Processos

O conceito mais central em qualquer sistema operacional é o processo: uma abstração de um programa em execução. Um processo é apenas uma instância de um programa em execução, incluindo os valores atuais do contador do programa, registradores e variáveis. Processos podem ser criados e terminados dinamicamente. Cada processo tem seu próprio espaço de endereçamento.

Todos os computadores modernos frequentemente realizam várias tarefas ao mesmo tempo. As pessoas acostumadas a trabalhar com computadores talvez não estejam totalmente cientes desse fato, então alguns exemplos podem esclarecer este ponto. Primeiro, considere um servidor da web, em que solicitações de páginas da web chegam de toda parte. Quando uma solicitação chega, o servidor confere para ver se a página requisitada está em cache. Se estiver, ela é enviada de volta; se não, uma solicitação de acesso ao disco é iniciada para buscá-la. No entanto, do ponto de vista da CPU, as solicitações de acesso ao disco levam uma eternidade. Enquanto espera que uma solicitação de acesso ao disco seja concluída, muitas outras solicitações podem chegar. Se há múltiplos discos presentes, algumas ou todas as solicitações mais recentes podem ser enviadas para os outros discos muito antes de a primeira solicitação ter sido concluída. Está claro que algum método é necessário para modelar e controlar essa concorrência. Processos (e especialmente threads) podem ajudar nisso.

Agora considere um PC de usuário. Quando o sistema é inicializado, muitos processos são secretamente iniciados, quase sempre desconhecidos para o usuário. Por exemplo, um processo pode ser inicializado para esperar pela chegada de e-mails. Outro pode ser executado em prol do programa antivírus para conferir periodicamente se há novas definições de vírus disponíveis. Além disso, processos explícitos de usuários podem ser executados, imprimindo arquivos e salvando as fotos do usuário em um pen-drive, tudo isso enquanto o usuário está navegando na Web. Toda essa atividade tem de ser gerenciada, e um sistema de multiprogramação que dê suporte a múltiplos processos é muito útil nesse caso.

Em qualquer sistema de multiprogramação, a CPU muda de um processo para outro rapidamente, executando cada um por dezenas ou centenas de milissegundos. Enquanto, estritamente falando, em qualquer dado instante a CPU está executando apenas um processo, no curso de 1s ela pode trabalhar em vários deles, dando a ilusão do paralelismo. Às vezes, as pessoas falam em pseudoparalelismo neste contexto, para diferenciar do verdadeiro paralelismo de hardware dos sistemas multiprocessadores (que têm duas ou mais CPUs compartilhando a mesma memória física). Ter controle sobre múltiplas atividades em paralelo é algo difícil para as pessoas realizarem.

O modelo de processo

Conceitualmente, cada processo tem sua própria CPU virtual. Na verdade, a CPU real troca a todo momento de processo em processo, mas, para compreender o sistema, é muito mais

fácil pensar a respeito de uma coleção de processos sendo executados em (pseudo) paralelo do que tentar acompanhar como a CPU troca de um programa para o outro. Esse mecanismo de trocas rápidas é chamado de multiprogramação.

Com o chaveamento rápido da CPU entre os processos, a taxa pela qual um processo realiza a sua computação não será uniforme e provavelmente nem reproduzível se os mesmos processos forem executados outra vez. Desse modo, processos não devem ser programados com suposições predefinidas sobre a temporização.

A diferença entre um processo e um programa é sutil, mas absolutamente crucial. Uma analogia poderá ajudá-lo aqui: considere um cientista de computação que gosta de cozinhar e está preparando um bolo de aniversário para sua filha mais nova. Ele tem uma receita de um bolo de aniversário e uma cozinha bem estocada com todas as provisões: farinha, ovos, açúcar, extrato de baunilha etc. Nessa analogia, a receita é o programa, isto é, o algoritmo expresso em uma notação adequada, o cientista de computação é o processador (CPU) e os ingredientes do bolo são os dados de entrada. O processo é a atividade consistindo na leitura da receita, busca de ingredientes e preparo do bolo por nosso cientista.

Agora imagine que o filho do cientista de computação aparece correndo chorando, dizendo que foi picado por uma abelha. O cientista de computação registra onde ele estava na receita (o estado do processo atual é salvo), pega um livro de primeiros socorros e começa a seguir as orientações. Aqui vemos o processador sendo trocado de um processo (preparo do bolo) para um processo mais prioritário (prestar cuidado médico), cada um tendo um programa diferente (receita versus livro de primeiros socorros). Quando a picada de abelha tiver sido cuidada, o cientista de computação volta para o seu bolo, continuando do ponto onde ele havia parado.

Criação de processos

Quatro eventos principais fazem com que os processos sejam criados:

1. Inicialização do sistema.
2. Execução de uma chamada de sistema de criação de processo por um processo em execução.
3. Solicitação de um usuário para criar um novo processo.
4. Início de uma tarefa em lote.

Término de processos

1. Saída normal (voluntária).
2. Erro fatal (involuntário).
3. Saída por erro (voluntária). (Ex: executar uma instrução ilegal)
4. Morto por outro processo (involuntário). (Ex: processo executa uma chamada de sistema dizendo ao sistema operacional para matar outro processo)

Hierarquias de processos

Em alguns sistemas, quando um processo cria outro, o processo pai e o processo filho continuam a ser associados de certas maneiras. O processo filho pode em si criar mais processos, formando uma hierarquia de processos.

EXEMPLO: Quando um usuário envia um sinal do teclado, o sinal é entregue a todos os membros do grupo de processos associados com o teclado no momento (em geral todos os processos ativos que foram criados na janela atual). Individualmente, cada processo pode pegar o sinal, ignorá-lo, ou assumir a ação predefinida, que é ser morto pelo sinal.

Estados de processos

Embora cada processo seja uma entidade independente, com seu próprio contador de programa e estado interno, processos muitas vezes precisam interagir entre si. Um processo pode gerar alguma saída que outro processo usa como entrada.

Os três estados nos quais um processo pode se encontrar:
1. Em execução (realmente usando a CPU naquele instante).
2. Pronto (executável, temporariamente parado para deixar outro processo ser executado).
3. Bloqueado (incapaz de ser executado até que algum evento externo aconteça).

Em execução -> Bloqueado(1) ou pronto(2)
Pronto -> Em execução(3)
Bloqueado -> Pronto(4)

Ex: cat chapter1 chapter2 chapter3 | grep tree

O primeiro processo, executando cat, gera como saída a concatenação dos três arquivos. O segundo processo, executando grep, seleciona todas as linhas contendo a palavra “tree”. Dependendo das velocidades relativas dos dois processos (que dependem tanto da complexidade relativa dos programas, quanto do tempo de CPU que cada um teve), pode acontecer que grep esteja pronto para ser executado, mas não haja entrada esperando por ele. Ele deve então ser bloqueado até que alguma entrada esteja disponível.

As transições 2 e 3 são causadas pelo escalonador de processos, uma parte do sistema operacional, sem o processo nem saber a respeito delas. A transição 2 ocorre quando o escalonador decide que o processo em andamento foi executado por tempo suficiente, e é o momento de deixar outro processo ter algum tempo de CPU. A transição 3 ocorre quando todos os outros processos tiveram sua parcela justa e está na hora de o primeiro processo chegar à CPU para ser executado novamente. O escalonamento, isto é, decidir qual processo deve ser executado, quando e por quanto tempo, é um assunto importante.

O nível mais baixo de um sistema operacional estruturado em processos controla interrupções e escalonamento. Acima desse nível estão processos sequenciais.

O nível mais baixo do sistema operacional é o escalonador, com uma variedade de processos acima dele. Todo o tratamento de interrupções e detalhes sobre o início e parada de processos estão ocultos naquilo que é chamado aqui de escalonador, que, na verdade, não tem muito código. O resto do sistema operacional é bem estruturado na forma de processos. No entanto, poucos sistemas reais são tão bem estruturados como esse.

