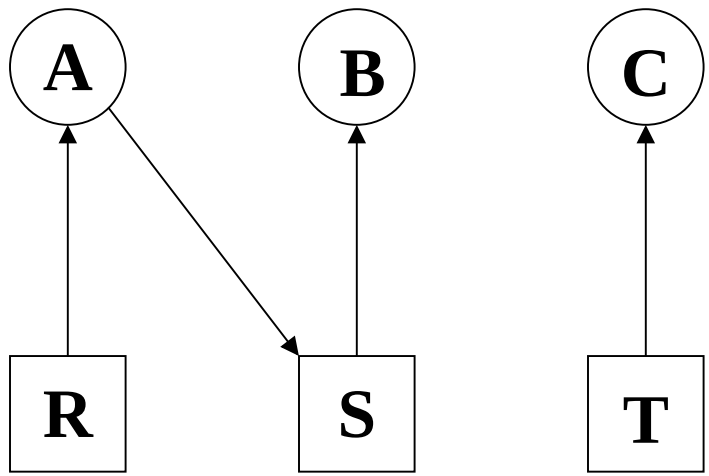
- Há generalizações do grafo de recursos para detetar bloqueio mutúo quando existem mais de 1 exemplar de cada tipo.

Exemplo: Grafos de Alocação e *Deadlocks*

- Seja a seguinte evolução temporal dos requisitos de recursos de 3 processos:

Proc. A	Proc. B	Proc. C
req. R	req. S	req. T
req. S	req. T	req. R
lib. R	lib. S	lib. T
lib. S	lib. T	lib. R

- Admite-se que:
  - os processos executam em *round-robin*;
  - há comutação de processos imediatamente após um processo pedir o recurso.



Estratégias de Ataque ao Problema de *Deadlock*

1. Prevenir

   avoidance solução dinâmica
   prevention solução estática
2. Detetar e recuperar
3. Ignorar o problema: algoritmo da avestruz (Tanenbaum)

Sumário

Algoritmo da Avestruz

- Ignorar o problema: “pretender” que não existe.
- Pode ser aceitável.
  - se a probabilidade de *deadlock* fôr *muito baixa*;
  - o custo de qualquer das outras soluções fôr *elevado*.
- Unix tenta evitar situações de *deadlock*, mas não as elimina completamente.
- É um compromisso entre:
  - correcção;
  - conveniência/eficiência;

  análogo a muitos em engenharia.

. . . viver é gerir riscos.

Sumário

Detetar e Recuperar

- Tem 2 partes:
  1. Detetar o *deadlock*:
     - O algoritmo basicamente testa se há alguma ordem de terminação dos processos.
  2. Recuperar da situação:
     - preempção de recursos;
     - *roll-back* de processos;
     - terminação de processos.

Deteção de *Deadlock*

- Faz uso das seguintes estruturas de dados:

  Vector dos recursos existentes, E, tem m elementos, i.e. tantos quantos os tipos de recursos. O valor de cada elemento é o número de recursos do tipo correspondente existentes no sistema.

  Vector de recursos disponíveis, D, com m elementos.

  Matriz dos recursos atribuídos, A, de dimensão $n \times m$, em que n é o número de processos. A linha i é o vector com os recursos atribuídos ao processo P_i.

  Matriz dos pedidos pendentes, P, de dimensão $n \times m$. A linha i é o vector com os pedidos pendentes de recursos pelo processo P_i.

Exemplo de Deteção de *Deadlock*

$$E = \begin{bmatrix} 4 & 2 & 3 & 1 \end{bmatrix} \quad D = \begin{bmatrix} 2 & 1 & 0 & 0 \end{bmatrix}$$

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \quad P = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- Testar se há uma ordem de terminação dos processos.
- P_1 e P_2 estão bloqueados: os seus pedidos não podem ser satisfeitos.
- Mas P_3 pode executar e se terminar libertará os seus recursos.
- Depois os pedidos de P_2 podem ser satisfeitos e P_2 poderá terminar, libertando os seus recursos.
- Finalmente, os pedidos de P_1 poderão ser satisfeitos e P_1 poderá terminar: **não** há *deadlock*.

Algoritmos de Deteção

- Os algoritmos de deteção são “caros”: $O(m \times n^2)$.

  De facto, a complexidade dum algoritmo não é a história completa: há constantes a considerar e os valores dos parâmetros – se m ou n forem pequenos o custo pode ser baixo.
- Quando devem ser executados?
  - sempre que é feito um pedido;
  - periodicamente;
  - quando a utilização do CPU diminui.

Recuperação de *Deadlock* (1/2)

Preempção de recursos:

- depende do recurso em causa:
  - não é sempre aplicável;
  - pode requerer intervenção humana,

Rollback de processos:

- fazer o *checkpoint* do estado dos processos, periodicamente;
- em caso de *deadlock*, identificar um processo que se *rolled-back* quebrará o *deadlock*.

Terminar um processo no ciclo: de preferência, dever-se-á terminar um processo que possa ser reexecutado de início sem problemas de maior.

Recuperação de *Deadlock* (2/2)

- Em qualquer dos casos, é preciso escolher uma vítima. Alguns critérios:
  - o processo que possui menos recursos;
  - o processo que possui mais recursos;
  - o processo que usou menos o CPU.
- Pode ainda, usar-se o seguinte critério:
  - o menor conjunto de processos que quebrará o *deadlock*.
- Com excepção de *roll-back* as outras soluções podem deixar o sistema num estado *inconsistente*.
- Fazer o *checkpointing* do estado dos processos pode não ser suficiente para garantir *roll-backs* consistentes:
  - E se o processo *rolled-back* modificou um ficheiro desde o último *checkpoint*?

