Sistemas com múltiplos processadores

Todos querem mais ciclos de CPU. Não importa quanta potência computacional exista, ela nunca será suficiente. Quanto mais rápido um computador executar, mais calor ele gerará, e quanto menor o computador, mais difícil é se livrar desse calor. Um meio de aumentar a velocidade é com computadores altamente paralelos. Essas máquinas consistem em muitas CPUs, cada uma delas executando a uma velocidade “normal”.

Computadores altamente paralelos são usados com frequência para processamento pesado de computação numérica. Problemas como prever o clima, modelar o fluxo de ar em torno da asa de uma aeronave, simular a economia mundial ou compreender interações de receptores de drogas no cérebro são atividades computacionalmente intensivas. Suas soluções exigem longas execuções em muitas CPUs ao mesmo tempo.

Colocar 1 milhão de computadores não relacionados em uma sala é algo fácil de se fazer, desde que você tenha dinheiro suficiente e uma sala grande o bastante. Espalhar 1 milhão de computadores não relacionados mundo afora é mais fácil ainda, pois resolve o segundo problema. A dificuldade surge quando você quer que eles se comuniquem entre si para trabalhar juntos em um único problema. Em consequência, muito trabalho foi investido na tecnologia de interconexão, e diferentes tecnologias de interconexão levaram a tipos de sistemas qualitativamente distintos e diferentes organizações de software.

(a) Um multiprocessador de memória compartilhada.
(b) Um multicomputador de troca de mensagens.
(c) Sistema distribuído com rede de longa distância.

Toda a comunicação entre os componentes eletrônicos (ou óticos) em última análise resume-se a enviar mensagens — cadeias de bits bem definidas — entre eles. As diferenças encontram-se na escala de tempo, escala de distância e organização lógica envolvida. Em um extremo encontram-se os multiprocessadores de memória compartilhada, nos quais algo entre duas e 1.000 CPUs se comunicam via uma memória compartilhada. Nesse modelo, toda CPU tem acesso igual a toda a memória física e pode ler e escrever palavras individuais usando instruções LOAD e STORE. Acessar uma palavra de memória normalmente leva de 1-10 ns. Como veremos, atualmente é comum colocar mais do que um núcleo de processamento em um único chip de CPU, com os núcleos compartilhando acesso à memória principal (e às vezes até compartilhando caches). Em outras palavras, o modelo de múltiplos computadores de memória compartilhada pode ser implementado usando CPUs fisicamente separadas, múltiplos núcleos em uma única CPU, ou uma combinação desses fatores.

Em seguida vem o sistema da Figura 8.1(b) na qual os pares de CPU-memória estão conectados por uma interconexão de alta velocidade. Esse tipo de sistema é chamado de multicomputador de troca de mensagens. Cada memória é local a uma única CPU e pode ser acessada somente por aquela CPU. As CPUs comunicam-se enviando múltiplas mensagens via interconexão. Com uma boa interconexão, uma mensagem curta pode ser enviada em 10-50 μs, que ainda assim é um tempo muito mais longo do que o tempo de

acesso da memória na Figura 8.1(a). Não há uma memória global compartilhada nesse projeto. Multicomputadores (isto é, sistemas de trocas de mensagens) são muito mais fáceis de construir do que multiprocessadores (de memória compartilhada), mas eles são mais difíceis de programar.

O terceiro modelo, que está ilustrado (c), conecta sistemas de computadores completos via uma rede de longa distância, como a internet, para formar um sistema distribuído. Cada um desses tem sua própria memória e os sistemas comunicam-se por troca de mensagens. A única diferença real entre a (b) e a (c) é que, na segunda, computadores completos são usados e os tempos das mensagens são muitas vezes 10-100 ms. Esse longo atraso força esses sistemas fracamente acoplados a serem usados de maneiras diferentes das dos sistemas fortemente acoplados da Figura 8.1(b). Os três tipos de sistemas diferem em seus atrasos por algo em torno de três ordens de magnitude. Essa é a diferença entre um dia e três anos.

Multiprocessadores

Um multiprocessador de memória compartilhada é um sistema de computadores no qual duas ou mais CPUs compartilham acesso total a uma RAM comum. Um programa executando em qualquer uma das CPUs vê um espaço de endereçamento virtual normal (geralmente paginado). A única propriedade incomum que esse sistema tem é que a CPU pode escrever algum valor em uma palavra de memória e então ler a palavra de volta e receber um valor diferente (porque outra CPU o alterou). Quando organizada corretamente, essa propriedade forma a base da comunicação entre processadores: uma CPU escreve alguns dados na memória e outra lê esses dados.

Na maioria dos casos, sistemas operacionais de multiprocessadores são sistemas operacionais normais. Eles lidam com chamadas de sistema, realizam gerenciamento de memória, fornecem um sistema de arquivos e gerenciam dispositivos de E/S. Mesmo assim, existem algumas áreas nas quais eles possuem características especiais. Elas incluem a sincronização de processos, gerenciamento de recursos e escalonamento.

Multiprocessadores são populares e atraentes porque eles oferecem um modelo de comunicação simples: todas as CPUs compartilham uma memória comum.