Implementação de processos

Para implementar o modelo de processos, o sistema operacional mantém uma tabela (um arranjo de estruturas) chamada de tabela de processos, com uma entrada para cada um deles. Um processo pode ser interrompido milhares de vezes durante sua execução, mas a ideia fundamental é que, após cada interrupção, o processo retorne precisamente para o mesmo estado em que se encontrava antes de ser interrompido.

Tratamento de interrupção

Associada com cada classe de E/S há um local (geralmente em um local fixo próximo da parte inferior da memória) chamado de vetor de interrupção. Ele contém o endereço da rotina de serviço de interrupção. Suponha que o processo do usuário 3 esteja sendo executado quando ocorre uma interrupção de disco. O contador de programa do processo do usuário 3, palavra de estado de programa, e, às vezes, um ou mais registradores são colocados na pilha (atual) pelo hardware de interrupção. O computador, então, desvia a execução para o endereço especificado no vetor de interrupção. Isso é tudo o que o hardware faz. Daqui em diante, é papel do software, em particular, realizar a rotina do serviço de interrupção.

1. O hardware empilha o contador de programa etc.

2. O hardware carrega o novo contador de programa a partir do arranjo de interrupções.

3. O vetor de interrupções em linguagem de montagem salva os registradores.

4. O procedimento em linguagem de montagem configura uma nova pilha.

5. O serviço de interrupção em C executa (em geral lê e armazena temporariamente a entrada).

6. O escalonador decide qual processo é o próximo a executar.

7. O procedimento em C retorna para o código em linguagem de montagem.

8. O procedimento em linguagem de montagem inicia o novo processo atual.

Modelando a multiprogramação

Quando a multiprogramação é usada, a utilização da CPU pode ser aperfeiçoada. Colocando a questão de maneira direta, se o processo médio realiza computações apenas 20% do tempo em que está na memória, então com cinco processos ao mesmo tempo na memória, a CPU deve estar ocupada o tempo inteiro.

Threads

Threads são sequências de instruções dentro de um processo que podem ser executadas de forma concorrente entre si.

Para algumas aplicações é útil ter múltiplos threads de controle dentro de um único processo. Esses threads são escalonados independentemente e cada um tem sua própria pilha, mas todos os threads em um processo compartilham de um espaço de endereçamento comum. Threads podem ser implementados no espaço do usuário ou no núcleo.

A principal razão para se ter threads é que em muitas aplicações múltiplas atividades estão ocorrendo simultaneamente e algumas delas podem bloquear de tempos em tempos. Ao decompormos uma aplicação dessas em múltiplos threads sequenciais que são executados em quase paralelo, o modelo de programação torna-se mais simples.

Apenas agora com os threads acrescentamos um novo elemento: a capacidade para as entidades em paralelo compartilharem um espaço de endereçamento e todos os seus dados entre si. Essa capacidade é essencial para determinadas aplicações, razão pela qual ter múltiplos processos (com seus espaços de endereçamento em separado) não funcionará.

Um segundo argumento para a existência dos threads é que como eles são mais leves do que os processos, eles são mais fáceis (isto é, mais rápidos) para criar e destruir do que os processos.

Um processador de texto com três threads

Threads podem ajudar aqui. Suponha que o processador de texto seja escrito como um programa com dois threads. Um thread interage com o usuário e o outro lida com a reformatação em segundo plano. Tão logo a frase é apagada da página 1, o thread interativo diz ao de reformatação para reformatar o livro inteiro. Enquanto isso, o thread interativo continua a ouvir o teclado e o mouse e responde a comandos simples como rolar a página 1 enquanto o outro thread está trabalhando com afinco no segundo plano. Com um pouco de sorte, a reformatação será concluída antes que o usuário peça para ver a página 600, então ela pode ser exibida instantaneamente.

Com três threads, o modelo de programação é muito mais simples: o primeiro thread apenas interage com o usuário, o segundo reformata o documento quando solicitado, o terceiro escreve os conteúdos da RAM para o disco periodicamente.

Deve ficar claro que ter três processos em separado não funcionaria aqui, pois todos os três threads precisam operar no documento. Ao existirem três threads em vez de três processos, eles compartilham de uma memória comum e desse modo têm acesso ao documento que está sendo editado. Com três processos isso seria impossível.

Um servidor web multithread

Uma maneira de organizar o servidor da web: um thread, o despachante, lê as requisições de trabalho que chegam da rede. Após examinar a solicitação, ele escolhe um thread operário ocioso (isto é, bloqueado) e passa para ele a solicitação, possivelmente escrevendo um ponteiro para a mensagem em uma palavra especial associada com cada thread. O despachante então acorda o operário adormecido, movendo-o do estado bloqueado para o estado pronto.

Quando o operário desperta, ele verifica se a solicitação pode ser satisfeita a partir do cache da página da web, ao qual todos os threads têm acesso. Se não puder, ele começa uma operação read para conseguir a página do disco e é bloqueado até a operação de disco ser concluída. Quando o thread é bloqueado na operação de disco, outro thread é escolhido para ser executado, talvez o despachante, a fim de adquirir mais trabalho, ou possivelmente outro operário esteja pronto para ser executado agora.

O modelo de thread clássico

O que os threads acrescentam para o modelo de processo é permitir que ocorram múltiplas execuções no mesmo ambiente, com um alto grau de independência uma da outra. Ter múltiplos threads executando em paralelo em um processo equivale a ter múltiplos processos executando em paralelo em um computador. No primeiro caso, os threads compartilham um espaço de endereçamento e outros recursos. No segundo caso, os processos compartilham memórias físicas, discos, impressoras e outros recursos. Como threads têm algumas das propriedades dos processos, às vezes eles são chamados de processos leves. O termo multithread também é usado para descrever a situação de permitir múltiplos threads no mesmo processo

(A) Temos três processos tradicionais. Cada processo tem seu próprio espaço de endereçamento e um único thread de controle. Cada um deles opera em um espaço de endereçamento diferente.

(B) Temos um único processo com três threads de controle. Embora em ambos os casos tenhamos três threads. Todos os três compartilham o mesmo espaço de endereçamento.

Threads diferentes em um processo não são tão independentes quanto processos diferentes. Todos os threads têm exatamente o mesmo espaço de endereçamento, o que significa que eles também compartilham as mesmas variáveis globais. Tendo em vista que todo thread pode acessar todo espaço de endereçamento de memória dentro do espaço de endereçamento do processo, um thread pode ler, escrever, ou mesmo apagar a pilha de outro thread. Não há proteção entre threads, porque (1) é impossível e (2) não seria necessário. Ao contrário de processos distintos, que podem ser de usuários diferentes e que podem ser hostis uns com os outros, um processo é sempre propriedade de um único usuário, que presumivelmente criou múltiplos threads de maneira que eles possam cooperar, não lutar. Além de compartilhar um espaço de endereçamento, todos os threads podem compartilhar o mesmo conjunto de arquivos abertos, processos filhos, alarmes e sinais, e assim por diante.

O que estamos tentando alcançar com o conceito de thread é a capacidade para múltiplos threads de execução de compartilhar um conjunto de recursos de maneira que possam trabalhar juntos intimamente para desempenhar alguma tarefa.

Implementando threads no espaço do usuário

Há dois lugares principais para implementar threads: no espaço do usuário e no núcleo. A escolha é um pouco controversa, e uma implementação híbrida também é possível.

O primeiro método é colocar o pacote de threads inteiramente no espaço do usuário. O núcleo não sabe nada a respeito deles. Até onde o núcleo sabe, ele está gerenciando processos comuns de um único thread. A primeira vantagem, e mais óbvia, é que o pacote de threads no nível do usuário pode ser implementado em um sistema operacional que não dá suporte aos threads.

Quando os threads são gerenciados no espaço do usuário, cada processo precisa da sua própria tabela de threads privada para controlá-los naquele processo.

Eles permitem que cada processo tenha seu próprio algoritmo de escalonamento customizado. Para algumas aplicações, por exemplo, aquelas com um thread coletor de lixo, é uma vantagem não ter de se preocupar com um thread ser parado em um momento inconveniente. Eles também escalam melhor, já que threads de núcleo sempre exigem algum espaço de tabela e de pilha no núcleo, o que pode ser um problema se houver um número muito grande de threads.

