Impasses

Os sistemas computacionais estão cheios de recursos que podem ser usados somente por um processo de cada vez. Exemplos comuns incluem impressoras, unidades de fita para backup de dados da empresa e entradas nas tabelas internas do sistema. Ter dois processos escrevendo simultaneamente para a impressora gera uma saída ininteligível. Ter dois processos usando a mesma entrada da tabela do sistema de arquivos invariavelmente levará a um sistema de arquivos corrompido. Em consequência, todos os sistemas operacionais têm a capacidade de conceder (temporariamente) acesso exclusivo a um processo a determinados recursos.

Para muitas aplicações, um processo precisa de acesso exclusivo a não somente um recurso, mas a vários. Suponha, por exemplo, que dois processos queiram cada um gravar um documento escaneado em um disco Blu-ray. O processo A solicita permissão para usar o scanner e ela lhe é concedida. O processo B é programado diferentemente e solicita o gravador Blu-ray primeiro e ele também lhe é concedido. Agora A pede pelo gravador Blu-ray, mas a solicitação é suspensa até que B o libere. Infelizmente, em vez de liberar o gravador Blu-ray, B pede pelo scanner. A essa altura ambos os processos estão bloqueados e assim permanecerão para sempre. Essa situação é chamada de impasse (deadlock).

Recursos

Uma classe importante de impasses envolve recursos para os quais algum processo teve acesso exclusivo concedido. Esses recursos incluem dispositivos, registros de dados, arquivos e assim por diante. Para tornar a discussão dos impasses mais geral possível, vamos nos referir aos objetos concedidos como recursos. Um recurso pode ser um dispositivo de hardware ou um fragmento de informação. Resumindo, um recurso é qualquer coisa que precisa ser adquirida, usada e liberada com o passar do tempo.

Recursos preemptíveis e não preemptíveis

Um recurso preemptível é aquele que pode ser retirado do processo proprietário sem causar-lhe prejuízo algum. A memória é um exemplo de um recurso preemptível.

Um recurso não preemptível, por sua vez, é um recurso que não pode ser tomado do seu proprietário atual sem potencialmente causar uma falha. Se um processo começou a ser executado em um Blu-ray, tirar o gravador Blu-ray dele de repente e dá-lo a outro processo resultará em um Blu-ray bagunçado. Gravadores Blu-ray não são preemptíveis em um momento arbitrário.

A questão se um recurso é preemptível depende do contexto. Em um PC padrão, a memória é preemptível porque as páginas sempre podem ser enviadas para o disco para depois recuperá-las. No entanto, em um smartphone que não suporta trocas (swapping) ou paginação, impasses não podem ser evitados simplesmente trocando uma porção da memória.

Em geral, impasses envolvem recursos não preemptíveis. Impasses potenciais que envolvem recursos preemptíveis normalmente podem ser solucionados realocando recursos de um processo para outro.

A sequência abstrata de eventos necessários para usar um recurso é dada a seguir.
1. Solicitar o recurso.
2. Usar o recurso.
3. Liberar o recurso.

Se o recurso não está disponível:
– processo pode ser bloqueado
– pode retornar uma falha com um código de erro

Aquisição de recursos

Para alguns tipos de recursos, como registros em um sistema de banco de dados, cabe aos processos do usuário, em vez do sistema, gerenciar eles mesmos o uso de recursos. Uma maneira de permitir isso é associar um semáforo a cada recurso. Esses semáforos são todos inicializados com 1. Também podem ser usadas variáveis do tipo mutex. Os três passos listados são então implementados como um down no semáforo para aquisição e utilização do recurso e, por fim, um up no semáforo para liberação do recurso.

Introdução aos impasses

Um impasse pode ser definido formalmente como a seguir: Um conjunto de processos estará em situação de impasse se cada processo no conjunto estiver esperando por um evento que apenas outro processo no conjunto pode causar.

Como todos os processos estão esperando, nenhum deles jamais causará qualquer evento que possa despertar um dos outros membros do conjunto, e todos os processos continuam a esperar para sempre. Para esse modelo, presumimos que os processos têm um único thread e que nenhuma interrupção é possível para despertar um processo bloqueado.