Hardware de multiprocessador

Embora todos os multiprocessadores tenham a propriedade de que cada CPU pode endereçar toda a memória, alguns multiprocessadores têm a propriedade adicional de que cada palavra de memória pode ser lida tão rapidamente quanto qualquer outra palavra de memória. Essas máquinas são chamadas de multiprocessadores UMA (Uniform Memory Access — acesso uniforme à memória). Em comparação, multiprocessadores NUMA (Nonuniform Memory Access — acesso não uniforme à memória) não têm essa propriedade. Por que essa diferença existe tornar-se-á claro mais tarde. Examinaremos primeiro os multiprocessadores UMA e então seguiremos para os multiprocessadores NUMA.

Multiprocessadores UMA com arquiteturas baseadas em barramento

Os multiprocessadores mais simples são baseados em um único barramento. Duas ou mais CPUs e um ou mais módulos de memória usam o mesmo barramento para comunicação. Quando uma CPU quer ler uma palavra de memória, ela primeiro confere para ver se o barramento está ocupado. Se o barramento estiver ocioso, a CPU coloca o endereço da palavra que ela quer no barramento, envia alguns sinais de controle e espera até que a memória coloque a palavra desejada no barramento.

Se o barramento estiver ocupado quando uma CPU quiser ler ou escrever na memória, a CPU simplesmente espera até que o barramento se torne ocioso. Aqui se encontra o problema com esse projeto. Com duas ou três CPUs, a contenção para o barramento será gerenciável; com 32 ou 64 será insuportável. O sistema será totalmente limitado pela largura de banda do barramento, e a maioria das CPUs ficará ociosa a maior parte do tempo.

A solução para esse problema é adicionar uma cache para cada CPU. Quando uma palavra é referenciada, o seu bloco inteiro, chamado de linha de cache, é trazido para a cache da CPU que a referenciou.

Cada bloco de cache é marcado como somente de leitura (nesse caso, ele pode estar presente em múltiplas caches ao mesmo tempo) ou de leitura-escrita (nesse caso, ele não pode estar presente em nenhuma outra cache). Se uma CPU tenta escrever uma palavra que está em uma ou mais caches remotas, o hardware do barramento detecta a palavra e coloca um sinal no barramento informando todas as outras caches a respeito da escrita. Se outras caches têm uma cópia “limpa”, isto é, uma cópia exata do que está na memória, elas podem apenas descartar suas cópias e deixar que o escritor busque o bloco da cache da memória antes de modificá-lo. Se alguma outra cache tem uma cópia “suja” (isto é, modificada), ela deve escrever de volta para a memória antes que a escrita possa proceder ou transferi-la diretamente para o escritor pelo barramento. Esse conjunto de regras é chamado de protocolo de coerência de cache e é um entre muitos que existem.

Outra possibilidade ainda é que cada CPU não tem apenas uma cache, mas também uma memória local, privada, que ela acessa por um barramento (privado) dedicado. Para usar essa configuração de maneira otimizada, o compilador deve colocar nas memórias privadas todo o código do programa, cadeias de caracteres, constantes e outros dados somente de leitura, pilhas e variáveis locais. A memória compartilhada é usada então somente para variáveis compartilhadas que podem ser escritas. Na maioria dos casos, essa colocação cuidadosa reduzirá em muito o tráfego de barramento, mas exige uma cooperação ativa do compilador.

Multiprocessadores UMA que usam barramentos cruzados

Mesmo com o melhor sistema de cache, o uso de um único barramento limita o tamanho de um multiprocessador UMA para cerca de 16 ou 32 CPUs. Para ir além disso, é necessário um tipo diferente de rede de interconexão. O circuito mais simples para conectar n CPUs a k memórias é o barramento cruzado (crossbar switch).

Em cada interseção de uma linha horizontal (que chega) e uma linha vertical (que sai) há uma interseção (crosspoint). Uma interseção é uma pequena chave eletrônica que pode ser aberta ou fechada eletronicamente, dependendo de as linhas horizontais e verticais estarem conectadas ou não.

Uma das melhores propriedades do barramento cruzado é que ele é uma rede não bloqueante, significa que nenhuma CPU tem negada a conexão da qual ela precisa porque alguma interseção ou linha já está ocupada (presumindo que o módulo da memória em si está disponível). Nem todas as interconexões têm essa bela propriedade. Além disso, nenhum planejamento antecipado é necessário. Mesmo que sete conexões arbitrárias já estejam estabelecidas, sempre é possível conectar a CPU restante à memória restante.

Uma das piores propriedades da chave de crossbar é o fato de o número de interseções crescer como n^2.

Multiprocessadores UMA usando redes de comutação multiestágio

Um projeto de multiprocessador completamente diferente é baseado no comutador (switch) 2 × 2 simples Esse comutador tem duas entradas e duas saídas. Mensagens chegando de qualquer linha de entrada podem ser comutadas para qualquer linha de saída. Para nossos fins, as mensagens conterão até quatro partes, como mostrado na Figura 8.4(b). O campo Módulo diz qual memória usar. O endereço especifica um endereço dentro de um módulo. O CódigoOp fornece a operação, como READ ou WRITE. Por fim, o campo opcional Valor pode conter um operando, como uma palavra de 32 bits para ser escrita em um WRITE. O comutador inspeciona o campo Módulo e o utiliza para determinar se a mensagem deve ser enviada em X ou Y.

