Capítulo 7

Detecção Distribuída de Deadlock

Algoritmos de deadlock são caracterizados pelo tipo de implementação (centralizado ou distribuído), abordagem (prevenção, detecção, *avoidance*) e tipo de deadlock (recurso ou comunicação).

Vamos ver quatro algoritmos:

- Algoritmo **centralizado** de **prevenção de deadlock de recurso**
- Algoritmo distribuído de **prevenção de deadlock de recurso**
- Algoritmo distribuído de **detecção de deadlock de recurso**
- Algoritmo distribuído de **detecção de deadlock de comunicação**

Deadlock de Recurso

Survey Paper: Singhal, "Deadlock in Distributed Systems", IEEE Computer, Nov. 1989.

Condições para Deadlock

Considere um número de processos que desejam ganhar acesso a *n* recursos compartilhados (e.g. locks). A aquisição do recurso é feita de acordo com certa ordem. Um processo está descansando (idle) quando está esperando por recursos, ele está ativo caso contrário. Condições para ocorrência de deadlock:

1. Mais do que um recurso requer acesso exclusivo.
2. Recursos são apenas liberados voluntariamente por processos depois de serem usados (sem interrupção).
3. Mais de um processo deve estar usando um recurso exclusivamente e esperando acesso para outro recurso que correntemente está sendo usado.
4. Num grafo, tal que um arco do Processo Pi para Processo Pj, indica que Pi está esperando por um recurso em uso por Pj, então um ciclo indica deadlock.

e.g. Veja o capítulo de Bancos de Dados Distribuídos

O deadlock pode ser o resultado de impor esquema de lock de 2 fases para garantir que transações não serializáveis não ocorram.

Abordagens:

1. Prevenção de Deadlock (abordagem pessimista)

- É conseguida ou pela aquisição prévia dos recursos pelo processo antes de começar a executar ou pela interrupção de processo que detém o recurso.
- Ordem total rígida na aquisição de recurso
- Devem ser declaradas as necessidades máximas de recursos e calcular a ordenação total permissível entre as transações
    - requer o pré conhecimento do uso requerido
    - restritivo (não dinâmico)
    - degrada o desempenho pela limitação da concorrência
- Na Prevenção de Deadlock baseada em Rótulos de Tempo, é possível que muitos processos possam ser abortados sem ocorrência de deadlock no caso de estratégias "Wait Die" e "Wound-Wait".

2. "Deadlock Avoidance"

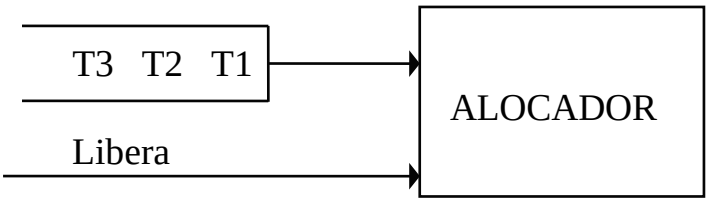
- O recurso é dado se **o estado global resultante é seguro** (o estado global inclui todos os processos e recursos).
- Todo local deve manter um registro do estado global, geralmente, grande capacidade de armazenamento e comunicação intensa são necessárias.
- A tarefa de verificar por estado global seguro deve ser mutuamente exclusiva e geralmente é muito intensa em termos de computação.

3. Detecção de Deadlock e Recuperação (abordagem otimista)

- Deixa que o deadlock ocorra. Os processos ficam bloqueados aguardando os recursos (ciclo de espera persistente é formado).
- Deve-se **detectar** o deadlock distribuído e **resolvê-lo**. Recuperação é feita com a liberação forçada de recursos (aborto de transações) por algum critério de custo.
- Detecção pode ser realizada concorrentemente com as atividades normais de um sistema.
- Deve-se evitar a detecção de pseudo-deadlock (que ocorre devido à falta de estado global atualizado, como em detecção de término).
- É mais utilizada do que as outras duas abordagens.

Exemplo: Algoritmo de Lomet

Abordagem **centralizada**:



O alocador tem o conhecimento do **estado global** da alocação de recurso. Ele pode prevenir ou detectar deadlocks de forma centralizada.