Na maioria dos casos, o evento que cada processo está esperando é a liberação de algum recurso atualmente possuído por outro membro do conjunto. Em outras palavras, cada membro do conjunto de processos em situação de impasse está esperando por um recurso que é de propriedade do processo em situação de impasse. Nenhum dos processos pode executar, nenhum deles pode liberar quaisquer recursos e nenhum pode ser desperto. O número de processos e o número e tipo de recursos possuídos e solicitados não têm importância. Esse resultado é válido para qualquer tipo de recurso, incluindo hardwares e softwares. Esse tipo de impasse é chamado de impasse de recurso, e é provavelmente o tipo mais comum, mas não o único.

Condições para ocorrência de impasses

Quatro condições têm de ser válidas para haver um impasse:

1. Condição de exclusão mútua. Cada recurso está atualmente associado a exatamente um processo ou está disponível.

2. Condição de posse e espera. Processos atualmente de posse de recursos que foram concedidos antes podem solicitar novos recursos.

3. Condição de não preempção. Recursos concedidos antes não podem ser tomados à força de um processo. Eles precisam ser explicitamente liberados pelo processo que os têm.

4. Condição de espera circular. Deve haver uma lista circular de dois ou mais processos, cada um deles esperando por um processo de posse do membro seguinte da cadeia.

Todas essas quatro condições devem estar presentes para que um impasse de recurso ocorra.

Modelagem de impasses

Um arco direcionado de um nó de recurso (quadrado) para um nó de processo (círculo) significa que o recurso foi previamente solicitado, concedido e está atualmente com aquele processo.

Um arco direcionado de um processo para um recurso significa que o processo está atualmente bloqueado esperando por aquele recurso.

Um ciclo no grafo significa que há um impasse envolvendo os processos e recursos no ciclo.

Quando os processos são executados sequencialmente, não existe a possibilidade de enquanto um processo espera pela E/S, outro poder usar a CPU.

Por ora, basta compreender que grafos de recursos são uma ferramenta que nos deixa ver se uma determinada sequência de solicitação/liberação pode levar a um impasse. Apenas atendemos às solicitações e liberações passo a passo e após cada passo conferimos o grafo para ver se ele contém algum ciclo. Se afirmativo, temos um impasse; se não, não há impasse.

Em geral, quatro estratégias são usadas para lidar com impasses.

1. Simplesmente ignorar o problema. Se você o ignorar, quem sabe ele ignore você.
2. Detecção e recuperação. Deixe-os ocorrer, detecte-os e tome as medidas cabíveis.
3. Evitar dinamicamente pela alocação cuidadosa de recursos.
4. Prevenção, ao negar estruturalmente uma das quatro condições

Algoritmo do avestruz

A abordagem mais simples é o algoritmo do avestruz: enfie a cabeça na areia e finja que não há um problema. Assuma uma probabilidade muito baixa de ocorrência de impasses. Evitar o alto custo em termo de desempenho na eliminação de impasses.

Detecção e recuperação de impasses

Uma segunda técnica é a detecção e recuperação. Quando essa técnica é usada, o sistema não tenta evitar a ocorrência dos impasses. Em vez disso, ele os deixa ocorrer, tenta detectá-los quando acontecem e então toma alguma medida para recuperar-se após o fato.

Detecção de impasses com um recurso de cada tipo

Vamos começar com o caso mais simples: existe apenas um recurso de cada tipo. Um sistema assim poderia ter um scanner, um gravador Blu-ray, uma plotter e uma unidade de fita, mas não mais do que um de cada classe de recurso. Em outras palavras, estamos excluindo sistemas com duas impressoras por ora.

Se esse grafo de recursos contém um ou mais ciclos, há um impasse. Qualquer processo que faça parte de um ciclo está em situação de impasse. Se não existem ciclos, o sistema não está em impasse.

Embora seja relativamente simples escolher os processos em situação de impasse mediante inspeção visual de um grafo simples, para usar em sistemas reais precisamos de um algoritmo formal para detectar impasses.

