Técnicas de Prevenção de Deadlock Em Sistemas Operacionais Modernos

Ricardo Henrique Brunetto[1]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)
Maringá – PR – Brasil

ra94182@uem.br

Resumo. *Este artigo visa a fornecer uma concisa revisão bibliográfica sobre os resumos das técnicas de prevenção de Deadlocks de recursos em Sistemas Operacionais Modernos, apresentando dados gerais de outros trabalhos científicos.*

Contents

1. Introdução

Embora um computador possa ter diversos recursos acoplados e providos, grande parte deles pode ser alocado a apenas um processo por vez (Ex: impressoras, tabelas internas do sistema, unidade de disco, etc). Ao considerar o Sistema Operacional como um Gerente de Recursos, a responsabilidade por garantir o acesso exclusivo de um processo a esses recursos é dele.

O **Deadlock** (ou Impasse) ocorre quando os processos necessitam de mais de um recurso exclusivamente e acabam mutuamente se bloqueando, fazendo com que pemaneçam em execução, porém incapazes de prossegui com seus processamentos. Assim, os Deadlocks ocorrem **independente do Sistema Operacional**. Existem condições específicas para que o Deadlock ocorra e serão abordadas ainda nesta Seção.

O Deadlock não é exclusivo de processos (ou mesmo de software), podendo ocorrer entre máquinas e outras situações em que haja disputa de recursos. Isso implica que um Deadlock pode ocorrer tanto com requisições simultâneas a dispositivos de I/O (Entrada/Saída) quanto em um sistema de banco de dados, por exemplo, em que o acesso a diversos registros acabam bloqueados em dependência mútua. Contudo, o enfoque deste trabalho serão os Deadlocks de recurso.

Deadlocks podem ocorrer de diversas maneiras e uma delas envolve uma classe específica de recursos. Assim, para fins de definição, considerar-se-ão **recursos** como sendo os objetos acessados pelos processos. Dentre os vários recursos disponíveis, alguns possuem várias instâncias idênticas, enquanto outros são únicos. Dessa forma, um recurso pode ser adquirido, utilizado e posteriormente liberado. Dentre os recursos, existem em essência dois tipos:

- **Preemptíveis**: podem ser retirados dos seus proprietários sem prejuízo. (*Ex: Memória e Processador.*)
- **Não Preemptíveis**: podem gerar prejuízos à computação caso seja retirado forçadamente do proprietário. (*Ex: Gravadora de CD e Impressoras.*)

Esta distinção é importante pois, em geral, Deadlocks envolvem recursos não preemptíveis. De forma geral, um processo requisita um recurso, o usa e o libera. Dessa forma, se um recurso não preemptível for requisitado e não estiver disponível, o processo deverá aguardar. Contudo, se o processo detentor do recurso também estiver bloqueado aguardando a liberação de outro, e assim sucessivamente, até que um k-ésimo processo esteja aguardando o primeiro processo, então todos os processos estarão em Deadlock.

Dessa forma, [Tanenbaum 2008] define:

Um conjunto de processos estará em situação de impasse se todo processo pertencente ao conjunto estiver esperando por um evento que somente outro processo desse mesmo conjunto poderá fazer acontecer.

Assim, existem basicamente quatro condições que devem ocorrer simultaneamente para que haja um Deadlock de recurso:

1. **Condição de Exclusão Mútua**: Cada recurso em dado instante estará ou disponível ou associado a um único processo;
2. **Condição de Posse e Espera**: Processos que já detêm recursos podem solicitar outros;

3. **Condição de Não Preempção**: Recursos não podem ser forçadamente tomados de um processo;
4. **Condição de Espera Circular**: A dependência deve ocorrer de modo que um processo i dependa de um recurso que está de posse do processo $i + 1\%n$, considerando $n \geq 2$.

   Existem diversas estratégias para lidar com Deadlocks:

- Ignorar o problema;
- Detectar e recuperar-se;
- Prevenir, neutralizando uma das quatro condições acima;

  Por motivos óbvios, este artigo focará no último item desta lista.