Nossos comutadores 2 × 2 podem ser arranjados de muitos modos para construir redes de comutação multiestágio maiores. Uma possibilidade é a rede ômega, simples e econômica, nela conectamos oito CPUs a oito memórias usane log2 n estágios, com n/2 comutadores por estágio, para um total de (n/2) log2 n comutadores, o que é muito melhor do que n^2 interseções, especialmente para grandes valores de n.do 12 chaves. De modo geral, para n CPUs e n memórias precisaríamos de log2 n estágios, com n/2 comutadores por estágio, para um total de (n/2) log2 n comutadores, o que é muito melhor do que n^2 interseções, especialmente para grandes valores de n.

O padrão de conexões da rede ômega é muitas vezes chamado de embaralhamento perfeito, tendo em vista que a mistura dos sinais em cada estágio lembra um baralho de cartas sendo cortado na metade e então misturado carta por carta.

Diferentemente do barramento cruzado, a rede ômega é uma rede bloqueante. Nem todo conjunto de solicitações pode ser processado simultaneamente. Conflitos podem ocorrer sobre o uso de um fio ou um comutador, assim como entre solicitações para a memória e respostas da memória.

Tendo em vista que é altamente desejável disseminar as referências de memória de maneira uniforme através dos módulos, uma técnica comum é usar os bits de baixa ordem como o número do módulo.

Um sistema de memória no qual palavras consecutivas estão em módulos diferentes é chamado de entrelaçado. Memórias entrelaçadas maximizam o paralelismo porque a maioria das referências de memória é para endereços consecutivos. Também é possível projetar redes de comutação que sejam não bloqueantes e ofereçam múltiplos caminhos de cada CPU para cada módulo de memória a fim de distribuir melhor o tráfego.

Multiprocessadores NUMA

Multiprocessadores UMA de barramento único são geralmente limitados a não mais do que algumas dúzias de CPUs, e multiprocessadores com barramento cruzado ou redes de comutação precisam de muito hardware (caro) e não são tão maiores assim. Para conseguir mais do que 100 CPUs, algo tem de ceder. Em geral, o que cede é a ideia de que todos os módulos de memória tenham o mesmo tempo de acesso. Essa concessão leva à ideia de multiprocessadores NUMA, como mencionado. De maneira semelhante aos seus primos UMA, eles fornecem um espaço de endereçamento único através de todas as CPUs, mas diferentemente das máquinas UMA, o acesso aos módulos de memória locais é mais rápido do que o acesso aos módulos remotos. Desse modo, todos os programas UMA executarão sem mudança em máquinas NUMA, mas o desempenho será pior do que em uma máquina UMA.

Máquinas NUMA têm três características fundamentais que todas elas possuem e que juntas as distinguem de outros multiprocessadores:

1. Há um único espaço de endereçamento visível para todas as CPUs.
2. O acesso à memória remota é feito por instruções LOAD e STORE.
3. O acesso à memória remota é mais lento do que o acesso à memória local.

Quando o tempo de acesso à memória remota não é escondido (porque não há utilização de cache), o sistema é chamado de NC-NUMA (Non Cache-coherent NUMA — NUMA sem cache coerente). Quando as caches são coerentes, o sistema é chamado de CC-NUMA (Cache-Coherent NUMA — NUMA com cache coerente).

Uma abordagem popular na construção de grandes multiprocessadores CC-NUMA é o multiprocessador baseado em diretórios. A ideia é manter um banco de dados dizendo onde está cada linha de cache e qual é o seu status. Quando uma linha de cache é referenciada, o banco de dados é questionado para descobrir onde ela está e se está limpa ou suja. Como esse banco de dados é questionado sobre cada instrução que toque a memória, ele tem de ser armazenado em um hardware de propósito especial extremamente rápido que pode responder em uma fração de um ciclo de barramento.

Para permitir que as linhas sejam armazenadas em cache em múltiplos nós, precisaríamos, por exemplo, de alguma maneira de localizar todas elas, a fim de invalidá-las ou atualizá-las em uma escrita. Em muitos processadores de múltiplos núcleos, uma entrada de diretório, portanto, consiste em um vetor de bits com um bit por núcleo. Um “1” indica que a linha de cache está presente no núcleo, e um “0”, que ela não está. Além disso, cada entrada de diretório contém tipicamente alguns bits a mais. Como consequência, o custo de memória do diretório aumenta consideravelmente.

Chips multinúcleo

Com o avanço da tecnologia de fabricação de chips, os transistores estão ficando cada dia menores, e é possível colocar mais e mais deles em um chip. Essa observação empírica é muitas vezes chamada de Lei de Moore, em homenagem ao cofundador da Intel, Gordon Moore, que a observou pela primeira vez. Em 1974, o Intel 8080 continha um pouco mais de 2.000 transistores, enquanto as CPUs do Xeon Nehalem-EX têm mais de 2 bilhões de transistores.

Uma questão óbvia é: “O que você faz com todos esses transistores?”. Uma opção é adicionar megabytes de cache ao chip.

A outra opção é colocar duas ou mais CPUs completas, normalmente chamadas de núcleos, no mesmo chip (tecnicamente, na mesma pastilha). Chips com dois, quatro e oito núcleos já são comuns; e você pode até comprar chips com centenas de núcleos. Não há dúvida de que mais núcleos estão a caminho. Caches ainda são cruciais e estão agora espalhadas pelo chip.