O que esse algoritmo faz é tomar cada nó, um de cada vez, como a raiz do que ele espera ser uma árvore e então realiza uma busca do tipo busca em profundidade nele. Se acontecer de ele voltar a um nó que já havia encontrado, então o algoritmo encontrou um ciclo. Se ele exaurir todos os arcos de um dado nó, ele retorna ao nó anterior. Se ele retornar até a raiz e não conseguir seguir adiante, o subgrafo alcançável a partir do nó atual não contém ciclo algum. Se essa propriedade for válida para todos os nós, o grafo inteiro está livre de ciclos, então o sistema não está em impasse.

O algoritmo não ótimo mas prova a possibilidade de detecção de impasses.

Detecção de impasses com múltiplos recursos de cada tipo

Seja m o número de classes de recursos, com E1 recursos de classe 1, E2 recursos de classe 2, e geralmente, Ei recursos de classe i ($1 \leq i \leq m$). E é o vetor de recursos existentes.

A qualquer instante, alguns dos recursos são alocados e não se encontram disponíveis. Seja A o vetor de recursos disponíveis, com Ai dando o número de instâncias de recurso i atualmente disponíveis.

Agora precisamos de dois arranjos, C, a matriz de alocação atual e R, a matriz de requisição. A i-ésima linha de C informa quantas instâncias de cada classe de recurso Pi atualmente possui. Desse modo, Cij é o número de instâncias do recurso j que são possuídas pelo processo i. Similarmente, Rij é o número de instâncias do recurso j que Pi quer

Se somarmos todas as instâncias do recurso j que foram alocadas e a isso adicionarmos todas as instâncias que estão disponíveis, o resultado será o número de instâncias existentes daquela classe de recursos.

Diz-se que cada processo, inicialmente, está desmarcado. À medida que o algoritmo progride, os processos serão marcados, indicando que eles são capazes de completar e portanto não estão em uma situação de impasse. Quando o algoritmo termina, sabe-se que quaisquer processos desmarcados estão em situação de impasse. Esse algoritmo presume o pior cenário possível: todos os processos mantêm todos os recursos adquiridos até que terminem.

O processo escolhido é então executado até ser concluído, momento em que ele retorna os recursos a ele alocados para o pool de recursos disponíveis. Ele então é marcado como concluído. Se todos os processos são, em última análise, capazes de serem executados até a sua conclusão, nenhum deles está em situação de impasse.

Quando procurar por impasses? Uma possibilidade é verificar todas as vezes que uma solicitação de recursos for feita. Isso é certo que irá detectá-las o mais cedo possível, mas é uma alternativa potencialmente cara em termos de tempo da CPU. Uma estratégia possível é fazer a verificação a cada k minutos, ou talvez somente quando a utilização da CPU cair abaixo de algum limiar. A razão para considerar a utilização da CPU é que se um número suficiente de processos estiver em situação de impasse, haverá menos processos executáveis, e a CPU muitas vezes estará ociosa.

Recuperação de um impasse

Suponha que nosso algoritmo de detecção de impasses teve sucesso e detectou um impasse. O que fazer então? Alguma maneira é necessária para recuperar o sistema e colocá-lo em funcionamento novamente.

Recuperação mediante preempção

Em alguns casos pode ser viável tomar temporariamente um recurso do seu proprietário atual e dá-lo a outro processo. Em muitos casos, pode ser necessária a intervenção manual, especialmente em sistemas operacionais de processamento em lote executando em computadores de grande porte.

A capacidade de tirar um recurso de um processo, entregá-lo a outro para usá-lo e então devolvê-lo sem que o processo note isso é algo altamente dependente da natureza do recurso. A recuperação dessa maneira é com frequência difícil ou impossível. Escolher o processo a ser suspenso depende em grande parte de quais processos têm recursos que podem ser facilmente devolvidos.

Recuperação mediante retrocesso
Se os projetistas de sistemas e operadores de máquinas souberem que a probabilidade da ocorrência de impasses é grande, eles podem arranjar para que os processos gerem pontos de salvaguarda (checkpoints) periodicamente. Gerar este ponto de salvaguarda de um processo significa que o seu estado é escrito para um arquivo, para que assim ele possa ser reinicializado mais tarde. O ponto de salvaguarda contém não apenas a imagem da memória, mas também o estado dos recursos, em outras palavras, quais recursos estão atualmente alocados para o processo. Para serem mais eficientes, novos pontos de salvaguarda não devem sobrescrever sobre os antigos, mas serem escritos para os arquivos novos, de maneira que à medida que o processo executa, toda uma sequência se acumula.