2. Prevenção e Modelagem

Evitar Deadlocks é uma tarefa cara e, muitas vezes, pouco eficiente. Não permitir que as condições 3 e 4 citadas na Seção 1 é a melhor forma de **prevenir** Deadlocks de ocorrerem. Para tanto, deve-se modelar as interações entre os processos e o Sistema Operacional.

De maneira ingênua, poder-se-ia bloquear a condição 3 adquirindo todos os recursos que um proceso necessitasse antes que o mesmo entrasse em execução. Isso não é possível de ser feito. Poder-se-ía, ainda, liberar todos os recursos já adquiridos caso não houvesse algum disponível. Isso poderia acarregar um erro de computação. Assim, é uma forma ineficiente de solucionar o problema.

Semelhantemente para a condição 4, poder-se-ía forçar um grafo de dependência de recursos acíclico (abordado na Seção 2.1) ou implementar uma solução com semelhante à do Jantar dos Filósofos. Isso seria eficiente, porém custoso [UFMG b].

As condições 1 e 2, por outro lado, não podem ser evitadas uma vez que não é possível virtualizar determinados recursos que fornecem serviços a um único processo por vez, e não há como contornar a situação de que os processos utilizarão mais de um recurso.

Isso porque, como pode-se perceber, a **ordem de escalonamento** dos processos pode interferir diretamente na ocorrência ou não de um Deadlock, muito embora o Sistema Operacional não saiba de antemão quais recursos cada processo pretende utilizar. De qualquer forma, o escalonador de processos do Sistema Operacional pode prevenir os Deadlocks.

Impedir que Deadlocks ocorram costuma ser mais difícil que detectá-los e então corrigir o problema. Isso porque, para prevenir um Deadlock, o Sistema Operacional deve estar ciente das condições de cada processo que está rodando e, assim, tomar uma decisão quanto ao escalonamento ou atendimento da requisição de outros processos. Logo, é possível evitar que um Deadlock ocorra quando determinadas informações do processo estão disponíveis antes da execução do mesmo. Além de ser uma tarefa cada vez mais complexa, dadas as disponibilidades de paralelismo atuais, é extremamente cara ao Sistema. Contudo, se aplicada da maneira correta e baseada em uma modelagem eficiente do problema, pode ter resultados positivos e tornar o sistema muito mais confiável.

Essa confiabilidade é importante principalmente em setores que dependem exclusivamente do aparato computacional para desenvolvimento ou manutenção das funções.

Assim, em alguns casos, impedir que o Deadlock ocorra pode trazer um benefício maior que o custo.

2.1. Modelagem de Grafos

É possível modelar as alocações de recursos através de um grafo dirigido, por exemplo, onde os vértices quadrados indicam recursos e os vértices circulares são processos, e as arestas indicidentes em recursos indicam dependências (bloqueio do processo que compartilha a aresta) e as indicidentes em processos indicam uso do recurso que compartilha a aresta. Assim, quando existirem ciclos, estar-se-á formando um Deadlock.

Dessa forma, pode-se impedir a condição 4 listada na Seção 1.

Contudo, o Sistema Opercaional não é obrigado a colocar os processos em execução em nenhuma ordem específica. O atendimento das requisições ocorrem conforme o algoritmo de escalonamento do sistema. O máximo que o Sistema Operacional pode fazer é bloquear um processo que pode gerar um Deadlock, não atendendo sua requisição de alocação de recurso até que a mesma possa ser feita em segurança (não formar mais ciclos).

Contudo, para que o sistema tenha controle desta condição de segurança, é necessário que ele implemente a modelagem por grafos. Dessa forma, o Sistema Operacional pode implementar o grafo como uma Matriz de Incidências [UFRGS] ou como uma Lista de Adjacências [UFCG]. Assim, ao inserir um processo, se o Sistema Operacional souber os recursos que este irá utilizar, pode fazer uma cópia da estrutua acrescida do vértice do processo em questão, para que simule o que ocorreria caso este estivesse em execução. Assim, aplica-se um algoritmo que busque por ciclos em um grafo (Ex: Busca em Profundidade [UFMG a]). Dessa forma, caso haja um ciclo com a novo vértice inserido, então ocorrerá um Deadlock. Assim, o Sistema Operacional não permite que o processo seja escalonado ou não atende às suas requisições de alocação de recursos e o bloqueia. O objetivo é manter o grafo de recursos acíclico.