Embora as CPUs possam ou não compartilhar caches, elas sempre compartilham a memória principal, e essa memória é consistente no sentido de que há sempre um valor único para cada palavra de memória. Circuitos de hardware especiais certificam-se de que se uma palavra está presente em duas ou mais caches e uma das CPUs modifica a palavra, ela é automática e atomicamente removida de todas as caches a fim de manter a consistência. Esse processo é conhecido como snooping (espionagem).

O resultado desse projeto é que chips multinúcleo são simplesmente multiprocessadores muito pequenos. Na realidade, esses chips são chamados às vezes de CMPs (Chip MultiProcessors — multiprocessadores em chip). A partir de uma perspectiva de software, CMPs não são realmente tão diferentes de multiprocessadores baseados em barramento ou de multiprocessadores que usam redes de comutação. No entanto, existem algumas diferenças. Para começo de conversa, em um multiprocessador baseado em barramento, cada uma das CPUs tem a sua própria cache. Uma cache L2 ou L3 compartilhada pode afetar o desempenho. Se um núcleo precisa de uma grande quantidade de memória de cache e outros não, esse projeto permite que o núcleo “faminto” pegue o que ele precisar. Por outro lado, a cache compartilhada também possibilita que um núcleo ganancioso prejudique os outros.

Uma área na qual CMPs diferem dos seus primos maiores é a tolerância a falhas. Pelo fato de as CPUs serem tão proximamente conectadas, falhas em componentes compartilhados podem derrubar múltiplas CPUs ao mesmo tempo, algo improvável em multiprocessadores tradicionais.

Além dos chips multinúcleo simétricos, onde todos os núcleos são idênticos, outra categoria comum de chip multinúcleo é o SoC (System on a Chip — sistema em um chip). Esses chips têm uma ou mais CPUs principais, mas também núcleos para fins especiais, como decodificadores de vídeo e áudio, criptoprocessadores, interfaces de rede e mais, levando a um sistema computacional completo em um chip.

Chips com muitos núcleos (manycore)
Multinúcleo significa simplesmente “mais de um núcleo”, mas quando o número de núcleos cresce bem além do alcance da contagem nos dedos, usamos outro nome. Chips com muitos núcleos (manycore) são multinúcleos que contêm dezenas, centenas, ou mesmo milhares de núcleos. Embora não exista um limiar claro, além do qual um multinúcleo torna-se um chip com muitos núcleos, uma distinção fácil que podemos fazer é que você provavelmente tem um chip com muitos núcleos quando não se importa de perder um ou dois

Outro problema com números realmente grandes de núcleos é que o equipamento necessário para manter suas caches coerentes torna-se muito complicado e muito caro. Muitos engenheiros preocupam-se com o fato de que a coerência de cache pode não ser escalável para muitas centenas de núcleos. Alguns chegam a defender que devemos abrir mão da ideia completamente. Eles temem que o custo dos protocolos de coerência no hardware seja tão alto que todos aqueles núcleos novos em folha não ajudarão muito o desempenho, pois o processador estará ocupado demais mantendo as caches em um estado consistente. Pior, ele teria de gastar memória demais no diretório (rápido) para fazê-lo. Essa situação é conhecida como parede de coerência.

Milhares de núcleos nem são mais tão especiais. Os chips com muitos núcleos mais comuns hoje, unidades de processamento gráfico, são encontrados em praticamente qualquer sistema computacional que não seja embarcado e tenha um monitor. Uma GPU (Graphic Processing Unit — unidade de processamento gráfico) é um processador com memória dedicada e, literalmente, milhares de pequenos núcleos. Comparado com processadores de propósito geral, GPUs gastam a maior parte de seu orçamento de transistores nos circuitos que realizam cálculos e menos em caches e lógica de controle. Elas são muito boas para diversas pequenas computações realizadas em paralelo, como renderizar polígonos em aplicações gráficas, mas não são tão boas em tarefas em série. Também são difíceis de programar. Embora GPUs possam ser úteis para sistemas operacionais (por exemplo, codificação ou processamento de tráfego de rede), não é provável que muito do próprio sistema operacional vá executar nas GPUs.

Uma diferença importante entre programar GPUs e processadores de propósito geral é que as GPUs são essencialmente máquinas de “dados múltiplos e instrução única”, o que significa que um grande número de núcleos executa exatamente a mesma instrução, mas em fragmentos diferentes de dados. Esse modelo de programação é ótimo para o paralelismo de dados, mas nem sempre conveniente para outros estilos de programação (como o paralelismo de tarefas).

Multinúcleos heterogêneos
Alguns chips integram uma GPU e uma série de núcleos de propósito geral na mesma pastilha. De modo similar, muitos SoCs contêm núcleos de propósito geral além de um ou mais processadores de propósitos especiais. Sistemas que integram múltiplos tipos diferentes de processadores em um único chip são conhecidos coletivamente como processadores multinúcleos heterogêneos. Um exemplo de um processador multinúcleo heterogêneo é a linha de processadores de rede IXP introduzida pela Intel em 2000 e atualizada regularmente com a última tecnologia de ponta. Os processadores de rede tipicamente contêm um único núcleo de controle de propósito geral (por exemplo, um

processador ARM executando Linux) e muitas dezenas de processadores de fluxo (stream processors) altamente especializados que são bons de verdade em processar pacotes de rede e não muito mais. Eles costumam ser usados em equipamentos de rede, como roteadores e firewalls.