Prevenção de Deadlock

Transações declaram quais recursos elas vão precisar.
Se Tj precisar de um recurso atualmente seguro por Ti, então adicione um arco de Ti para Tj (i.e. Ti **potencialmente bloqueia** Tj).

Ti -> Tj

Uma alocação que resulte num ciclo no grafo indicaria um deadlock **potencial**, então a alocação não deve ser permitida.

e.g. T1: necessidade máxima de x, y
T2: necessidade máxima de y, z
T3: necessidade máxima de z, x

Escalonamento: T1(x,y), T2(y,z), T3(z,x), O(1,x), O(2,y), O(3,z)
O(1,x): T1 -> T3
O(1,x), O(2,y): T2 -> T1 -> T3
O(1,x), O(2,y), O(3,z): T3 -> T2 ->T1 -> T3 ciclo!

Apesar de z estar disponível, O(3,z) está bloqueada uma vez que ela iria criar um ciclo por adicionando um arco de T3 para T2.
Esta abordagem facilmente congestiona as ligações para o alocador se o sistema distribuído for grande.

Abordagem Distribuída:

Replicar o grafo global em todo lugar, e tentando dessa forma garantir uma vista consistente dos pedidos pela rotulação (como no algoritmo de Exclusão Mútua de Lamport).

Contudo, será que todo lugar necessita replicar o grafo inteiro?

Considere uma projeção local do grafo global de acordo com o lugar do recurso:

Escalonamento: T1(x,y), T2(y,z), T3(z,x), O(1,x), O(2,y), O(3,z)

x no lugar Sx: T1 -> T3
y no lugar Sy: T2 -> T1
z no lugar Sz: T3 -> T2 i.e. insuficiente para prevenir deadlock!

Coloque uma condição local mais forte baseada na ordenação total de transações. Vamos adotar a rotulação de tempo.

TSi é o **rótulo de tempo** de transação i.

i.e. Ti é **permitido preceder** Tj sse **TSi < TSj** i.e. **Ti => Tj**

Grafos locais mostram as dependências potenciais e a ordenação de rótulo de tempo total.

Pode ser mostrado que para existir um ciclo no grafo global, deve existir um ciclo em pelo menos um grafo local.

e.g. se T1(x, y), T2(y,z), T3(z, x), O(1,x)
Vamos supor que T1=>T2=>T3 por rótulo de tempo (TS)
(por exemplo, TS de T1 é 30, TS de T2 é 40 e TS de T3 é 50)
então
no lugar Sx: T1 => T3 e T1-> T3 por O(1,x)
no lugar Sy: T1 => T2
no lugar Sz: T2 => T3

Se O(2,y) ocorresse, faria com que T2 potencialmente bloqueasse T1 em Sy e isto criaria um arco T2->T1 e potencialmente um ciclo. Nesta abordagem, não permitimos que O(2,y) ocorra!

E O(3,z)?
Se O(3,z) ocorresse, faria com que T3 potencialmente bloqueasse T2 em Sy e isto criaria um arco T3->T2 e potencialmente um ciclo. Nesta abordagem, não permitimos que O(3,z) ocorra!

Isto é suficiente para prevenir deadlock, mas não é necessário!

Notas:

- Esta abordagem requer uma declaração preliminar de necessidades de recurso de uma transação, i.e., esta abordagem é dessa forma estática.

Prevenção de Deadlock usando o Controle de Concorrência baseado em rótulos de tempo

- O controle de concorrência baseado em rótulos de tempo discutido no Capítulo de Banco de Dados Distribuídos (uso de RTM e WTM) não requer a declaração de necessidades máximas, e é mais dinâmico nos seus pedidos de recurso. Ele não bloqueia transações, mas requer que elas sejam re-inicializadas se um conflito acontecer.
- Dessa forma prevenção de deadlock pode-se basear apenas em rótulos de tempos! (Métodos “Wait-die” e “Wound-wait”)

No lugar x, recurso correntemente alocado a T1 e se T2 pede o recurso x:

Wait-Die (não preemptivo):

```
se  TS1 < TS2  então
        bloqueia T2                 “wait”
senão
        aborta T2                   “die”
```

Wound-Wait:

```
se  TS2  < TS1   então
        aborta T1                   “wound”
senão
        bloqueia T2                 “wait”
```

Detecção de Deadlock de Recurso por Computação Difusa

Chandy, Misra, Haas, “Distributed Deadlock Detection”, ACM TOCS, vol. 1, no 2, May 1983.