O problema com esta forma de modelagem é que ambas as implementações consomem memória excessiva quando a Tabela de Processos é grande. Além disso, há uma grande carga de processamento entre clonar o grafo existente, inserir o vértice novo e buscar todo o grafo por ciclos. Por fim, caso haja mais de um recurso desejado pelo processo, a ordem de requisição também deve ser levada em consideração e, como o Sistema Operacional geralmente não possui essa informação, todas as possibilidades devem ser consideradas para que haja 100% de eficiência. Isso tudo seria realizado a cada vez que um processo fosse executar. Dessa forma, aplicar este algoritmo torna-se inviável, embora eficaz.

É importante salientar que a prevenção de Deadlocks utilizando grafo de recursos pode ser generalizada para vários recursos do mesmo tipo.

2.2. Estados Seguros

Antes de aboradar um Estado Seguro, é interessante que sejam apresentadas as estruturas de dados fundamentais na definição dos algoritmos que se baseiam em tais estados. Supondo n processos e m recursos, tem-se os conceitos de:

- **Vetor de Recursos Existentes (E)**: composto por m elementos, onde o valor armazenado na posição i significa a quantidade existente no sistema do recurso i.

- **Vetor de Recursos Disponíveis (A)**: composto por m elementos, onde o valor armazenado na posição i significa a quantidade disponível do recurso i, ou seja, passíveis de serem alocadas.
- **Matriz de Alocação Atual (C)**: de ordem $n \times m$ (Processo $\times$ Recurso), onde o elemento C_{ij} é a quantidade do recurso j que está alocada para o processo i (está em uso).
- **Matriz de Requisição (R)**: de ordem $n \times m$ (Processo $\times$ Recurso), onde o elemento C_{ij} é a quantidade do recurso j que está sendo requisitada pelo processo i (quer usar).

Apresentadas as estruturas, define-se um **Estado Seguro** como sendo um estado onde não há uma situação de impasse (Deadlock) e se há alguma ordem possível em que os processos podem ser alocados para que todos cheguem à conclusão. Contudo, é importante salientar que um estado que não é seguro não significa necessariamente um Deadlock. Isso porque, em um Estado Seguro, o sistema pode garantir que a execução será finalizada para todos, enquanto que no Estado Não-Seguro, a garantia não pode ser dada, porém, ainda pode haver a possibilidade disso acontecer.

Assim, pode-se tomar uma decisão de executar ou não um determinado processo com base na quantidade de recursos que ele consome atualmente em face a quantidade que irá necessitar, contrastando com a mesma conjuntura dos demais processos que também envolvem estes recursos. Dessa forma, o Sistema é capaz de simular o controle de cada recurso (quantidade disponível e em uso, através das estruturas acima) caso determinado processo seja escalonado. Assim, é possível obter um veredito para execução ou não de determinado processo.

As vantagens desse método são a baixa necessidade de processamento relativo e a premissa. Contudo, o consumo de memória também é grande. A maior desvantagem, contudo, é a baixa perspectiva do algoritmo, ou seja, sua característica de tomar uma decisão baseando-se apenas no próximo estado a ser atingido. Obviamente, existem variantes que permitem maior flexibilidade nesse quesito, porém exageram na memória e no processamento, tornando-as, na maioria das vezes, inviáveis.

2.3. Trajetórias de Recursos

Os algoritmos mais importantes para evitar Deadlocks baseiam-se no conceito de Estados Seguros. Conforme previamente explanado, os Estados Seguros são, a grosso modo, instantes de tempo (baseado nas instruções executadas por cada processo) em que os recursos desejados não entram em conflito.

De uma forma algorítmica, o Sistema se resposabiliza por analisar quando cada um dos recursos será solicitado por cada um dos processos. Dessa forma, ele é capaz de construir uma trajetória de recursos de cada um deles e sobrepô-las, ou seja, combiná-las de forma a buscar regiões que não repersentam Estados Seguros, ou seja, regiões em que haverá solicitação simultânea dos recursos.