O mesmo é verdade para o GPU e os núcleos de propósito geral. No entanto, também é possível introduzir a heterogeneidade enquanto se mantém o mesmo conjunto de instruções. Por exemplo, uma CPU pode ter um pequeno número de núcleos “grandes”, com pipelines profundas e possivelmente velocidades de relógio altas, e um número maior de núcleos “pequenos” que são mais simples, menos poderosos e talvez executem em frequências mais baixas. Os núcleos poderosos são necessários para executar códigos que exigem processamento sequencial rápido, enquanto os núcleos pequenos são úteis para tarefas que podem ser executadas com eficiência em paralelo.

Tipos de sistemas operacionais para multiprocessadores

O software de multiprocessadores, em particular, sistemas operacionais de multiprocessadores. Várias abordagens são possíveis. Observe que todas são igualmente aplicáveis a sistemas multinúcleos, assim como sistemas com CPUs discretas.

Cada CPU tem o seu próprio sistema operacional

A maneira mais simples possível de organizar um sistema operacional de multiprocessadores é dividir estaticamente a memória em um número de partições igual ao de CPUs, e dar a cada CPU sua própria memória privada e sua própria cópia privada do sistema operacional. Na realidade, as n CPUs então operam como n computadores independentes. Uma otimização óbvia é permitir que todas compartilhem o código do sistema operacional e façam cópias privadas apenas das estruturas de dados do sistema operacional.

Esse esquema é ainda melhor do que ter n computadores separados, já que ele permite que todas as máquinas compartilhem um conjunto de discos e outros dispositivos de E/S, e também permite que a memória seja compartilhada de maneira flexível. Por exemplo, mesmo com a alocação de memória estática, uma CPU pode receber uma porção extra grande da memória, então pode lidar com grandes programas de maneira eficiente. Além disso, processos podem comunicar-se com eficiência um com o outro ao permitir que um produtor escreva dados diretamente na memória e permitindo que um consumidor a busque do lugar em que o produtor a escreveu. Ainda assim, a partir da perspectiva do sistema operacional, cada CPU ter o seu próprio sistema operacional é algo bastante primitivo.

Vale a pena mencionar quatro aspectos desse projeto que talvez não sejam óbvios. Primeiro, quando um processo faz uma chamada de sistema, ela é capturada e tratada na sua própria CPU usando as estruturas de dados nas tabelas daquele sistema operacional.

Segundo, tendo em vista que cada sistema operacional tem as suas próprias tabelas, ele também tem seu próprio conjunto de processos que escalona para si mesmo. Não há compartilhamento de processos.

Terceiro, não há compartilhamento de páginas físicas. Pode acontecer de a CPU 1 ter páginas de sobra enquanto a CPU 2 está paginando continuamente.

Quarto, e pior, se o sistema operacional mantém uma cache de buffer de blocos de disco recentemente usados, cada sistema operacional faz isso independentemente dos outros. Desse modo, pode acontecer de um determinado bloco de disco estar presente e sujo em múltiplas caches de buffer ao mesmo tempo, levando a resultados inconsistentes. A única maneira de evitar esse problema é eliminar as caches de buffer. Fazê-lo não é difícil, mas prejudica consideravelmente o desempenho.

Por essas razões, esse modelo raramente é usado em sistemas de produção, embora ele tenha sido nos primeiros dias dos multiprocessadores, quando o objetivo era viabilizar o mais rápido possível os sistemas operacionais existentes para alguns multiprocessadores novos.

Multiprocessadores “mestre-escravo”

Neste segundo modelo, todas as chamadas de sistema são redirecionadas para a CPU 1 para serem processadas ali. A CPU 1 também pode executar processos do usuário se restar tempo da CPU. Esse modelo é chamado de mestre-escravo.

O modelo mestre-escravo soluciona a maioria dos problemas do primeiro modelo. Há uma única estrutura de dados (por exemplo, uma lista ou um conjunto de listas priorizadas) que mantém o controle dos processos prontos. Quando uma CPU fica ociosa, ela pede ao sistema operacional na CPU 1 um processo para executar e ele lhe aloca um. Desse modo, jamais pode acontecer de uma CPU estar ociosa enquanto outra está sobrecarregada. De modo similar, páginas podem ser alocadas entre todos os processos dinamicamente e há apenas uma cache de buffer, então inconsistências jamais ocorrem.

O problema com esse modelo é que com muitas CPUs, o mestre tornar-se-á um gargalo. Afinal de contas, ele tem de lidar com todas as chamadas de sistema de todas as CPUs. Se, digamos, 10% de todo o tempo for gasto lidando com chamadas do sistema, então 10 CPUs praticamente saturarão o mestre, e com 20 CPUs ele estará completamente sobrecarregado. Assim, esse modelo é simples e executável para pequenos multiprocessadores, mas para os grandes ele falha.

Multiprocessadores simétricos

Nosso terceiro modelo, o SMP (Symmetric MultiProcessor — multiprocessador simétrico), elimina essa assimetria. Há uma cópia do sistema operacional na memória, mas qualquer CPU pode executá-lo. Quando uma chamada de sistema é feita, a CPU na qual ela foi feita chaveia para o núcleo e processa a chamada.