Sumário

Estados Seguros e Não-seguros

- Admitamos que 3 processos partilham só um tipo de recurso, e que há 10 unidades desse tipo.

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

(e)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

Unsafe State

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

Deadlock Avoidance (Solução Dinâmica)

Ideia antes de satisfazer um pedido, averiguar se essa satisfação conduz a um estado não-seguro

estratégia pessimista em comparação com *deteção & recuperação*.

Requisito desta técnica:

- é necessário o conhecimento *a priori* dos requisitos máximos de cada tipo de recurso por cada processo.

Algoritmo para *Deadlock Avoidance*

- Faz uso das seguintes estruturas de dados:

  Vector dos recursos existentes, E, tem m elementos, i.e. tantos quantos os tipos de recursos. O valor de cada elemento é o número de recursos do tipo correspondente existentes no sistema.

  Vector de recursos disponíveis, D, com m elementos.

  Matriz dos recursos atribuídos, A, de dimensão $n \times m$, em que n é o número de processos. A linha i é o vector com os recursos atribuídos ao processo P_i.

  Matriz dos recursos a atribuir, R, de dimensão $n \times m$. A linha i é o vector com os pedidos de recursos que o processo P_i ainda poderá fazer:

- O algoritmo é semelhante ao apresentado para deteção de *deadlock*, excepto em vez da matriz P usa-se a matriz R

Exemplo de *Deadlock Avoidance*

- Seja após a satisfação **condicionada** dum pedido:

$$E = \begin{bmatrix} 6 & 3 & 4 & 2 \end{bmatrix} \quad D = \begin{bmatrix} 1 & 0 & 2 & 0 \end{bmatrix}$$

$$A = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

  1. P_4 pode terminar: $D = \begin{bmatrix} 2 & 1 & 2 & 1 \end{bmatrix}$;
  2. P_1 pode terminar: $D = \begin{bmatrix} 5 & 1 & 3 & 2 \end{bmatrix}$;
  3. P_2 pode terminar: $D = \begin{bmatrix} 5 & 2 & 3 & 2 \end{bmatrix}$;
  4. P_3 pode terminar: $D = \begin{bmatrix} 6 & 3 & 4 & 2 \end{bmatrix}$;

- O pedido pode ser satisfeito: **não** conduz a *deadlock* mesmo que os processos usem os recursos ao máximo.

Considerações sobre *Deadlock Avoidance*

- Um pedido de recursos só é satisfeito, se:
  - os recursos que ficarem disponíveis permitirem a terminação de todos os processos que detém pelo menos um recurso, mesmo que cada um destes processos use o número máximo de recursos correspondente
- Infelizmente, a utilidade deste algoritmo (*Banker's Algorithm*, de Dijkstra) é no melhor dos casos limitada:
  - normalmente, os processos não conhecem *a priori* as suas necessidades máximas.
- Possível excepção poderão ser alguns sistemas de tempo-real
  - Mas neste caso há algoritmos de escalonamento do CPU que garantem ausência de bloqueio mútuo

Prevenir *Deadlocks* (Solução Estática)

Ideia garantir que 1 das 4 condições necessárias nunca ocorre:

1. Exclusão mútua;
2. Espera com retenção;
3. Não-preempção;
4. Espera circular.

deadlock avoidance é uma solução dinâmica: o programa executa um algoritmo para determinar se a satisfação do pedido pode conduzir a *deadlock*.

deadlock prevention é uma solução estática: impede-se a ocorrência de *deadlock* por concepção do programa.

Prevenir *Deadlocks*: Não-exclusividade

Exclusão mútua, p.ex., recorrendo a processos auxiliares - *printer spooler*:

- nem todos os recursos se prestam a *spooling*;
- a contenção no acesso a outros recursos (p.ex. disco, no caso do *printer spooler*) pode também dar origem a *deadlocks*.

Princípio, em qualquer dos casos, deve-se:

- atribuir um recurso apenas quando necessário;
- minimizar o número de processos que partilham um recurso.

Prevenir *Deadlocks*: Espera sem Retenção

Espera com retenção de recursos:

- Pedir todos os recursos necessários antes de iniciar tarefa:
  - sofre do mesmo problema que *deadlock avoidance*.
- Alternativamente, um processo deverá libertar todos os recursos que possui, quando o pedido de um recurso conduziria a um bloqueio:
  - quando desbloqueado, o processo terá que readquirir todos os recursos de novo;
  - é uma variante da solução anterior, na qual os recursos necessários são descobertos dinamicamente.

Técnica usada por vezes ao nível do *kernel* do SO.

Prevenir *Deadlocks*: Preempção e Não-circularidade

Preempção dos recursos:

- retirar recursos a um processo (sem a sua cooperação) é inviável na maioria dos casos.

Espera circular pode ser evitada por vezes. Por exemplo:

- definindo uma ordem total para todos os tipos de recursos;
- impondo que o pedido dos recursos respeite essa ordem.

Infelizmente, nem sempre é viável.

Mais sobre Gestão de Recursos

- Na prática, usa-se uma combinação destas técnicas.

Míngua (*starvation*) de processos pode ocorrer também associada à gestão de recursos:

- quando o número de recursos disponível é inferior ao número de recursos pedido, o sistema tem que decidir a que processos atribuir os recursos disponíveis;
- se o algoritmo usado favorecer alguns processos em relação a outros, estes últimos poderão ser continuamente preteridos no acesso aos recursos e assim impossibilitados de fazer qualquer progresso.

- O escalonamento de processos é um problema de gestão dum recurso particular: o CPU.

Leitura Adicional

Sistemas Operativos

- Secção 6.4

Modern Operating Systems, 2nd. Ed.

- Secções 3.1 a 3.6 inclusivé, excepto Secção 3.2.2

Operating Systems Concepts, 7th. Ed.

- Cap. 7