Nesses casos, o Sistema analisa um processo que deseja ter sua requisição de alocação atendida com outro processo que já está em execução e, e m algum momento, utilizará (ou está utilizando) algum recurso que o novo processo possa estar interessado. Logo, quando as trajetórias de ambos os processos são traçadas, evidenciam-se os instantes (estados) em que ambos os processos estarão compartilhando um determinado

recurso. Como a Regra de Exclusão Mútua (vide Seção 1) deve estar em vigor, não é possível que o processo candidato tenha sua requisição atendida, sendo bloqueado pelo sistema até que o processo atual finalize o uso dos recursos conflitantes.

A representação diagramática deste algoritmo torna-o mais didático. Contudo, computacionamente não se utilizam diagramas. Em termos de implementação, o Sisteam Operacional obrigatóriamente deve ter disponível os recursos que o processo pretende utilizar. Feito isso, pode-se criar uma tabela no sistema que relacione um determinado recurso e os processos que estão utilizando em face ao que pretende fazê-lo. Logo, mapeia-se os recursos para os processos que estão usufruindo dele e os que são candidatos. Como parte-se do princípio que não há Deadlocks formados, a análise executada pelo Sistema torna-se precisa, caso os dados necessários estejam disponíveis.

2.4. Algoritmo do Banqueiro

O algoritmo que realmente pode evitar Deadlocks foi desenvolvido por Dijkstra em 1959 e faz uso de uma técnica de detecção de Deadlocks. A base do algoritmo não difere da lógica fundamental do Estado Seguro apresentado na Seção 2.2, porém, com uma analogia simples: um banqueiro de uma cidade pequena negocia com um grupo singelo de clientes para liberar crédito caso a negociação leve a um estado seguro [Tanenbaum 2008].

Em termos técnicos, o algoritmo aponta se um processo pode executar de maneira segura ou não.

Existem basicamente duas versões para o algoritmo, baseado no modelo de recursos que os processos podem utilizar e que o sistema oferece.

2.4.1. Único Recurso

Quando se tem um único recurso de um mesmo tipo disponível, a situação pode ser resolvida com menos esforço computacional.

Nesse caso, o banqueiro (entende-se Sistema Operacional) disponibilza a seus clientes (entende-se processos) uma linha de crédito com um número máximo de k de unidades de crédito (entende-se recursos) para saque. Contudo, ele possui apenas x (com $x \leq k$) unidades no caixa para retirada, uma vez que ele acredita que nem todos os clientes precisarão retirar seus investimentos imediatamente. Assim, os clientes fazem requisições periódicas de empréstimos (requisições de recursos) e o banqueiro avalia se é viável aceitar a requisição.

O método aplicado para tomar tal decisão, contudo, é o alicerce do algoritmo. Para avaliar a viabilidade, o Sistema faz o seguinte: A execução é permitida se a soma dos recursos requisitados por um processo é menor que os recursos disponíveis no sistema. Logo, se:

- Algum processo P detentor de $r \geq 0$ recursos requisita uma quantidade $a \geq 0$ de recursos e o sistema possui b disponíveis, onde $a \leq b$ e os a recursos podem permitir que P finalize sua execução (liberando todos os $r + a$ recursos para alocação em outros processos e que seja suficiente para que algum outro seja satisfeito), então P tem sua requisição atendida, visto que o sistema se encontra em um **Estado Seguro**.

- Caso contrário, P é bloqueado.

Nota-se que, caso o sistema fosse conduzido a um estado não-seguro, então o banqueiro basicamente poderia não ser capaz de atender nenhum cliente futuro caso eles solicitassem uma quantidade máxima de crédito. Isso é apenas uma possibilidade, pois os clientes podem não solicitar toda quantia que podem, porém essa é uma situação com a qual o banqueiro não pode considerar concreta. Por isso o estado não é seguro. Pode não ser necessariamente um Deadlock, mas pode conduzir a um em sequência.