Esse modelo equilibra processos e a memória dinamicamente, tendo em vista que há apenas um conjunto de tabelas do sistema operacional. Ele também elimina o gargalo da CPU mestre, já que não há mestre, mas ele introduz os seus próprios problemas. Em particular, se duas ou mais CPUs estão executando o código do sistema operacional ao

mesmo tempo, pode ocorrer um desastre. Imagine CPUs simultaneamente escolhendo o mesmo processo para executar ou reivindicando a mesma página de memória livre. A maneira mais simples em torno desses problemas é associar um mutex (isto é, variável de travamento) ao sistema operacional, tornando todo o sistema uma grande região crítica. Quando uma CPU quer executar um código de sistema operacional, ela tem de adquirir o mutex primeiro. Se o mutex estiver travado, ela simplesmente espera. Dessa maneira, qualquer CPU pode executar o sistema operacional, mas apenas uma de cada vez. Essa abordagem é algo chamado de grande trava de núcleo (big kernel lock). Esse modelo funciona, mas é quase tão ruim quanto o modelo mestre-escravo.

Divisão do sistema operacional em múltiplas regiões críticas independentes que não interagem umas com as outras. Cada região crítica é protegida por seu próprio mutex, então apenas uma CPU de cada vez pode executá-la. Dessa maneira, muito mais paralelismo pode ser conseguido. No entanto, pode muito bem acontecer de algumas tabelas, como a de processos, serem usadas por múltiplas regiões críticas. Por exemplo, a tabela de processos é necessária para o escalonamento, mas também para a chamada de sistema fork, assim como o tratamento de sinais. Cada tabela que pode ser usada por múltiplas regiões críticas precisa do seu próprio mutex. Dessa maneira, cada região crítica pode ser executada e cada tabela crítica pode ser acessada por apenas uma CPU de cada vez.

Sincronização de multiprocessadores

As CPUs em um multiprocessador frequentemente precisam ser sincronizadas.

Para começo de conversa, primitivas de sincronização adequadas são realmente necessárias. Se um processo em uma máquina uniprocessadora (apenas uma CPU) realiza uma chamada de sistema que exige acessar alguma tabela de núcleo crítica, o código de núcleo pode simplesmente desabilitar as interrupções antes de tocar na tabela. Ele pode então fazer seu trabalho sabendo que será capaz de terminar sem a intromissão de qualquer outro processo querendo tocar a tabela antes de estar terminada. Em um multiprocessador, desabilitar interrupções afeta apenas a CPU realizando a tarefa. Outras CPUs continuam a executar e ainda podem tocar a tabela crítica. Em consequência, um protocolo de mutex apropriado deve ser usado e respeitado por todas as CPUs para garantir que a exclusão mútua funcione.

O cerne de qualquer protocolo de mutex prático é uma instrução especial que permite que uma palavra de memória seja inspecionada e ajustada em uma operação indivisível. O que essa instrução faz é ler uma palavra de memória e armazená-la em um registrador. Em simultâneo, ela escreve um 1 (ou algum outro valor que não seja zero) na palavra de memória. É claro, são necessários dois ciclos de barramento para realizar a leitura e a escrita de memória. Em um uniprocessador, enquanto a instrução não puder ser interrompida no meio do caminho, o TSL sempre funciona como o esperado.

A instrução TSL tem de primeiro travar o barramento, evitando que outras CPUs o acessem, então realizar ambos os acessos de memória e em seguida destravar o barramento. Em geral, o travamento do barramento é feito com a solicitação do barramento usando o protocolo de solicitação de barramento usual, então sinalizando (isto é, ajustando para um

valor lógico 1) alguma linha de barramento especial até que ambos os ciclos tenham sido completados. Enquanto essa linha especial estiver sendo sinalizada, nenhuma outra CPU terá o direito de acesso ao barramento.

Se o TSL for corretamente implementado e usado, ele garante que a exclusão mútua pode funcionar. No entanto, esse método de exclusão mútua usa uma trava giratória, porque a CPU requisitante apenas permanece em um laço estreito testando a variável de travamento o mais rápido que ela pode. Não apenas ele desperdiça completamente o tempo da CPU requisitante (ou CPUs), como também coloca uma carga enorme sobre o barramento ou memória, desacelerando seriamente todas as outras CPUs que estão tentando realizar seu trabalho normal.

À primeira vista, pode parecer que a presença do armazenamento em cache deveria eliminar o problema da contenção de barramento, mas não elimina.

Uma ideia ainda melhor é dar a cada CPU que deseja adquirir o mutex sua própria variável de travamento. A variável deve residir em um bloco de cache não utilizado diferente para evitar conflitos. O algoritmo funciona fazendo que a CPU que falha em adquirir uma trava aloque uma variável de travamento e junte-se ao fim de uma lista de CPUs esperando pela trava. Quando a atual detentora da trava sair da região crítica, ela libera a variável privada que a primeira CPU na lista está testando (na sua própria cache). Essa CPU então adentra a região crítica. Quando ela finaliza, ela libera a variável que a sua sucessora está usando, e assim por diante. Embora o protocolo seja de certa maneira complicado (para evitar que duas CPUs se juntem ao fim da lista simultaneamente), ele é eficiente e livre de inanição.