Processos P fazem pedidos de recursos através de seus controladores de recurso local.
Ci Controlador do site onde o processo Pi está hospedado.
Se vários processos estiverem hospedados (por exemplo, 3, 4 e 5) no mesmo site, o controlador é único (**C3** = C4 = C5).
Controladores de recurso C comunicam para pedir recursos não-locais e manter um mapa de dependências de pedidos de recurso (i.e. grafos locais com dependências não-locais).
Uma transação é composta por um conjunto de processos rodando em diferentes nós. Pix é o processo rodando no nó i da transação x. Cada transação pode ter apenas um processo no nó i.
Um processo Pix (Processo i da transação x) aguardando por recurso que é controlado por Cj inicia detecção em **Ci** por emitindo um *probe* (i, i, j) (**sonda**) para todos os controladores dependentes j.
Cada controlador Cj propaga o probe (i, j, k) para o controlador Ck quando (transação x depende da transação y e) Pjy (Processo j da transação y) está desocupado e está esperando por processo que segura o recurso em Ck.

Durante a detecção de deadlock, os controladores usam suas informações de dependência de recurso para construir e manter um array Booleano para marcar dependências globais (inicialmente falso) i.e. algoritmo de difusão de marcações.

dependente(k, i) => controlador Ck sabe que *Pi depende de Pk* (Pi espera por recurso que é controlado por Ck).
dependente(k, i) é verdade apenas se Pk (Ck) sabe que Pi é dependente de P_k.
dependente(k, k) => deadlock!

ALGORÍTMO

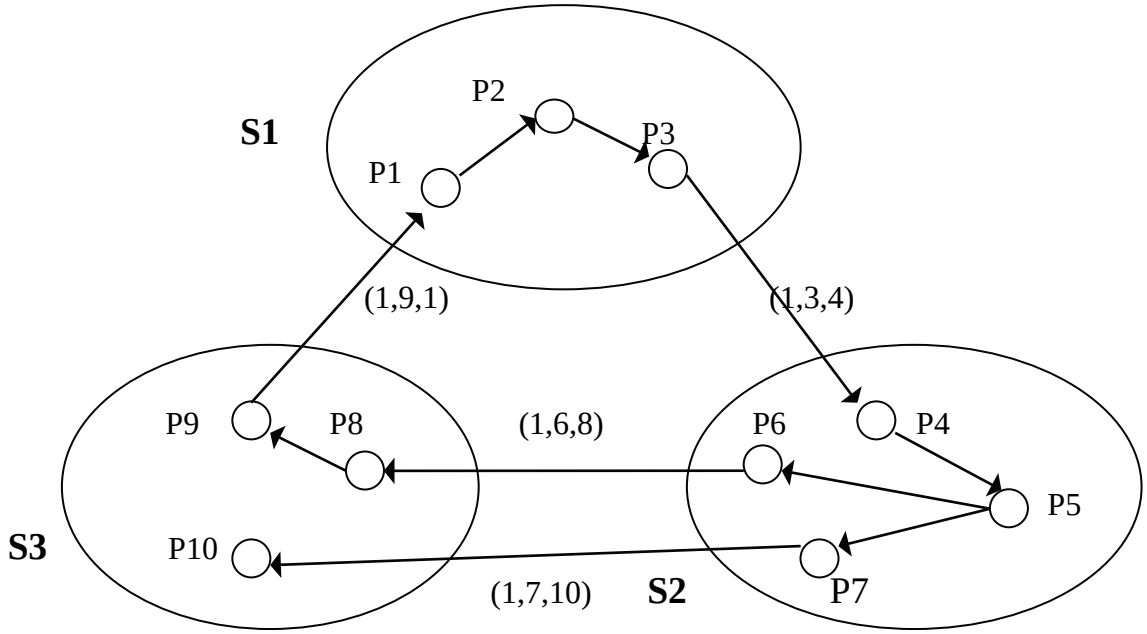
Ck Controlador do site onde o processo Pk está hospedado. O número de controladores é igual ao número de sites.