Apesar do seu melhor desempenho, pacotes de threads de usuário têm alguns problemas importantes. Primeiro, o problema de como chamadas de sistema bloqueantes são implementadas. Suponha que um thread leia de um teclado antes que quaisquer teclas tenham sido acionadas. Deixar que o thread realmente faça a chamada de sistema é algo inaceitável, visto que isso parará todos os threads. Uma das principais razões para ter

threads era permitir que cada um utilizasse chamadas com bloqueio, enquanto evitaria que um thread bloqueado afetasse os outros. Com chamadas de sistema bloqueantes, é difícil ver como essa meta pode ser alcançada prontamente.

Outro problema com pacotes de threads de usuário é que se um thread começa a ser executado, nenhum outro naquele processo será executado a não ser que o primeiro thread voluntariamente abra mão da CPU.

Outro — e o mais devastador — argumento contra threads de usuário é que os programadores geralmente desejam threads precisamente em aplicações nas quais eles são bloqueados com frequência, por exemplo, em um servidor web com múltiplos threads. Esses threads estão constantemente fazendo chamadas de sistema.

Implementando threads no espaço do núcleo

Quando um thread quer criar um novo ou destruir um existente, ele faz uma chamada de núcleo, que então faz a criação ou a destruição atualizando a tabela de threads do núcleo.

Quando um thread é destruído, ele é marcado como não executável, mas suas estruturas de dados de núcleo não são afetadas de outra maneira. Depois, quando um novo thread precisa ser criado, um antigo é reativado, evitando parte da sobrecarga.

Threads de núcleo não exigem quaisquer chamadas de sistema novas e não bloqueantes. Além disso, se um thread em um processo provoca uma falta de página, o núcleo pode facilmente conferir para ver se o processo tem quaisquer outros threads executáveis e, se assim for, executar um deles enquanto espera que a página exigida seja trazida do disco. Sua principal desvantagem é que o custo de uma chamada de sistema é substancial, então se as operações de thread (criação, término etc.) forem frequentes, ocorrerá uma sobrecarga muito maior.

Embora threads de núcleo sejam melhores do que threads de usuário em certos aspectos-chave, eles são também indiscutivelmente mais lentos.

Implementações híbridas

Várias maneiras foram investigadas para tentar combinar as vantagens de threads de usuário com threads de núcleo. Uma maneira é usar threads de núcleo e então multiplexar os de usuário em alguns ou todos eles. Quando essa abordagem é usada, o programador pode determinar quantos threads de núcleo usar e quantos threads de usuário multiplexar para cada um. Esse modelo proporciona o máximo em flexibilidade.

Com essa abordagem, o núcleo está consciente apenas dos threads de núcleo e os escalona. Alguns desses threads podem ter, em cima deles, múltiplos threads de usuário

multiplexados, os quais são criados, destruídos e escalonados exatamente como threads de usuário em um processo executado em um sistema operacional sem capacidade de múltiplos threads. Nesse modelo, cada thread de núcleo tem algum conjunto de threads de usuário que se revezam para usá-lo.

Threads pop-up

Threads costumam ser úteis em sistemas distribuídos. Um exemplo importante é como mensagens que chegam são tratadas.

A abordagem tradicional é ter um processo ou thread que esteja bloqueado em uma chamada de sistema receive esperando pela mensagem que chega. Quando uma mensagem chega, ela é aceita, aberta, seu conteúdo examinado e processada. No entanto, uma abordagem completamente diferente também é possível, na qual a chegada de uma mensagem faz o sistema criar um novo thread para lidar com a mensagem. Esse thread é chamado de thread pop-up.

Uma vantagem fundamental de threads pop-up é que como são novos, eles não têm história alguma — registradores, pilha, o que quer que seja — que devem ser restaurados.

Executar um thread pop-up no espaço núcleo normalmente é mais fácil e mais rápido do que colocá-lo no espaço do usuário. Também, um thread pop-up no espaço núcleo consegue facilmente acessar todas as tabelas do núcleo e os dispositivos de E/S, que podem ser necessários para o processamento de interrupções. Por outro lado, um thread de núcleo com erros pode causar mais danos que um de usuário com erros. Por exemplo, se ele for executado por tempo demais e não liberar a CPU, dados que chegam podem ser perdidos para sempre.

Convertendo código de um thread em código multithread

Muitos programas existentes foram escritos para processos monothread. Convertê-los para multithreading é muito mais complicado do que pode parecer em um primeiro momento.

O código de um thread em geral consiste em múltiplas rotinas, exatamente como um processo. Essas rotinas podem ter variáveis locais, variáveis globais e parâmetros. Variáveis locais e de parâmetros não causam problema algum, mas variáveis que são globais para um thread, mas não globais para o programa inteiro, são um problema. Essas são variáveis que são globais no sentido de que muitos procedimentos dentro do thread as usam (como poderiam usar qualquer variável global), mas outros threads devem logicamente deixá-las sozinhas.

- Conflitos entre threads sobre o uso de uma variável global
- Threads podem ter variáveis globais individuais

Comunicação entre processos

Processos quase sempre precisam comunicar-se com outros processos. Por exemplo, em um pipeline do interpretador de comandos, a saída do primeiro processo tem de ser passada para o segundo, e assim por diante até o fim da linha. Então, há uma necessidade por comunicação entre os processos, de preferência de uma maneira bem estruturada sem usar interrupções.

Processos podem se comunicar uns com os outros usando primitivas de comunicação entre processos, por exemplo, semáforos, monitores ou mensagens. Essas primitivas são usadas para assegurar que jamais dois processos estejam em suas regiões críticas ao mesmo tempo, uma situação que leva ao caos. Um processo pode estar sendo executado, ser executável, ou bloqueado, e pode mudar de estado quando ele ou outro executar uma das primitivas de comunicação entre processos. A comunicação entre threads é similar.

Condições de corrida

Em alguns sistemas operacionais, processos que estão trabalhando juntos podem compartilhar de alguma memória comum que cada um pode ler e escrever.

A velocidade de execução relativa de processos concorrentes pode gerar resultados diferentes.

Situações em que dois ou mais processos estão lendo ou escrevendo alguns dados compartilhados e o resultado final depende de quem executa precisamente e quando, são chamadas de condições de corrida.

Regiões críticas

Como evitar as condições de corrida? A chave para evitar problemas aqui e em muitas outras situações envolvendo memória compartilhada, arquivos compartilhados e tudo o mais compartilhado é encontrar alguma maneira de proibir mais de um processo de ler e escrever os dados compartilhados ao mesmo tempo.

Colocando a questão em outras palavras, o que precisamos é de exclusão mútua, isto é, alguma maneira de se certificar de que se um processo está usando um arquivo ou variável compartilhados, os outros serão impedidos de realizar a mesma coisa.

O problema de evitar condições de corrida também pode ser formulado de uma maneira abstrata. Durante parte do tempo, um processo está ocupado realizando computações internas e outras coisas que não levam a condições de corrida. No entanto, às vezes um processo tem de acessar uma memória compartilhada ou arquivos, ou realizar outras tarefas críticas que podem levar a corridas. Essa parte do programa onde a memória compartilhada é acessada é chamada de região crítica ou seção crítica.

Precisamos que quatro condições se mantenham para chegar a uma boa solução:

1. Dois processos jamais podem estar simultaneamente dentro de suas regiões críticas.

2. Nenhuma suposição pode ser feita a respeito de velocidades ou do número de CPUs.

3. Nenhum processo executando fora de sua região crítica pode bloquear qualquer processo.

4. Nenhum processo deve ser obrigado a esperar eternamente para entrar em sua região crítica.

Exclusão mútua com espera ocupada

• *Desabilitando interrupções*: Em um sistema de processador único, a solução mais simples é fazer que cada processo desabilite todas as interrupções logo após entrar em sua região crítica e as reabilitar um momento antes de partir.

Em geral, essa abordagem é pouco atraente, pois não é prudente dar aos processos de usuário o poder de desligar interrupções. E se um deles desligasse uma interrupção e nunca mais a ligasse de volta? Isso poderia ser o fim do sistema.