Quando um impasse é detectado, é fácil ver quais recursos são necessários. Para realizar a recuperação, um processo que tem um recurso necessário é retrocedido até um ponto no tempo anterior ao momento em que ele adquiriu aquele recurso, reiniciando em um de seus pontos de salvaguarda anteriores.

Recuperação mediante a eliminação de processos
A maneira mais bruta de eliminar um impasse, mas também a mais simples, é matar um ou mais processos. Uma possibilidade é matar um processo no ciclo. Com um pouco de sorte, os outros processos serão capazes de continuar. Se isso não ajudar, essa ação pode ser repetida até que o ciclo seja rompido.

Como alternativa, um processo que não está no ciclo pode ser escolhido como vítima a fim liberar os seus recursos. Nessa abordagem, o processo a ser morto é cuidadosamente escolhido porque ele tem em mãos recursos que algum processo no ciclo precisa.

Sempre que possível, é melhor matar um processo que pode ser reexecutado desde o início sem efeitos danosos. Por exemplo, uma compilação sempre pode ser reexecutada, pois tudo o que ela faz é ler um arquivo- -fonte e produzir um arquivo-objeto.

Por outro lado, um processo que atualiza um banco de dados nem sempre pode executar uma segunda vez com segurança. Se ele adicionar 1 a algum campo de uma tabela no banco de dados, executá-lo uma vez, matá-lo e então executá-lo novamente adicionará 2 ao campo, o que é incorreto.

Evitando impasses

Na maioria dos sistemas os recursos são solicitados um de cada vez. O sistema precisa ser capaz de decidir se conceder um recurso é seguro ou não e fazer a alocação somente quando for. Desse modo, surge a questão: existe um algoritmo que possa sempre evitar o impasse fazendo a escolha certa o tempo inteiro? A resposta é um sim qualificado — podemos evitar impasses, mas somente se determinadas informações estiverem disponíveis.

Trajetórias de recursos

Os principais algoritmos para evitar impasses são baseados no conceito de estados seguros.

Todo ponto no diagrama representa um estado de junção dos dois processos. No início, o estado está em p, com nenhum processo tendo executado quaisquer instruções. Se o escalonador escolher executar A primeiro, chegamos ao ponto q, no qual A executou uma série de instruções, mas B não executou nenhuma. No ponto q a trajetória torna-se vertical, indicando que o escalonador escolheu executar B. Com um único processador, todos os caminhos devem ser horizontais ou verticais, jamais diagonais. Além disso, o movimento é sempre para o norte ou leste, jamais para o sul ou oeste

Quando A cruza a linha I1 no caminho de r para s, ele solicita a impressora e esta lhe é concedida. Quando B atinge o pont o t, ele solicita a plotter.

As regiões sombreadas são especialmente interessantes. A região com linhas inclinadas à direita representa ambos os processos, tendo a impressora. A regra da exclusão mútua torna impossível entrar nessa região. Similarmente, a região sombreada no outro sentido representa ambos processos tendo a plotter e é igualmente impossível.

Estados seguros e inseguros

Diz-se de um estado que ele é seguro se existir alguma ordem de escalonamento na qual todos os processos puderem ser executados até sua conclusão mesmo que todos eles subitamente solicitem seu número máximo de recursos imediatamente.

Vale a pena observar que um estado inseguro não é um estado em situação de impasse. A diferença entre um estado seguro e um inseguro é que a partir de um seguro o sistema pode garantir que todos os processos terminarão; a partir de um estado inseguro, nenhuma garantia nesse sentido pode ser dada.

O algoritmo do banqueiro para um único recurso

Ele é modelado da maneira pela qual um banqueiro de uma cidade pequena poderia lidar com um grupo de clientes para os quais ele concedeu linhas de crédito. O que o algoritmo faz é conferir para ver se conceder a solicitação leva a um estado inseguro. Se afirmativo, a solicitação é negada. Se conceder a solicitação conduz a um estado seguro, ela é levada adiante.

Nessa analogia, os clientes são processos, as unidades são, digamos, unidades de fita, e o banqueiro é o sistema operacional.