2.4.2. Múltiplos Recursos

Ainda na analogia fornecida pela Seção 2.4.1, o algotimo pode ser generalizado. Em termos computacionais, o esforço requerido para processamento é consideravelmente maior, embora a essência do algoritmo permaneça: garantir estados seguros.

Para tanto, são necessárias as estruturas de dados apresentadas na Seção 2.2. O método para que o banqueiro (sistema) libere crédito (recurso) para um cliente (processo) é:

1. Encontre uma linha i em R onde $R_{ij} \leq A_{ij}, \forall j \in \{1, \ldots, m\}$ (ou seja, uma linha onde todos os valores de R sejam menores que os de A em suas respectivas posições). Se não ocorre $\exists i \in R$, então o sistema chegará a um Deadlock.
2. Se as quantidades são válidas e fazem o processo terminar, então termina-se o processo e adiciona-se ao vetor A todos os recursos que o mesmo possuía.
3. Repete-se 1 e 2 até que todos os processos estejam terminados.

A ordem de escolha das linhas em 1 não é relevante, ainda que haja mais de uma opção.

Embora o algoritmo desenvolvido por Dijikstra seja funcional e realmente eficiente, na prática não se pode aproveitá-lo como fora projetado. Isso porque raramente o Sistema operacional conhece o máximo de recursos que os processos irão utilizar. Muitas vezes, nem os próprios processo conhecem esta propriedade, uma vez que este número pode ser dinamicamente determinado durante a execução do programa. Em outros casos, recursos podem repentinamente tornar-se indisponíveis, o que pode gerar inconsistência nos valores utilizados para controle.

3. Evitando as Condições de Deadlock

Embora modelar o problema e desenvolver algoritmos para tratá-los e evitar que os Deadlocks ocorram possa parecer uma boa premissa, em geral acaba por não funcionar como se deveria. O maior desafio dos algoritmos abordados na Seção 2 é saber quantos recursos cada processo irá utilizar (ainda que o máximo).

Por outro lado, se ao menos uma das quatro condições para ocorrência de Deadlocks (vide Seção 1) forem eliminadas, então o Deadlock não ocorrerá. Dessa forma, desenvolvem-se técnicas para evitar que cada uma delas ocorra, dentro dos limites do possível.

3.1. Condição de Exclusão Mútua

Caso nunca ocorra de determinado recurso ser alocado a um único processo, então não haverá Deadlock. Fornecer acesso simultâneo a dois processos em um recurso que deve

ser utilizado apenas por um pode prejudicar a computação (resultado final). Contudo, há uma ténica interessante para controle de múltiplas requisições a recursos que suportam apenas uma utilização por vez: **spooling**.

O **Spool** é uma técnica que, a grosso modo, bufferiza as requisições ao recurso e executa em ordem pela qual as mesmas foram realizadas. Em geral, esse controle é realizado por um *daemon* que, em suma, **não depende de nenhum outro recurso**.

Ainda sim podem ocorrer Deadlocks em outros recursos que sejam necessários para realizar a requisição ao recurso controlado pelo *daemon*. Para tanto, é interessante buscar evitar alocar um recurso que não seja absolutamente necessário e assegurar que um seleto grupo de processos possa ter acesso a determinado recurso.

3.2. Condição de Posse e Espera

Se for possível impedir que um determinado processo que já possui um recurso requisite outro, então será possível eliminar os Deadlocks, uma vez que não poderá ocorrer um bloqueio circular mútuo de recursos não-preemptíveis.

Conforme já comentado na Seção 2, uma forma de fazer isso ocorrer seria forçar os processos a solicitarem todos os recursos que pretendem utilizar antes da execução. De posse destes recursos, o sistema seria capaz de relacionar as necessidades mútuas dos processos. Assim, se tudo estiver disponível, o processo terá alocado tudo que necessita e não haverá possibilidade de impasse. Caso haja conflito, nenhum recurso será alocado ao processo e ele será bloqueado (Uma alternativa seria exigir que o processo que esteja requisitando um recurso libere temporariamente todos os recursos que possui para, então, tentar obter todos os que precisa simultaneamente).