A conclusão é: desabilitar interrupções é muitas vezes uma técnica útil dentro do próprio sistema operacional, mas não é apropriada como um mecanismo de exclusão mútua geral para processos de usuário.

• *Variáveis tipo trava:* Como uma segunda tentativa, vamos procurar por uma solução de software. Considere ter uma única variável (de trava) compartilhada, inicialmente 0. Quando um processo quer entrar em sua região crítica, ele primeiro testa a trava. Se a trava é 0, o processo a configura para 1 e entra na região crítica. Se a trava já é 1, o processo apenas espera até que ela se torne 0. Desse modo, um 0 significa que nenhum processo está na região crítica, e um 1 significa que algum processo está em sua região crítica.

Infelizmente, essa ideia contém exatamente a mesma falha fatal que vimos no diretório de spool. Suponha que um processo lê a trava e vê que ela é 0. Antes que ele possa configurar a trava para 1, outro processo está escalonado, executa e configura a trava para 1. Quando o primeiro processo executa de novo, ele também configurará a trava para 1, e dois processos estarão nas suas regiões críticas ao mesmo tempo.

• *Alternância explícita:* A variável do tipo inteiro turn, inicialmente 0, serve para controlar de quem é a vez de entrar na região crítica e examinar ou atualizar a memória compartilhada. Inicialmente, o processo 0 inspeciona turn, descobre que ele é 0 e entra na sua região crítica. O processo 1 também encontra lá o valor 0 e, portanto, espera em um laço fechado testando continuamente turn para ver quando ele vira 1. Testar continuamente uma variável até que algum valor apareça é chamado de espera ocupada. Em geral ela deve ser evitada, já que desperdiça tempo da CPU. Apenas quando há uma expectativa razoável de que a

espera será curta, a espera ocupada é usada. Uma trava que usa a espera ocupada é chamada de trava giratória (spin lock).

Na realidade, essa solução exige que os dois processos alternem-se estritamente na entrada em suas regiões críticas para, por exemplo, enviar seus arquivos para o spool. Apesar de evitar todas as corridas, esse algoritmo não é realmente um sério candidato a uma solução, pois viola a condição 3.

• *Solução de Peterson*: Antes de usar as variáveis compartilhadas (isto é, antes de entrar na região crítica), cada processo chama enter_region com seu próprio número de processo, 0 ou 1, como parâmetro. Essa chamada fará que ele espere, se necessário, até que seja seguro entrar. Após haver terminado com as variáveis compartilhadas, o processo chama leave_region para indicar que ele terminou e para permitir que outros processos entrem, se assim desejarem.

Exclusão mútua com instrução test-and-set

Agora vamos examinar uma proposta que exige um pouco de ajuda do hardware. (Test and Set Lock — teste e configure trava) que funciona da seguinte forma: ele lê o conteúdo da palavra lock da memória para o registrador RX e então armazena um valor diferente de zero no endereço de memória lock. As operações de leitura e armazenamento da palavra são seguramente indivisíveis — nenhum outro processador pode acessar a palavra na memória até que a instrução tenha terminado. A CPU executando a instrução TSL impede o acesso ao barramento de memória para proibir que outras CPUs acessem a memória até ela terminar.

Exclusão mútua com instrução exchange

Uma instrução alternativa para TSL é XCHG, que troca os conteúdos de duas posições atomicamente; por exemplo, um registrador e uma palavra de memória. O código é essencialmente o mesmo que a solução com TSL. Todas as CPUs Intel x86 usam a instrução XCHG para a sincronização de baixo nível.

Problema do Produtor-Consumidor

Produtor: produz um item e insere no buffer (quando há espaço livre). Se o buffer estava vazio, acorda o Consumidor. Quando o buffer está cheio, espera.

Consumidor: acessa o buffer e, quando existe, retira um item. Quando o buffer está vazio, espera. Se retirou um item do buffer totalmente cheio, acorda o produtor.

Acesso ao buffer deve ser feito com exclusão mútua.

Operações sleep e wakeup

sleep(): interrompe o processo que chamou a operação sleep (coloca o processo para “dormir”)

wakeup(p): reativa o processo p (“acorda” o processo p)

Semáforos

Essa era a situação em 1965, quando E. W. Dijkstra (1965) sugeriu usar uma variável inteira para contar o número de sinais de acordar salvos para uso futuro. Em sua proposta, um novo tipo de variável, que ele chamava de semáforo, foi introduzido. Um semáforo podia ter o valor 0, indicando que nenhum sinal de despertar fora salvo, ou algum valor positivo se um ou mais sinais de acordar estivessem pendentes.

Dijkstra propôs ter duas operações nos semáforos, hoje normalmente chamadas de down e up (generalizações de sleep e wakeup, respectivamente).

Conferir o valor, modificá-lo e possivelmente dormir são feitos como uma única ação atômica indivisível.

A operação up incrementa o valor de um determinado semáforo. Se um ou mais processos estiverem dormindo naquele semáforo, incapaz de completar uma operação down anterior, um deles é escolhido pelo sistema (por exemplo, ao acaso) e é autorizado a completar seu down. Desse modo, após um up com processos dormindo em um semáforo, ele ainda estará em 0, mas haverá menos processos dormindo nele. A operação de incrementar o semáforo e despertar um processo também é indivisível. Nenhum processo é bloqueado realizando um up, assim como nenhum processo é bloqueado realizando um wakeup no modelo anterior

Possui uma variável inteira (S) que só pode assumir valor maior ou igual a 0 (zero), uma fila associada e duas operações primitivas, executadas atomicamente:

down(S):
```
if S == 0 sleep(); /* processo vai para a fila de S */
S = S – 1;
```

up(S):
```
if empty(fila_S) S = S + 1;
else wakeup(first(fila_S)); /* acorda primeiro da fila de S
```

Semáforos que são inicializados para 1 e usados por dois ou mais processos para assegurar que apenas um deles consiga entrar em sua região crítica de cada vez são chamados de semáforos binários.

O outro uso dos semáforos é para a sincronização. Os semáforos full e empty são necessários para garantir que determinadas sequências ocorram ou não. Nesse caso, eles

asseguram que o produtor pare de executar quando o buffer estiver cheio, e que o consumidor pare de executar quando ele estiver vazio. Esse uso é diferente da exclusão mútua.

Mutexes

Quando a capacidade do semáforo de fazer contagem não é necessária, uma versão simplificada, chamada mutex, às vezes é usada. Mutexes são bons somente para gerenciar a exclusão mútua de algum recurso ou trecho de código compartilhados. Eles são fáceis e eficientes de implementar, o que os torna especialmente úteis em pacotes de threads que são implementados inteiramente no espaço do usuário.

Um mutex é uma variável compartilhada que pode estar em um de dois estados: destravado ou travado. Em consequência, apenas 1 bit é necessário para representá- -lo, mas na prática muitas vezes um inteiro é usado, com 0 significando destravado e todos os outros valores significando travado.

Monitores

Técnica de sincronização proposta por Brinch, Hansen e Hoare (1974). Encapsulamento de estruturas de dados e procedimentos. Ideia similar a classes e métodos. Toda sincronização dos processos é feita no monitor usando variáveis de condição (cond) e primitivas explícitas para interromper (wait) e reativar (signal) os processos. Apenas um processo, por vez, “entra” no monitor (exclusão mútua garantida).

Um monitor é uma coleção de rotinas, variáveis e estruturas de dados que são reunidas em um tipo especial de módulo ou pacote. Processos podem chamar as rotinas em um monitor sempre que eles quiserem, mas eles não podem acessar diretamente as estruturas de dados internos do monitor a partir de rotinas declaradas fora dele.

Os monitores têm uma propriedade importante que os torna úteis para realizar a exclusão mútua: apenas um processo pode estar ativo em um monitor em qualquer dado instante.

Cabe ao compilador implementar a exclusão mútua nas entradas do monitor, mas uma maneira comum é usar um mutex ou um semáforo binário. Como o compilador, não o programador, está arranjando a exclusão mútua, é muito menos provável que algo dê errado. De qualquer maneira, a pessoa escrevendo o monitor não precisa ter ciência de como o compilador arranja a exclusão mútua. Basta saber que ao transformar todas as regiões críticas em rotinas de monitores, dois processos jamais executarão suas regiões críticas ao mesmo tempo.