Teste contínuo versus chaveamento

Até o momento, presumimos que uma CPU precisando de um mutex impedido apenas espera por ele, testando intermitentemente, ou ligando-se a uma lista de CPUs à espera. Às vezes, não há uma alternativa para a CPU requisitante que não seja esperar. Por exemplo, suponha que alguma CPU esteja ociosa e precise acessar a lista compartilhada de processos prontos para obter um processo para executar. Se a lista estiver bloqueada, a CPU não pode simplesmente decidir suspender o que ela está fazendo e executar outro processo, pois fazer isso exigiria ler a lista de processos prontos. Ela tem de esperar até poder adquirir a lista.

No entanto, em outros casos, há uma escolha. Por exemplo, se algum thread em uma CPU precisa acessar a cache de buffer do sistema de arquivos e ela está travada no momento, a CPU pode decidir chavear para um thread diferente em vez de esperar. Observe que essa questão não ocorre em um uniprocessador porque a espera não tem sentido quando não há outra CPU para liberar a variável de travamento. Se um thread tenta adquirir uma variável de travamento e falha, ele será sempre bloqueado para dar oportunidade ao proprietário da variável de ser executado e liberá-la.

Presumindo que o teste contínuo e o chaveamento de thread sejam ambas opções possíveis, a análise de custo-benefício entre os dois pode ser feita como a seguir. O teste contínuo desperdiça ciclos da CPU diretamente. Testar uma variável de travamento não é um trabalho produtivo. O chaveamento, no entanto, também desperdiça ciclos da CPU,

tendo em vista que o estado do thread atual deve ser salvo, a variável de travamento na lista de processos prontos deve ser adquirida, um thread deve ser escolhido, o seu estado deve ser carregado e ele deve ser inicializado. Além disso, a cache da CPU conterá todos os blocos errados, então muitas faltas de cache de alto custo ocorrerão à medida que o novo thread começar a executar. Faltas de TLB também são prováveis. Em consequência, deve ocorrer um chaveamento de volta para o thread original, seguido de mais faltas na cache. Os ciclos gastos para realizar esses dois chaveamentos de contexto mais todas as faltas na cache são desperdiçados.

Uma alternativa é sempre realizar o teste contínuo. Uma segunda alternativa é sempre chavear. Mas uma terceira alternativa é tomar uma decisão independente cada vez que um mutex travado for encontrado. No momento em que a decisão precisa ser tomada, não se sabe se é melhor testar ou chavear, mas para qualquer dado sistema, é possível anotar todas as atividades e analisá -las mais tarde offline. Então é possível dizer em retrospectiva qual decisão foi a melhor e quanto tempo foi desperdiçado no melhor caso. Esse algoritmo de “espelho retrovisor” torna-se assim uma referência contra a qual algoritmos exequíveis podem ser mensurados.

Escalonamento de multiprocessadores

Antes de examinarmos como o escalonamento é realizado em multiprocessadores, é necessário determinar o que está sendo escalonado.

É importante sabermos se os threads são threads de núcleo ou de usuário. Se a execução de threads for feita por uma biblioteca no espaço do usuário e o núcleo não souber de nada a respeito dos threads, então o escalonamento ocorre em uma base por processo como sempre ocorreu. Se o núcleo não faz nem ideia de que o thread existe, dificilmente ele poderá escaloná-los.

Com os threads de núcleo, o quadro é diferente. Aqui o núcleo está ciente de todos os threads e pode fazer a sua escolha entre os threads pertencentes a um processo. Nesses sistemas, a tendência é que o núcleo escolha um thread para executar, com o processo ao qual ele pertence tendo apenas um pequeno papel (ou talvez nenhum) no algoritmo de seleção do thread.

Processo versus thread não é a única questão de escalonamento. Em um uniprocessador, o escalonamento é unidimensional. A única questão que deve ser respondida (repetidamente) é: “Qual thread deve ser executado em seguida?”. Em um multiprocessador, o escalonamento tem duas dimensões. O escalonador tem de decidir em qual thread executar e em qual CPU. Essa dimensão extra complica muito o escalonamento em multiprocessadores.

Outro fator complicador é que em alguns sistemas, todos os threads são não relacionados, pertencendo a diferentes processos e não tendo nada a ver um com o outro. Em outros, eles vêm em grupos, todos pertencendo à mesma aplicação e trabalhando juntos. Um exemplo da primeira situação é um sistema de servidores no qual usuários independentes

iniciam processos independentes. Os threads de processos diferentes não são relacionados e cada um pode ser escalonado sem levar em consideração o outro.

Além disso, às vezes é útil escalonar threads que se comunicam extensivamente, digamos de uma maneira produtor-consumidor, não apenas ao mesmo tempo, mas também próximos no espaço. Por exemplo, eles podem beneficiar-se do compartilhamento de caches. Da mesma maneira, em arquiteturas NUMA, pode ajudar se eles acessarem a memória que está próxima.

Tempo compartilhado

Vamos primeiro abordar o caso do escalonamento de threads independentes; mais tarde, consideraremos como escalonar threads relacionados. O algoritmo de escalonamento mais simples para lidar com threads não relacionados é ter uma única estrutura de dados em todo sistema para os threads prontos, possivelmente apenas uma lista, mais provavelmente um conjunto de threads em diferentes prioridades.