Iniciador:
Para controlador Ci do processo desocupado Pi:
se Pi está em deadlock local então
declare deadlock
senão
se para todo Pa, Pb tal que
Pi localmente dependente de Pa
e Pa esperando por Pb
e Pa, Pb em diferentes controladores então
envie probe(i, a, b)

Para um controlador Ck recebendo probe(i, j, k):
se Pk desocupado
e dependente(k, i) = falso
e Pk não deu reply positivo para todos os pedidos de Pj então
dependente(k, i) = verdade
se k = i então
declare deadlock
senão
se para todo Pa, Pb tal que
Pk localmente dependente de Pa
e Pa esperando por Pb
e Pa, Pb em diferentes controladores então
envie probe(i, a, b)

Para um controlador quando um processo Pk torna-se ativo (quando recebe o recurso):
para todo i
dependente(k, i) := falso

Exemplo: O processo 1 inicia detecção de deadlock.



P1 envia probe (1,3,4) para o controlador C4 no lugar S2.
Uma vez que P6 está esperando por P8 e P7 está esperando por P10, C4 envia probes (1,6,8) e (1,7,10) para C8.
C8 por sua vez envia probe (1, 9, 1), C1 declara que P1 está em deadlock.

Se P9 não estivesse aguardando recursos seguros por P1, duas ações aconteceriam:

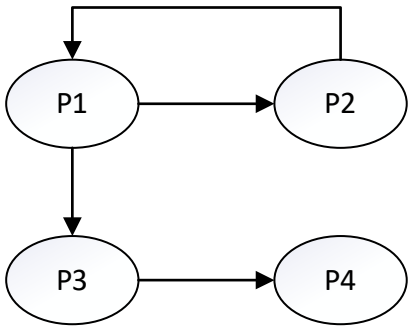
- O probe seria descartado, pois P9 estaria ativo.
- P9 iria liberar os recursos para P8 que por sua vez, liberaria recurso para P6. Nesse caso, a variável dependente (6, i) = falso para todos os Pi que dependem de P6.

Deadlock de Comunicação

Considere um conjunto de processos. Processos podem estar **bloqueados** (esperando por mensagem de outros processos) ou **ativos**.

Num grafo tal que um arco do Processo Pi para Pj indica que Pi está esperando por comunicação de Pj, então um ciclo **não** necessariamente indica um deadlock.

Por exemplo, sejam quatro processos no estado aguardando comunicação tal que P1 aguarda comunicação de P2 ou P3. O grafo, portanto tem arcos de P1 para P2 e de P1 para P3. Vamos supor também que P2 aguarda comunicação de P1 somente e P3 esteja aguardando comunicação de um outro processo ativo, P4. Se o processo ativo P4 enviar uma mensagem para P3 pode ser que este último envie uma mensagem para P1.



Portanto para termos um deadlock de comunicação, é necessário que todos os processos **alcançáveis** de Pi no grafo devem estar bloqueados.

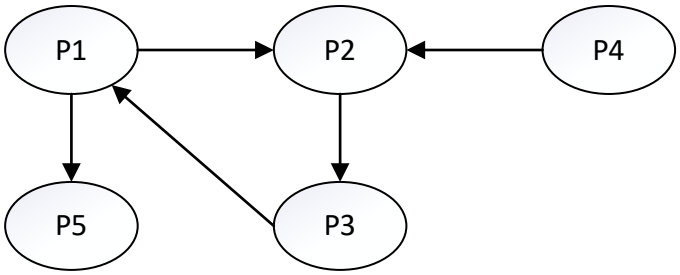
Condições para Deadlock

Conjunto Dependente DSi do processo Pi é o conjunto de todos os processos dos quais Pi está esperando comunicação.

Um conjunto S de processos está em deadlock de comunicação sse:

- Todos Pi em S estão bloqueados.
- Para todos os processos Pi em S: DSi está em S.
- **Nenhuma mensagem está em transito** entre quaisquer processos em S.

Exemplo:
S= {P1, P2, P3, P4, P5}
Todos os processos em S estão bloqueados
DS1 = {P2, P5}
DS2 = {P3}
DS3 = {P1}
DS4 = {P2}
DS5 = {}



Se não existir mensagem em trânsito entre Si e Sj para quaisquer Si e Sj de S então S está em deadlock de comunicação!