Um estado inseguro não precisa levar a um impasse, tendo em vista que um cliente talvez não precise de toda a linha de crédito disponível, mas o banqueiro não pode contar com esse comportamento.

Se afirmativo, presume-se que os empréstimos a esse cliente serão ressarcidos, e o cliente agora mais próximo do limite é conferido, e assim por diante. Se todos os empréstimos puderem ser ressarcidos por fim, o estado é seguro e a solicitação inicial pode ser concedida.

O algoritmo do banqueiro com múltiplos recursos

O algoritmo do banqueiro pode ser generalizado para lidar com múltiplos recursos.

Duas matrizes. A primeira, à esquerda, mostra quanto de cada recurso está atualmente alocado para cada um dos cinco processos. A matriz à direita mostra de quantos recursos cada processo ainda precisa a fim de terminar. Essas matrizes são simplesmente C e R.

Os três vetores mostram os recursos existentes, E, os recursos possuídos, P, e os recursos disponíveis, A, respectivamente.

O algoritmo para verificar se um estado é seguro pode ser descrito agora.

1. Procure por uma linha, R, cujas necessidades de recursos não atendidas sejam todas menores ou iguais a A. Se essa linha não existir, o sistema irá em algum momento chegar a um impasse, dado que nenhum processo pode executar até o fim (presumindo que os processos mantenham todos os recursos até sua saída).

2. Presuma que o processo da linha escolhida solicita todos os recursos que ele precisa (o que é garantido que seja possível) e termina. Marque esse processo como concluído e adicione todos os seus recursos ao vetor A.

3. Repita os passos 1 e 2 até que todos os processos estejam marcados como terminados (caso em que o estado inicial era seguro) ou nenhum processo cujas necessidades de recursos possam ser atendidas seja deixado (caso em que o sistema não era seguro).

Se vários processos são elegíveis para serem escolhidos no passo 1, não importa qual seja escolhido: o pool de recursos disponíveis ou fica maior, ou na pior das hipóteses, fica o mesmo.

Embora na teoria o algoritmo seja maravilhoso, na prática ele é essencialmente inútil, pois é raro que os processos saibam por antecipação quais serão suas necessidades máximas de recursos. Além disso, o número de processos não é fixo, mas dinamicamente variável, à medida que novos usuários se conectam e desconectam. Ademais, recursos que se acreditava estarem disponíveis podem subitamente desaparecer.

Alguns sistemas implementam heurísticas similares ao algoritmo do banqueiro.

Prevenção de impasses

Tendo visto que evitar impasses é algo essencialmente impossível, pois isso exige informações a respeito de solicitações futuras, que não são conhecidas, como os sistemas

reais os evitam? A resposta é voltar para as quatro condições colocadas por Coffman et al. (1971) para ver se elas podem fornecer uma pista. Se pudermos assegurar que pelo menos uma dessas condições jamais seja satisfeita, então os impasses serão estruturalmente impossíveis.

Atacando a condição de exclusão mútua

Se nunca acontecer de um recurso ser alocado exclusivamente para um único processo, jamais teremos impasses. Para dados, o método mais simples é tornar os dados somente para leitura, de maneira que os processos podem usá-los simultaneamente. No entanto, está igualmente claro que permitir que dois processos escrevam na impressora ao mesmo tempo levará ao caos. Utilizar a técnica de spooling na impressora, vários processos podem gerar saídas ao mesmo tempo. Nesse modelo, o único processo que realmente solicita a impressora física é o daemon de impressão. Dado que o daemon jamais solicita quaisquer outros recursos, podemos eliminar o impasse para a impressora.

O que aconteceria se dois processos ainda não tivessem completado suas saídas, embora tivessem preenchido metade do espaço disponível de spool com suas saídas? Nesse caso, teríamos dois processos que haveriam terminado parte de sua saída, mas não toda, e não poderiam continuar. Nenhum processo será concluído, então teríamos um impasse no disco.

Logo, evitar alocar um recurso a não ser que seja absolutamente necessário, e tentar certificar-se de que o menor número possível de processos possa, realmente, requisitar o recurso.