Aqui há o mesmo problema de todos os métodos da Seção 2: nem todos os processos têm conhecimento da quantidade ou sequer dos tipos de recursos que vão utilizar antes de iniciar a execução. Além disso, não há otimização no uso dos recursos, podendo ocasionar desperdício, ou seja, um recurso que já está disponível permanece ocioso até que o processo faça uso de todos os outros. Pode ocorrer, ainda de um determinado processo que faz uso de um número grande de recursos permanecer por muito tempo sem ser executado, aguardando que todos os recursos estejam simultaneamente disponíveis.

Apesar dessas outras desvantagens, existem Sistemas Operacionais que manipulam arquivos em lotes que exigem a correlação prévia dos recursos que o processo irá utilizar. Embora desperdice recursos e transfira uma grande carga de trabalho para o programador, **não há risco de ocorrer Deadlock**.

3.3. Condição de Não-Preempção

Alguns recursos podem ser virtualizados para posterior tratamento pelo Sistema Operacional para evitar que um recurso exclusivo seja tomado à força de um processo para que outro faça uso. Um exemplo seria utilizar **spool** (vide Seção 3.1) em situações que necessitem armazenamento.

Outros recursos não podem ser virtualizados (*Ex: Registros em Bancos de Dados*). Nesses casos, tomar o recurso à força e implementar uma técnica de correção pode ser a melhor opção, embora também não seja aplicável a todos os recursos e, inevitavelmente, favoreça a ocorrência de Deadlocks.

3.4. Condição de Espera Circular

Diferentemente das outras condições, a Espera Circular pode ser eliminada de diversas formas. A mais simples, já apresentada na Seção 3.2 exige que cada processo tenha acesso a apenas um recurso de cada vez. Para tomar um segundo recurso, o primeiro deve ser descartado. Isso, obviamente, pode gerar problemas em processos que transfiram dados entre os recursos, por exemplo (*Ex: do disco para impressora*).

Outra forma é enumerar todos os recursos e fazer com que processos possam requisitar quantos recursos desejarem, porém em ordem numérica. Isso garante que todos os processos requisitem recursos em uma mesma ordem e não haja a dependência circular. Dessa forma, um Grafo de Alocação de Recursos (vide Seção 2.1) não apresentará ciclos. A cada vez, um dos recursos alocados será o de numeração maior (caso a enumeração tenha ocorrido em ordem crescente). Ainda que um processo possa "passar na frente de outro" na numeração dos recursos, haverá certeza que o mesmo não dependerá de um recurso de valor menor e corra o risco de bloquear outro processo ou ser bloqueado. Dessa forma, existe um cenário onde todos os processos terminarão e não ocorrerá Deadlock.

Embora a ordenaçaõ numérica elimine os Deadlocks, pode não ser possível encontrar uma ordem que satisfaça a todos. Quando os recursos envolvem Tabela de Processos, Disco e etc, o número de formas de utilização pode ser grande o suficiente para que não haja ordem que satisfaça todos os processos envolvidos. Contudo, interromper a Condição de Espera Circular costuma ser uma das melhores técnicas de Prevenção de Deadlocks.

References

Tanenbaum, A. S. (2008). *Modern Operating Systems*, volume 3. Pearson.

UFCG, U. F. d. C. G. Representação de grafos. `http://www.dsc.ufcg.edu.br/~abrantes/CursosAnteriores/TG051/Representacao.pdf`. Acessado em: 09/01/2018.

UFMG, U. F. d. M. G. Algoritmos em grafos. `http://www2.dcc.ufmg.br/livros/algoritmos-edicao2/cap7/transp/completo4/cap7.pdf`. Acessado em: 06/01/2018.

UFMG, U. F. d. M. G. Deadlocks. `http://homepages.dcc.ufmg.br/~scampos/cursos/so/aulas/aula9.html`. Acessado em: 06/01/2018.

UFRGS, U. F. d. R. G. d. S. Os coeficientes do polinômio característico da matriz de adjacência de grafos. `https://www.lume.ufrgs.br/bitstream/handle/10183/155814/Poster_49770.pdf?sequence=2`. Acessado em: 09/01/2018.