Deadlock de Comunicação:

- processos sabem das dependências locais
- esperando por qualquer mensagem para a quebra de deadlock
- *knot* de processos bloqueados (S é um *knot* se todos os sucessores de qualquer processo Pi em S estão em S)

Deadlock de Recurso:

- controladores dos recursos sabem das dependências locais
- esperando por todos os recursos para a quebra de deadlock
- ciclo de processos

Detecção de Deadlock de Comunicação por Computação Difusa

Chandy, Misra, Haas, “Distributed Deadlock Detection”, ACM TOCS, vol. 1, no 2, May 1983.

Suposições:

- Tempo de transmissão é arbitrário, mas finito (sem perda)
- **Preserva a sequência de mensagens (**não há ultrapassagem de mensagem)

Propriedades:

- Usa conexões de comunicação das mensagens de aplicação
- Código simétrico

Princípios

- Qualquer processo que está aguardando comunicação pode iniciar a detecção de deadlock.
- O iniciador Pi é a raiz da computação difusa, com DSi como seus descendentes. Pi inicia mensagens de controle “query”.
- Um processo aguardando comunicação (ou bloqueado) Pj propaga novos “queries” para DSj e dá reply para um query repetido.
- Um processo ativo descarta queries e replies.
- Se Pi receber um reply de confirmação de todos os processos em DSi então Pi está em deadlock.

Mensagens:

- Mensagem query (i, m, j, k): iniciador Pi, m-ésimo query (detecção) de Pj para Pk
- Mensagem reply(i, m, k, j): reply do query (i, m, j, k) de Pk para Pj
- Query do Iniciador: query (i, m, i, j)
- Reply de confirmação: reply (i, m, j, i) => deadlock

Cada processo **Pk** mantém quatro arrays como se segue. Cada array contém entradas relativas aos processos envolvidos (1..n):

latest(i)	maior número de seqüência *m* em qualquer query (i, m, j, k) (inicialmente 0) garante resposta para o último query de i
engager(i)	i≠k, é j se Pj é responsável pelo valor do latest(i) (inicialmente com valor arbitrário) registra o pai na árvore para reply
num(i)	número total de mensagens query(i, m, k, j) enviados por Pk menos o número total de mensagens reply(i, m, j, k) recebidas por Pk. (inicialmente 0) para dar reply para o pai na árvore.
wait(i)	verdade sse Pk descansando (aguardando, bloqueado) desde a última atualização de latest(i) (inicialmente falso) para detectar mudanças de estado.

engager(i) e num(i) registram a informação da árvore de varredura para a computação difusa.

ALGORÍTMO

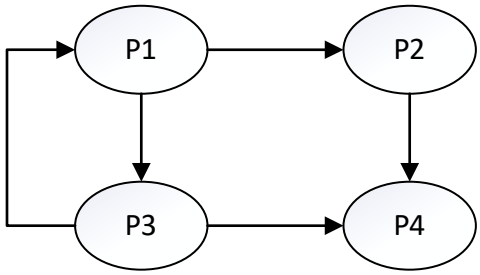
Pi descansando inicia query:
    inc latest(i), wait(i) := verdade, num(i):= |DSi|
    para todo Pj em DSi: envie query(i, latest(i), i, j)

processo ativo Pk:
    para todo i: wait(i):= falso
    descarte todos os queries e replies enquanto ativo.

para processo descansando **Pk**:
    recebe query(i, m, j, k)
    se m > latest(i) então
        latest(i) := m, wait(i):= true, num(i):= |DSk|
        engager(i):= j
        se num(i) != 0 então
            para todo processo em DSk envie query(i, m, k, r)
        senão
            envie reply(i, m, k, engager(i))
    senão
        se wait(i) & m = latest(i) então
            send reply(i, m, k, j)
ou
    recebe reply(i, m, r, k)
    se wait(i) & m=latest(i) então
        dec num(i)
        se num(i) = 0 então
            se i=k então
                declare Pk em deadlock
            senão
                envie reply(i, m, k, engager(i))

Exemplo:

Considere 4 processos com os seguintes estados:
P1: descansando, aguardando por P2 ou P3
P2: descansando, aguardando por P4
P3: descansando, aguardando por P1 ou P4
P4: descansando



Suponha que tempo de comunicação é 1ms.
Cada processo tem os 4 arrays de trabalho devidamente inicializados.