Atacando a condição de posse e espera

Se pudermos evitar que processos que já possuem recursos esperem por mais recursos, poderemos eliminar os impasses. Uma maneira de atingir essa meta é exigir que todos os processos solicitem todos os seus recursos antes de iniciar a execução. Se tudo estiver disponível, o processo terá alocado para si o que ele precisar e pode então executar até o fim. Se um ou mais recursos estiverem ocupados, nada será alocado e o processo simplesmente esperará.

Um problema imediato com essa abordagem é que muitos processos não sabem de quantos recursos eles precisarão até começarem a executar. Na realidade, se soubessem, o algoritmo do banqueiro poderia ser usado. Outro problema é que os recursos não serão usados de maneira otimizada com essa abordagem.

Uma maneira ligeiramente diferente de romper com a condição de posse e espera é exigir que um processo que solicita um recurso primeiro libere temporariamente todos os recursos atualmente em suas mãos. Então ele tenta, de uma só vez, conseguir todos os recursos de que precisa.

Atacando a condição de não preempção

Atacar a terceira condição (não preempção) também é uma possibilidade. Se a impressora foi alocada a um processo e ele está no meio da impressão de sua saída, tomar à força a impressora porque uma plotter de que esse processo também necessita não está disponível é algo complicado na melhor das hipóteses e impossível no pior cenário. No entanto, alguns recursos podem ser virtualizados para essa situação. Promover o spooling da saída da impressora para o disco e permitir que apenas o daemon da impressora acesse a impressora real elimina impasses envolvendo a impressora, embora crie o potencial para um impasse sobre o espaço em disco. Com discos grandes, no entanto, ficar sem espaço em disco é algo improvável de acontecer.

Atacando a condição da espera circular

A espera circular pode ser eliminada de várias maneiras. Uma delas é simplesmente ter uma regra dizendo que um processo tem o direito a apenas um único recurso de cada vez. Se ele precisar de um segundo recurso, precisa liberar o primeiro. Para um processo que precisa copiar um arquivo enorme de uma fita para uma impressora, essa restrição é inaceitável.

Outra maneira de se evitar uma espera circular é fornecer uma numeração global de todos os recursos. Agora a regra é esta: processos podem solicitar recursos sempre que eles quiserem, mas todas as solicitações precisam ser feitas em ordem numérica. Um processo pode solicitar primeiro uma impressora e então uma unidade de fita, mas ele não pode solicitar primeiro uma plotter e então uma impressora. Com essa regra, o grafo de alocação de recursos jamais pode ter ciclos.

Com mais do que dois processos a mesma lógica se mantém. A cada instante, um dos recursos alocados será o mais alto. O processo possuindo aquele recurso jamais pedirá por um recurso já alocado. Ele finalizará, ou, na pior das hipóteses, requisitará até mesmo recursos de ordens maiores, todos os quais disponíveis. Em consequência, ele finalizará e libertará seus recursos. Nesse ponto, algum outro processo possuirá o recurso de mais alta ordem e também poderá finalizar. Resumindo, existe um cenário no qual todos os processos finalizam, de maneira que não há impasse algum.

Embora ordenar numericamente os recursos elimine o problema dos impasses, pode ser impossível encontrar uma ordem que satisfaça a todos. Quando os recursos incluem entradas da tabela de processos, espaço em disco para spooling, registros travados de bancos de dados e outros recursos abstratos, o número de recursos potenciais e usos diferentes pode ser tão grande que nenhuma ordem teria chance de funcionar.

Condição	**Abordagem contra impasses**
Exclusão mútua	Usar spool em tudo
Posse e espera	Requisitar todos os recursos necessários no início
Não preempção	Retomar os recursos alocados
Espera circular	Ordenar numericamente os recursos

Outras questões

Travamento em duas fases, impasses que não envolvem recursos e inanições.

Travamento em duas fases

A abordagem muitas vezes usada é chamada de travamento em duas fases (two-phase locking). Na primeira fase, o processo tenta travar todos os registros de que precisa, um de cada vez. Se for bem-sucedido, ele começa a segunda fase, desempenhando as atualizações e liberando as travas. Nenhum trabalho de verdade é feito na primeira fase

Se, durante a primeira fase, algum registro for necessário que já esteja travado, o processo simplesmente libera todas as travas e começa a primeira fase desde o início. De certa maneira, essa abordagem é similar a solicitar todos os recursos necessários antecipadamente, ou pelo menos antes que qualquer ato irreversível seja feito. Em algumas versões do travamento em duas fases não há liberação e reinício se um registro travado for encontrado durante a primeira fase. Nessas versões, pode ocorrer um impasse. Ex: registros em bancos de dados.