Talvez você esteja pensando que as operações wait e signal parecem similares a sleep e wakeup, que vimos antes e tinham condições de corrida fatais. Bem, elas são muito

similares, mas com uma diferença crucial: sleep e wakeup fracassaram porque enquanto um processo estava tentando dormir, o outro tentava despertá- -lo. Com monitores, isso não pode acontecer. A exclusão mútua automática nas rotinas de monitor garante que se, digamos, o produtor dentro de uma rotina de monitor descobrir que o buffer está cheio, ele será capaz de completar a operação wait sem ter de se preocupar com a possibilidade de que o escalonador possa trocar para o consumidor um instante antes de wait ser concluída. O consumidor não será nem deixado entrar no monitor até que wait seja concluído e o produtor seja marcado como não mais executável.

Um problema com monitores, e também com semáforos, é que eles foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs, todas com acesso a uma memória comum. Podemos evitar as corridas ao colocar os semáforos na memória compartilhada e protegê-los com instruções TSL ou XCHG. Quando movemos para um sistema distribuído consistindo em múltiplas CPUs, cada uma com sua própria memória privada e conectada por uma rede de área local, essas primitivas tornam-se inaplicáveis. A conclusão é que os semáforos são de um nível baixo demais e os monitores não são utilizáveis, exceto em algumas poucas linguagens de programação. Além disso, nenhuma das primitivas permite a troca de informações entre máquinas.

Troca de Mensagens

Permite a sincronização de processos quando não há memória compartilhada(Sistemas distribuídos e redes). O modelo de troca de mensagens pode ser implementado em sistemas com memória compartilhada (mais genérico). Primitivas de envio (send) e recepção (receive). A comunicação pode ser síncrona (bloqueante) ou assíncrona (não bloqueante). A maioria dos sistemas implementa a receive bloqueante e a send não bloqueante.

Se nenhuma mensagem estiver disponível, o receptor pode bloquear até que uma chegue. Alternativamente, ele pode retornar imediatamente com um código de erro.

O consumidor começa enviando N mensagens vazias para o produtor. Sempre que o produtor tem um item para dar ao consumidor, ele pega uma mensagem vazia e envia de volta uma cheia. Assim, o número total de mensagens no sistema segue constante pelo tempo, de maneira que elas podem ser armazenadas em um determinado montante de memória previamente conhecido.

Se o produtor trabalhar mais rápido que o consumidor, todas as mensagens terminarão cheias, esperando pelo consumidor; o produtor será bloqueado, esperando por uma vazia voltar. Se o consumidor trabalhar mais rápido, então o inverso acontecerá: todas as mensagens estarão vazias esperando pelo produtor para enchê-las; o consumidor será bloqueado, esperando por uma mensagem cheia.

Barreiras

Sincronização de dois ou mais processos que cooperam para a execução de alguma tarefa, dividida em fases. Um processo só pode passar para a fase seguinte quando os demais processos também já tiverem completado a fase atual.

Quando um processo atinge a barreira, ele é bloqueado até que todos os processos tenham atingido a barreira. Isso permite que grupos de processos sincronizem.

Como exemplo de um problema exigindo barreiras, considere um problema típico de relaxação na física ou engenharia. Há tipicamente uma matriz que contém alguns valores iniciais. Os valores podem representar temperaturas em vários pontos em uma lâmina de metal. A ideia pode ser calcular quanto tempo leva para o efeito de uma chama colocada em um canto propagar-se através da lâmina.

Começando com os valores atuais, uma transformação é aplicada à matriz para conseguir a segunda versão; por exemplo, aplicando as leis da termodinâmica para ver quais serão todas as temperaturas posteriormente a ΔT. Então o processo é repetido várias vezes, fornecendo as temperaturas nos pontos de amostra como uma função do tempo à medida que a lâmina aquece. O algoritmo produz uma sequência de matrizes ao longo do tempo, cada uma para um determinado ponto no tempo.

Agora imagine que a matriz é muito grande (por exemplo, 1 milhão por 1 milhão), de maneira que processos paralelos sejam necessários (possivelmente em um multiprocessador) para acelerar o cálculo. Processos diferentes funcionam em partes diferentes da matriz, calculando os novos elementos de matriz a partir dos valores anteriores de acordo com as leis da física. No entanto, nenhum processo pode começar na iteração n + 1 até que a iteração n esteja completa, isto é, até que todos os processos tenham terminado seu trabalho atual. A maneira de se alcançar essa meta é programar cada processo para executar uma operação barrier após ele ter terminado sua parte da iteração atual. Quando todos tiverem terminado, a nova matriz (a entrada para a próxima iteração) será terminada, e todos os processos serão liberados simultaneamente para começar a próxima iteração.

Escalonamento

Quando um computador é multiprogramado, ele frequentemente tem múltiplos processos ou threads competindo pela CPU ao mesmo tempo. Essa situação ocorre sempre que dois ou mais deles estão simultaneamente no estado pronto.

Se apenas uma CPU está disponível, uma escolha precisa ser feita sobre qual processo será executado em seguida. A parte do sistema operacional que faz a escolha é chamada de escalonador, e o algoritmo que ele usa é chamado de algoritmo de escalonamento.

Nos velhos tempos dos sistemas em lote com a entrada na forma de imagens de cartões em uma fita magnética, o algoritmo de escalonamento era simples: apenas execute o próximo trabalho na fita.

Alguns computadores de grande porte ainda combinam serviço em lote e de compartilhamento de tempo, exigindo que o escalonador decida se um trabalho em lote ou um usuário interativo em um terminal deve ir em seguida.

Além de escolher o processo certo a ser executado, o escalonador também tem de se preocupar em fazer um uso eficiente da CPU, pois o chaveamento de processos é algo caro. Para começo de conversa, uma troca do modo usuário para o modo núcleo precisa ocorrer. Então o estado do processo atual precisa ser salvo, incluindo armazenar os seus registros na tabela de processos para que eles possam ser recarregados mais tarde. Em alguns sistemas, o mapa de memória (por exemplo, os bits de referência à memória na tabela de páginas) precisa ser salvo da mesma maneira. Em seguida, um novo processo precisa ser selecionado executando o algoritmo de escalonamento. Após isso, a MMU (memory management unit — unidade de gerenciamento de memória) precisa ser recarregada com o mapa de memória do novo processo. Por fim, o novo processo precisa ser inicializado. Além de tudo isso, a troca de processo pode invalidar o cache de memória e as tabelas relacionadas, forçando-o a ser dinamicamente recarregado da memória principal duas vezes (ao entrar no núcleo e ao deixá-lo). De modo geral, realizar muitas trocas de processos por segundo pode consumir um montante substancial do tempo da CPU, então, recomenda-se cautela.

Quase todos os processos alternam surtos de computação com solicitações de E/S (disco ou rede). Muitas vezes, a CPU executa por um tempo sem parar, então uma chamada de sistema é feita para ler de um arquivo ou escrever para um arquivo. Quando a chamada de sistema é concluída, a CPU calcula novamente até que ela precisa de mais dados ou tem de escrever mais dados, e assim por diante. Observe que algumas atividades de E/S contam como computação.

Os primeiros são chamados limitados pela computação ou limitados pela CPU; os segundos são chamados limitados pela E/S. Processos limitados pela CPU geralmente têm longos surtos de CPU e então esporádicas esperas de E/S, enquanto os processos limitados pela E/S têm surtos de CPU curtos e esperas de E/S frequentes. Observe que o fator chave é o comprimento do surto da CPU, não o comprimento do surto da E/S. Processos limitados pela E/S são limitados pela E/S porque eles não computam muito entre solicitações de E/S, não por terem tais solicitações especialmente demoradas. Eles levam o mesmo tempo para emitir o pedido de hardware para ler um bloco de disco, independentemente de quanto tempo levam para processar os dados após eles chegarem.

Quando escalonar?