P1	Lat	Eng	Num	Wait
1	0		2	V
2	0		0	V
3	0		0	V
4	0		0	V

P2	Lat	Eng	Num	Wait
1	0		0	V
2	0		0	V
3	0		0	V
4	0		0	V

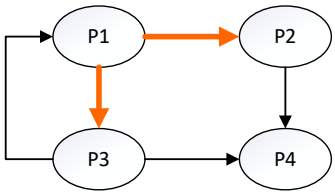
P3	Lat	Eng	Num	Wait
1	0		0	V
2	0		0	V
3	0		0	V
4	0		0	V

P4	Lat	Eng	Num	Wait
1	0		0	V
2	0		0	V
3	0		0	V
4	0		0	V

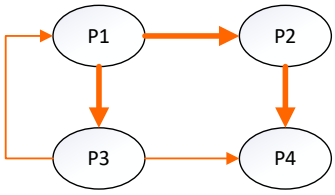
Como apenas P1 inicia a detecção, apenas a linha 1 dos arrays dos processos será atualizada.

Sequência de Execução

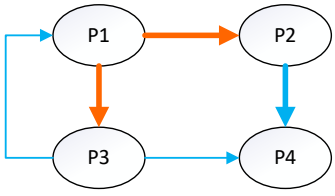
Tempo	Ação	
0	P1 inicia a detecção	
	P1 envia query (1, 1, 1, 2) e (1, 1, 1, 3)	DS1={P2, P3}



1	P2 recebe query (1, 1, 1, 2)	
	P2.engager(1):=1	
	P2.num(1):=1	
	P2 envia query(1, 1, 2, 4)	DS2={P4}
1	P3 recebe query (1, 1, 1, 3)	
	P3.engager(1):=1	
	P3.num(1):=2	
	P3 envia query(1, 1, 3, 1) e query(1, 1, 3, 4)	DS3={P1, P4}



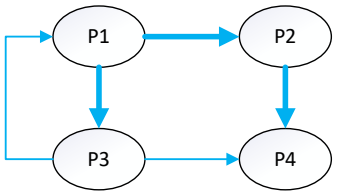
2	P1 recebe query(1, 1, 3, 1)	
	P1 envia reply(1, 1, 1, 3)	
2	P4 recebe query(1, 1, 2, 4)	
	P4.engager(1):=**2**	
	P4 envia reply(1, 1, 4, 2)	DS4 = {}
2	P4 recebe query(1, 1, 3, 4)	
	P4 envia reply(1, 1, 4, 3)	



3	P2 recebe reply(1, 1, 4, 2)	
	P2.num(1):=0	
	P2 envia reply(1, 1, 2, 1)	
3	P3 recebe reply(1, 1, 1, 3)	

P3.num(1):=1

3 P3 recebe reply(1, 1, 4, 3)
P3.num(1):=0
P3 envia reply(1, 1, 3, 1)



4 P1 recebe reply(1, 1, 2, 1)
P1.num(1):=1

4 P1 recebe reply(1, 1, 3, 1)
P1.num(1):=0
P1 declara deadlock de comunicação!

Engager = -1 significa que não tem pai, pois é o processo iniciador

Exercício 1: Se o processo P4 não estivesse descansando, mostre uma possível execução.

Exercício 2: Rode o algoritmo com dois processos iniciando a detecção ao mesmo tempo: P1 e P2.

Comentários Finais

- Correteza de algoritmos: necessidade de apoio de métodos formais.
- Desempenho de algoritmos.
- Resolução de Deadlock: requer o conhecimento de todos os processos envolvidos no deadlock e de todos os recursos obtidos pelos processos. Requer o aborto de um dos processos, liberar os recursos, dar os recursos aos processos em deadlock e excluir as informações de detecção nos lugares. Eventualmente a resolução pode recuperar mais de um deadlock (quando o arco removido pertence a dois ciclos).