No entanto, essa estratégia em geral não é aplicável. Em sistemas de tempo real e sistemas de controle de processos, por exemplo, não é aceitável apenas terminar um processo no meio do caminho porque um recurso não está disponível e começar tudo de novo. Tampouco é aceitável reiniciar se o processo já leu ou escreveu mensagens para a rede, arquivos atualizados ou qualquer coisa que não possa ser repetida seguramente. O algoritmo funciona apenas naquelas situações em que o programador arranjou as coisas muito cuidadosamente de modo que o programa pode ser parado em qualquer ponto durante a primeira fase e reiniciado. Muitas aplicações não podem ser estruturadas dessa maneira.

Impasses de comunicação

O impasse de recursos é um problema de sincronização de competição. Processos independentes completariam seus serviços se a sua execução não sofresse a competição de outros processos. Um processo trava recursos a fim de evitar estados de recursos inconsistentes causados pelo acesso intercalado a recursos. O acesso intervalado a recursos bloqueados, no entanto, proporciona o impasse de recursos.

Outro tipo de impasse pode ocorrer em sistemas de comunicação (por exemplo, redes), em que dois ou mais processos comunicam-se enviando mensagens.Um arranjo comum é o processo A enviar uma mensagem de solicitação ao processo B, e então bloquear até B enviar de volta uma mensagem de resposta. Suponha que a mensagem de solicitação se perca. A está bloqueado esperando pela resposta. B está bloqueado esperando por uma solicitação pedindo a ele para fazer algo. Temos um impasse.

A não possui nenhum recurso que B quer, e vice-versa. Essa situação é chamada de impasse de comunicação para contrastá-la com o impasse de recursos mais comum. O impasse de comunicação é uma anomalia de sincronização de cooperação. Os processos

nesse tipo de impasse não poderiam completar o serviço se executados independentemente.

Felizmente, existe outra técnica que pode ser empregada para acabar com os impasses de comunicação: controles de limite de tempo (timeouts). Na maioria dos sistemas de comunicação, sempre que uma mensagem é enviada para a qual uma resposta é esperada, um temporizador é inicializado. Se o limite de tempo for ultrapassado antes de a resposta chegar, o emissor da mensagem presume que ela foi perdida e a envia de novo.

Livelock

Em algumas situações, um processo tenta ser educado abrindo mão dos bloqueios que ele já adquiriu sempre que nota que não pode obter o bloqueio seguinte de que precisa. Então ele espera um milissegundo, digamos, e tenta de novo. Em princípio, isso é bom e deve ajudar a detectar e a evitar impasses. No entanto, se o outro processo faz a mesma coisa exatamente no mesmo momento, eles estarão na situação de duas pessoas tentando passar uma pela outra quando ambas educadamente dão um passo para o lado e, no entanto, nenhum progresso é possível, pois elas seguem dando um passo ao lado na mesma direção ao mesmo tempo.

É claro, nenhum processo é bloqueado e poderíamos até dizer que as coisas estão acontecendo, então isso não é um impasse. Ainda assim, nenhum progresso é possível, então temos algo equivalente: um livelock.

Inanição

Um problema relacionado de muito perto com o impasse e o livelock é a inanição (starvation). Em um sistema dinâmico, solicitações para recursos acontecem o tempo todo. Alguma política é necessária para tomar uma decisão sobre quem recebe qual recurso e quando. Essa política, embora aparentemente razoável, pode levar a alguns processos nunca serem servidos, embora não estejam em situação de impasse.

Ex: Impressora dando preferência a arquivos menores.

A inanição pode ser evitada com uma política de alocação de recursos primeiro a chegar, primeiro a ser servido. Com essa abordagem, o processo que estiver esperando há mais tempo é servido em seguida. No devido momento, qualquer dado processo será consequentemente o mais antigo e, desse modo, receberá o recurso de que necessita.