● Criação de um processo : uma decisão precisa ser tomada a respeito de qual processo, o pai ou o filho, deve ser executado. Tendo em vista que ambos os processos estão em um estado pronto, trata-se de uma decisão de escalonamento normal e pode ser qualquer uma, isto é, o escalonador pode legitimamente escolher executar o processo pai ou o filho em seguida.

● Término de um processo: Esse processo não pode mais executar (já que ele não existe mais), então algum outro precisa ser escolhido do conjunto de processos prontos. Se nenhum está pronto, um processo ocioso gerado pelo sistema normalmente é executado

● Bloqueio para execução de E/S: em um semáforo, ou por alguma outra razão, outro processo precisa ser selecionado para executar. Às vezes, a razão para bloquear pode ter um papel na escolha. Por exemplo, se A é um processo importante e ele está esperando por B para sair de sua região crítica, deixar que B execute em seguida permitirá que ele saia de sua região crítica e desse modo deixe que A continue. O problema, no entanto, é que o escalonador geralmente não tem a informação necessária para levar essa dependência em consideração.

● Interrupção de dispositivo de E/S: Se a interrupção veio de um dispositivo de E/S que agora completou seu trabalho, algum processo que foi bloqueado esperando pela E/S pode agora estar pronto para executar.

● Limite de tempo (interrupção de relógio): Se um hardware de relógio fornece interrupções periódicas a 50 ou 60 Hz ou alguma outra frequência, uma decisão de escalonamento pode ser feita a cada interrupção ou a cada k-ésima interrupção de relógio.

Escalonamento preemptivo e não preemptivo

● Preemptivo: pode interromper o processo após um tempo máximo ou por regra de prioridade

● Não-preemptivo: processo usa o processador até chamar uma operação de E/S (ou terminar)

Realizar o escalonamento preemptivo exige que uma interrupção de relógio ocorra ao fim do intervalo para devolver o controle da CPU de volta para o escalonador. Se nenhum relógio estiver disponível, o escalonamento não preemptivo é a única solução.

Categorias de Algoritmos de Escalonamento

• Lote
    ● Ambiente não interativos (folha de pagamento, controle de estoque, etc.)

Sistemas em lote ainda são amplamente usados no mundo de negócios para folhas de pagamento, estoques, contas a receber, contas a pagar, cálculos de juros (em bancos), processamento de pedidos de indenização (em companhias de seguro) e outras tarefas periódicas. Em sistemas em lote, não há usuários esperando impacientemente em seus terminais para uma resposta rápida a uma solicitação menor. Em consequência, algoritmos não preemptivos, ou algoritmos preemptivos com longos períodos para cada processo são muitas vezes aceitáveis. Essa abordagem reduz os chaveamentos de processos e melhora

o desempenho. Na realidade, os algoritmos em lote são bastante comuns e muitas vezes aplicáveis a outras situações também, o que torna seu estudo interessante, mesmo para pessoas não envolvidas na computação corporativa de grande porte.

• Interativo
    - Preempção é fundamental

Em um ambiente com usuários interativos, a preempção é essencial para evitar que um processo tome conta da CPU e negue o serviço para os outros. Mesmo que nenhum processo execute de modo intencional para sempre, um erro em um programa pode levar um processo a impedir indefinidamente que todos os outros executem. A preempção é necessária para evitar esse comportamento. Os servidores também caem nessa categoria, visto que eles normalmente servem a múltiplos usuários (remotos), todos os quais estão muito apressados, assim como usuários de computadores.

• Tempo Real
    - Às vezes não é necessário utilizar preempção (processos curtos - CPU)

Em sistemas com restrições de tempo real, a preempção às vezes, por incrível que pareça, não é necessária, porque os processos sabem que eles não podem executar por longos períodos e em geral realizam o seu trabalho e bloqueiam rapidamente. A diferença com os sistemas interativos é que os de tempo real executam apenas programas que visam ao progresso da aplicação à mão. Sistemas interativos são sistemas para fins gerais e podem executar programas arbitrários que não são cooperativos e talvez até mesmo maliciosos.

Objetivos do algoritmo de escalonamento

Todos os sistemas

• Justiça: Processos comparáveis devem receber serviços comparáveis. Conceder a um processo muito mais tempo de CPU do que para um processo equivalente não é justo. É claro que categorias diferentes de processos podem ser tratadas diferentemente. Pense sobre controle de segurança e elaboração da folha de pagamento em um centro de computadores de um reator nuclear.

• Aplicação da política: De certa maneira relacionado com justiça está o cumprimento das políticas do sistema. Se a política local é que os processos de controle de segurança são executados sempre que quiserem, mesmo que isso signifique atraso de 30 segundos da folha de pagamento, o escalonador precisa certificar-se de que essa política seja cumprida.

• Equilíbrio

Sistemas em lote

• Vazão: A vazão é o número de tarefas por hora que o sistema completa. Considerados todos os fatores, terminar 50 tarefas por hora é melhor do que terminar 40 tarefas por hora.

• Tempo de retorno: O tempo de retorno é estatisticamente o tempo médio do momento em que a tarefa em lote é submetida até o momento em que ela é concluída. Ele mede quanto tempo o usuário médio tem de esperar pela saída. Aqui a regra é: menos é mais.

• Utilização de CPU: Um algoritmo de escalonamento que tenta maximizar a vazão talvez não minimize necessariamente o tempo de retorno. Por exemplo, dada uma combinação de tarefas curtas e tarefas longas, um escalonador que sempre executou tarefas curtas e nunca as longas talvez consiga uma excelente vazão (muitas tarefas curtas por hora), mas à custa de um tempo de retorno terrível para as tarefas longas. Se as tarefas curtas seguissem chegando a uma taxa aproximadamente uniforme, as tarefas longas talvez nunca fossem executadas, tornando o tempo de retorno médio infinito, conquanto alcançando uma alta vazão.

A utilização da CPU é muitas vezes usada como uma métrica nos sistemas em lote. No entanto, ela não é uma boa métrica. O que de fato importa é quantas tarefas por hora saem do sistema (vazão) e quanto tempo leva para receber uma tarefa de volta (tempo de retorno).

Sistemas Interativos

• Tempo de resposta: O tempo entre emitir um comando e receber o resultado. Em um computador pessoal, em que um processo de segundo plano está sendo executado (por exemplo, lendo e armazenando e-mail da rede), uma solicitação de usuário para começar um programa ou abrir um arquivo deve ter precedência sobre o trabalho de segundo plano. Atender primeiro todas as solicitações interativas será percebido como um bom serviço.

• Proporcionalidade: Usuários têm uma ideia inerente (porém muitas vezes incorreta) de quanto tempo as coisas devem levar. Quando uma solicitação que o usuário percebe como complexa leva muito tempo, os usuários aceitam isso, mas quando uma solicitação percebida como simples leva muito tempo, eles ficam irritados. Por exemplo, se clicar em um ícone que envia um vídeo de 500 MB para um servidor na nuvem demorar 60 segundos, o usuário provavelmente aceitará isso como um fato da vida por não esperar que a transferência leve 5 s. Ele sabe que levará um tempo.

Por outro lado, quando um usuário clica em um ícone que desconecta a conexão com o servidor na nuvem após o vídeo ter sido enviado, ele tem expectativas diferentes. Se a desconexão não estiver completa após 30 s, o usuário provavelmente estará soltando algum palavrão e após 60 s ele estará espumando de raiva. Esse comportamento decorre da percepção comum dos usuários de que enviar um monte de dados supostamente leva muito mais tempo que apenas desconectar uma conexão.

Sistemas de Tempo Real

• Cumprimento de Prazos: Por exemplo, se um computador está controlando um dispositivo que produz dados a uma taxa regular, deixar de executar o processo de coleta de dados em tempo pode resultar em dados perdidos. Assim, a principal exigência de um sistema de tempo real é cumprir com todos (ou a maioria) dos prazos.

• Previsibilidade: Descumprir um prazo ocasional não é fatal, mas se o processo de áudio executar de maneira errática demais, a qualidade do som deteriorará rapidamente. O vídeo também é uma questão, mas o ouvido é muito mais sensível a atrasos que o olho. Para evitar esse problema, o escalonamento de processos deve ser altamente previsível e regular.

Escalonamento em Sistemas em lote

Primeiro a chegar, primeiro a ser servido

É provável que o mais simples de todos os algoritmos de escalonamento já projetados seja o primeiro a chegar, primeiro a ser servido (first-come, first- -served) não preemptivo. Com esse algoritmo, a CPU é atribuída aos processos na ordem em que a requisitam. Basicamente, há uma fila única de processos prontos. Quando a primeira tarefa entrar no sistema de manhã, ela é iniciada imediatamente e deixada executar por quanto tempo ela quiser.

Tarefa mais curta primeiro(SJF)

Algoritmo em lote não preemptivo que presume que os tempos de execução são conhecidos antecipadamente. Vale a pena destacar que a tarefa mais curta primeiro é ótima apenas quando todas as tarefas estão disponíveis simultaneamente. (NÃO É POSSÍVEL IMPLEMENTAR POIS O ESCALONADOR NÃO CONHECE O TEMPO DOS PROCESSOS).

Tempo restante mais curto em seguida(SRTN)

Uma versão preemptiva da tarefa mais curta primeiro é o tempo restante mais curto em seguida (shortest remaining time next). Com esse algoritmo, o escalonador escolhe o processo cujo tempo de execução restante é o mais curto. De novo, o tempo de execução precisa ser conhecido antecipadamente. Quando uma nova tarefa chega, seu tempo total é comparado com o tempo restante do processo atual. Se a nova tarefa precisa de menos tempo para terminar do que o processo atual, este é suspenso e a nova tarefa iniciada. Esse esquema permite que tarefas curtas novas tenham um bom desempenho. (NÃO É POSSÍVEL IMPLEMENTAR POIS O ESCALONADOR NÃO CONHECE O TEMPO DOS PROCESSOS).

Escalonamento em Sistemas interativos

Escalonamento por chaveamento circular (RR)

Um dos algoritmos mais antigos, simples, justos e amplamente usados é o circular (round-robin). A cada processo é designado um intervalo, chamado de seu quantum, durante o qual ele é deixado executar. Se o processo ainda está executando ao fim do quantum, a CPU sofrerá uma preempção e receberá outro processo. Se o processo foi bloqueado ou terminado antes de o quantum ter decorrido, o chaveamento de CPU será feito quando o processo bloquear, é claro. O escalonamento circular é fácil de implementar.

O escalonamento circular pressupõe implicitamente que todos os processos são de igual importância.

Escalonamento por prioridades

Em uma universidade, por exemplo, uma ordem hierárquica começaria pelo reitor, os chefes de departamento em seguida, então os professores, secretários, zeladores e, por fim, os estudantes. A necessidade de levar em consideração fatores externos leva ao escalonamento por prioridades. A ideia básica é direta: a cada processo é designada uma prioridade, e o processo executável com a prioridade mais alta é autorizado a executar.

Para evitar que processos de prioridade mais alta executem indefinidamente, o escalonador talvez diminua a prioridade do processo que está sendo executado em cada tique do relógio (isto é, em cada interrupção do relógio). Se essa ação faz que a prioridade caia abaixo daquela do próximo processo com a prioridade mais alta, ocorre um chaveamento de processo. Como alternativa, pode ser designado a cada processo um quantum de tempo máximo no qual ele é autorizado a executar. Quando esse quantum for esgotado, o processo seguinte na escala de prioridade recebe uma chance de ser executado.

Prioridades podem ser designadas a processos estaticamente ou dinamicamente. Prioridades também podem ser designadas dinamicamente pelo sistema para alcançar determinadas metas. Por exemplo, alguns processos são altamente limitados pela E/S e passam a maior parte do tempo esperando para a E/S ser concluída. Sempre que um processo assim quer a CPU, ele deve recebê-la imediatamente, para deixá-lo iniciar sua próxima solicitação de E/S, que pode então proceder em paralelo com outro processo que estiver de fato computando. Fazer que o processo limitado pela E/S espere muito tempo pela CPU significará apenas tê-lo ocupando a memória por um tempo desnecessariamente longo.

Múltiplas filas

Os projetistas do CTSS logo perceberam que era mais eficiente dar aos processos limitados pela CPU um grande quantum de vez em quando, em vez de dar a eles pequenos quanta frequentemente (para reduzir as operações de troca). Por outro lado, dar a todos os processos um grande quantum significaria um tempo de resposta ruim, como já vimos. A solução foi estabelecer classes de prioridade. Processos na classe mais alta seriam executados por dois quanta. Processos na classe seguinte seriam executados por quatro quanta etc. Sempre que um processo consumia todos os quanta alocados para ele, era movido para uma classe inferior.

Como exemplo, considere um processo que precisasse computar continuamente por 100 quanta. De início ele receberia um quantum, então seria trocado. Da vez seguinte, ele receberia dois quanta antes de ser trocado. Em sucessivas execuções ele receberia 4, 8, 16, 32 e 64 quanta, embora ele tivesse usado apenas 37 dos 64 quanta finais para completar o trabalho. Apenas 7 trocas seriam necessárias (incluindo a carga inicial) em vez de 100 com um algoritmo circular puro. Além disso, à medida que o processo se aprofundasse nas filas de prioridade, ele seria usado de maneira cada vez menos frequente, poupando a CPU para processos interativos curtos.

Processo mais curto em seguida

Como a tarefa mais curta primeiro sempre produz o tempo de resposta médio mínimo para sistemas em lote, seria bom se ela pudesse ser usada para processos interativos também. Até certo ponto, ela pode ser. Processos interativos geralmente seguem o padrão de esperar pelo comando, executar o comando, esperar pelo comando, executar o comando etc. Se considerarmos a execução de cada comando uma “tarefa” em separado, então podemos minimizar o tempo de resposta geral executando a tarefa mais curta primeiro. O problema é descobrir qual dos processos atualmente executáveis é o mais curto. Uma abordagem é fazer estimativas baseadas no comportamento passado e executar o processo com o tempo de execução estimado mais curto.

Escalonamento garantido

Uma abordagem completamente diferente para o escalonamento é fazer promessas reais para os usuários a respeito do desempenho e então cumpri-las. Uma promessa realista de se fazer e fácil de cumprir é a seguinte: se n usuários estão conectados enquanto você está trabalhando, você receberá em torno de 1/n da potência da CPU. De modo similar, em um sistema de usuário único com n processos sendo executados, todos os fatores permanecendo os mesmos, cada um deve receber 1/n dos ciclos da CPU. Isso parece bastante justo.

Para cumprir essa promessa, o sistema deve controlar quanta CPU cada processo teve desde sua criação. Ele então calcula o montante de CPU a que cada um tem direito,

especificamente, o tempo desde a criação dividido por n. Tendo em vista que o montante de tempo da CPU que cada processo realmente teve também é conhecido, calcular o índice de tempo de CPU real consumido com o tempo de CPU ao qual ele tem direito é algo bastante direto. Um índice de 0,5 significa que o processo teve apenas metade do que deveria, e um índice de 2,0 significa que teve duas vezes o montante de tempo ao qual ele tinha direito. O algoritmo então executará o processo com o índice mais baixo até que seu índice aumente e se aproxime do de seu competidor. Então este é escolhido para executar em seguida.

Escalonamento por loteria

A ideia básica é dar bilhetes de loteria aos processos para vários recursos do sistema, como o tempo da CPU. Sempre que uma decisão de escalonamento tiver de ser feita, um bilhete de loteria será escolhido ao acaso, e o processo com o bilhete fica com o recurso. Quando aplicado ao escalonamento de CPU, o sistema pode realizar um sorteio 50 vezes por segundo, com cada vencedor recebendo 20 ms de tempo da CPU como prêmio.

Processos são iguais, mas alguns processos são mais iguais". Processos mais importantes podem receber bilhetes extras, para aumentar a chance de vencer.

Processos cooperativos podem trocar bilhetes se assim quiserem. Por exemplo, quando um processo cliente envia uma mensagem para um processo servidor e então bloqueia, ele pode dar todos os seus bilhetes para o servidor a fim de aumentar a chance de que o servidor seja executado em seguida. Quando o servidor tiver concluído, ele devolve os bilhetes de maneira que o cliente possa executar novamente. Na realidade, na ausência de clientes, os servidores não precisam de bilhete algum.

Escalonamento por fração justa

Até agora presumimos que cada processo é escalonado por si próprio, sem levar em consideração quem é o seu dono. Como resultado, se o usuário 1 inicia nove processos e o usuário 2 inicia um processo, com chaveamento circular ou com prioridades iguais, o usuário 1 receberá 90% da CPU e o usuário 2 apenas 10% dela.

Para evitar essa situação, alguns sistemas levam em conta qual usuário é dono de um processo antes de escaloná-lo. Nesse modelo, a cada usuário é alocada alguma fração da CPU e o escalonador escolhe processos de uma maneira que garanta essa fração. Desse modo, se dois usuários têm cada um 50% da CPU prometidos, cada um receberá isso, não importa quantos processos eles tenham em existência.

Escalonamento em Sistemas de tempo real

Sistemas em tempo real são geralmente categorizados como tempo real crítico, significando que há prazos absolutos que devem ser cumpridos — para valer! — e tempo real não

crítico, significando que descumprir um prazo ocasional é indesejável, mas mesmo assim tolerável.

Em ambos os casos, o comportamento em tempo real é conseguido dividindo o programa em uma série de processos, cada um dos quais é previsível e conhecido antecipadamente. Esses processos geralmente têm vida curta e podem ser concluídos em bem menos de um segundo. Quando um evento externo é detectado, cabe ao escalonador programar os processos de uma maneira que todos os prazos sejam atendidos.

Os eventos a que um sistema de tempo real talvez tenha de responder podem ser categorizados ainda como periódicos (significando que eles ocorrem em intervalos regulares) ou aperiódicos (significando que eles ocorrem de maneira imprevisível). Um sistema pode ter de responder a múltiplos fluxos de eventos periódicos.

Dependendo de quanto tempo cada evento exige para o processamento, tratar de todos talvez não seja nem possível. Por exemplo, se há m eventos periódicos e o evento i ocorre com o período Pi e exige Ci segundos de tempo da CPU para lidar com cada evento, então a carga só pode ser tratada se

$$\sum_{1}^{m} (Ci/Pi) <= 1$$

Diz-se de um sistema de tempo real que atende a esse critério que ele é escalonável. Isso significa que ele realmente pode ser implementado. Um processo que fracassa em atender esse teste não pode ser escalonado, pois o montante total de tempo de CPU que os processos querem coletivamente é maior do que a CPU pode proporcionar.

Algoritmos de escalonamento de tempo real podem ser estáticos ou dinâmicos. Os primeiros tomam suas decisões de escalonamento antes de o sistema começar a ser executado. Os últimos tomam suas decisões no tempo de execução, após ela ter começado. O escalonamento estático funciona apenas quando há uma informação perfeita disponível antecipadamente sobre o trabalho a ser feito, e os prazos que precisam ser cumpridos. Algoritmos de escalonamento dinâmico não têm essas restrições.

Problemas clássicos de IPC

O problema do jantar dos filósofos

O problema pode ser colocado de maneira bastante simples, como a seguir: cinco filósofos estão sentados em torno de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete é tão escorregadio que um filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos há um garfo.

Quando um filósofo fica suficientemente faminto, ele tenta pegar seus garfos à esquerda e à direita, um de cada vez, não importa a ordem. Se for bem-sucedido em pegar dois garfos, ele come por um tempo, então larga os garfos e continua a pensar.

A questão fundamental é: você consegue escrever um programa para cada filósofo que faça o que deve fazer e jamais fique travado?

Suponha que todos os cinco filósofos peguem seus garfos esquerdos simultaneamente. Nenhum será capaz de pegar seus garfos direitos, e haverá um impasse.

Poderíamos facilmente modificar o programa de maneira que após pegar o garfo esquerdo, o programa confere para ver se o garfo direito está disponível. Se não estiver, o filósofo coloca de volta o esquerdo sobre a mesa, espera por um tempo, e repete todo o processo. Essa proposta também fracassa, embora por uma razão diferente. Com um pouco de azar, todos os filósofos poderiam começar o algoritmo simultaneamente, pegando seus garfos esquerdos, vendo que seus garfos direitos não estavam disponíveis, colocando seus garfos

esquerdos de volta sobre a mesa, esperando, pegando seus garfos esquerdos de novo ao mesmo tempo, assim por diante, para sempre. Uma situação como essa, na qual todos os programas continuam a executar indefinidamente, mas fracassam em realizar qualquer progresso, é chamada de inanição (starvation).

Solução 1:
Uma melhoria que não apresenta impasse nem inanição é proteger os cinco comandos seguindo a chamada think com um semáforo binário. Antes de começar a pegar garfos, um filósofo realizaria um down em mutex. Após substituir os garfos, ele realizaria um up em mutex. Do ponto de vista teórico, essa solução é adequada. Do ponto de vista prático, ela tem um erro de desempenho: apenas um filósofo pode estar comendo a qualquer dado instante. Com cinco garfos disponíveis, deveríamos ser capazes de ter dois filósofos comendo ao mesmo tempo.

Solução 2:
A solução apresentada é livre de impasse e permite o máximo paralelismo para um número arbitrário de filósofos. Ela usa um arranjo, estado, para controlar se um filósofo está comendo, pensando, ou com fome (tentando conseguir garfos). Um filósofo pode passar para o estado comendo apenas se nenhum de seus vizinhos estiver comendo. Os vizinhos do filósofo i são definidos pelas macros LEFT e RIGHT.

O problema dos leitores e escritores

Imagine, por exemplo, um sistema de reservas de uma companhia aérea, com muitos processos competindo entre si desejando ler e escrever.

É aceitável ter múltiplos processos lendo o banco de dados ao mesmo tempo, mas se um processo está atualizando (escrevendo) o banco de dados, nenhum outro pode ter acesso, nem mesmo os leitores.

A questão é: como programar leitores e escritores?

Solução 1:
Nessa solução, para conseguir acesso ao banco de dados, o primeiro leitor realiza um down no semáforo db. Leitores subsequentes apenas incrementam um contador, rc. À medida que os leitores saem, eles decrementam o contador, e o último a deixar realiza um up no semáforo, permitindo que um escritor bloqueado, se houver, entre.

A solução apresentada aqui contém implicitamente uma decisão sutil que vale observar. Suponha que enquanto um leitor está usando o banco de dados, aparece outro leitor. Visto que ter dois leitores ao mesmo tempo não é um problema, o segundo leitor é admitido. Leitores adicionais também podem ser admitidos se aparecerem.

Agora suponha que um escritor apareça. O escritor pode não ser admitido ao banco de dados, já que escritores precisam ter acesso exclusivo, então ele é suspenso. Depois, leitores adicionais aparecem. Enquanto pelo menos um leitor ainda estiver ativo, leitores subsequentes serão admitidos. Como consequência dessa estratégia, enquanto houver

uma oferta uniforme de leitores, todos eles entrarão assim que chegarem. O escritor será mantido suspenso até que nenhum leitor esteja presente. Se um novo leitor aparecer, digamos, a cada 2 s, e cada leitor levar 5 s para realizar o seu trabalho, o escritor jamais entrará.

Solução 2:
Para evitar essa situação, o programa poderia ser escrito de maneira ligeiramente diferente: quando um leitor chega e um escritor está esperando, o leitor é suspenso atrás do escritor em vez de ser admitido imediatamente. Dessa maneira, um escritor precisa esperar por leitores que estavam ativos quando ele chegou, mas não precisa esperar por leitores que chegaram depois dele. A desvantagem dessa solução é que ela alcança uma concorrência menor e assim tem um desempenho mais baixo.

Pesquisas sobre processos e threads

Como um todo, processos, threads e escalonamento, não são mais os tópicos quentes de pesquisa que já foram um dia. A pesquisa seguiu para tópicos como gerenciamento de energia, virtualização, nuvens e segurança.