Ter uma única estrutura de dados de escalonamento usada por todas as CPUs compartilha o tempo delas de maneira semelhante à sua disposição em um sistema uniprocessador. Também proporciona um balanceamento de carga automático, pois nunca pode acontecer de uma CPU estar ociosa enquanto as outras estão sobrecarregadas. Duas desvantagens dessa abordagem são a contenção potencial para a estrutura de dados de escalonamento à medida que o número de CPUs cresce e a sobrecarga usual na realização do chaveamento de contexto quando um thread bloqueia para E/S.

Também é possível que um chaveamento de contexto aconteça quando expirar o quantum de um processo. Em um multiprocessador, esse fato apresenta determinadas propriedades que não estão presentes em um uniprocessador. Suponha que um thread esteja mantendo uma trava giratória quando seu quantum expira. Outras CPUs esperando na trava giratória simplesmente desperdiçam seu tempo testando até que o thread seja escalonado de novo e libere a trava.

Para driblar essa anomalia, alguns sistemas usam o escalonamento inteligente, no qual um thread adquirindo uma trava giratória ajusta um sinalizador de processo (processwide flag) para mostrar que atualmente detém a trava giratória. Quando ele libera a trava, ele baixa o sinalizador. O escalonador não para, então, um thread que retém uma trava giratória, mas em vez disso, dá a ele um pouco mais de tempo para completar sua região crítica e liberar a trava.

Outra questão relevante no escalonamento é o fato de que enquanto todas as CPUs são iguais, algumas são mais iguais. Em particular, quando o thread A executou por um longo tempo na CPU k, a cache da CPU k estará cheia de blocos de A. Se A for logo executado de novo, ele pode ter um desempenho melhor do que se ele for executado na CPU k, pois a cache de k ainda pode conter alguns dos blocos de A. Ter blocos da cache pré-carregados aumentará a taxa de acerto da cache e, desse modo, a velocidade do thread. Além disso, a TLB também pode conter as páginas certas, reduzindo suas faltas.

Alguns multiprocessadores levam esse efeito em consideração e usam o que é chamado de escalonamento por afinidade. A ideia básica aqui é fazer um esforço sério para que um thread execute na mesma CPU que ele executou da última vez. Uma maneira de criar essa afinidade é usar um algoritmo de escalonamento de dois níveis. Quando um thread é criado, ele é designado para uma CPU, por exemplo, baseado em qual CPU tem a menor carga no momento. Essa alocação de threads para CPUs é o nível mais alto do algoritmo. Como resultado dessa política, cada CPU adquire a sua própria coleção de threads.

O escalonamento real dos threads é o nível mais baixo do algoritmo. Ele é feito por cada CPU separadamente, usando prioridades ou algum outro meio. Ao tentar manter um thread na mesma CPU por sua vida inteira, a afinidade de cache é maximizada. No entanto, se uma CPU não tem threads para executar, ela toma um de outra CPU em vez de ficar ociosa.

O escalonamento em dois níveis traz três benefícios. Primeiro, ele distribui a carga de maneira aproximadamente uniforme entre as CPUs disponíveis. Segundo, quando possível, é obtida uma vantagem por afinidade de cache. Terceiro, ao dar a cada CPU sua própria lista pronta, a contenção para as listas prontas é minimizada, pois tentativas de usar a lista pronta de outra CPU são relativamente raras.

Compartilhamento de espaço

A outra abordagem geral para o escalonamento de multiprocessadores pode ser usada quando threads são relacionados uns com os outros de alguma maneira. Também, muitas vezes ocorre que um único processo tem múltiplos threads que trabalham juntos. Por exemplo, se os threads de um processo se comunicam muito, é interessante tê-los executando ao mesmo tempo. O escalonamento de múltiplos threads ao mesmo tempo através de múltiplas CPUs é chamado de compartilhamento de espaço.

O algoritmo de compartilhamento de espaço mais simples funciona dessa maneira. Presuma que um grupo inteiro de threads relacionados é criado ao mesmo tempo. No momento em que ele é criado, o escalonador confere para ver se há tantas CPUs livres quanto há threads. Se existirem, cada thread recebe sua própria CPU dedicada (isto é, não multiprogramada) e todos são inicializados. Se não houver, nenhum dos threads pode ser inicializado até que haja um número suficiente de CPUs disponíveis. Cada thread detém sua CPU até que termine, momento em que ela é colocada de volta para o pool de CPUs disponíveis. Se um thread bloquear na E/S, ele continua a segurar a CPU, que está simplesmente ociosa até o thread despertar. Quando aparecer o próximo lote de threads, o mesmo algoritmo é aplicado.

Em qualquer instante no tempo, o conjunto de CPUs é dividido estaticamente em uma série de divisões, cada uma executando os threads de um processo. Com o passar do tempo, o número e tamanho das divisões mudam à medida que novos threads são criados e os antigos são concluídos e terminam.

Periodicamente, decisões de escalonamento precisam ser tomadas. Em sistemas de uniprocessadores, o trabalho mais curto primeiro é um algoritmo bem conhecido para o escalonamento de lote. O algoritmo análogo para um multiprocessador é escolher o

processo que estiver precisando do menor número de ciclos de CPUs, isto é, o thread cujo contador de CPU versus tempo de execução seja o menor entre os candidatos. No entanto, na prática, essa informação raramente encontra-se disponível, então é difícil levar o algoritmo adiante. Na realidade, estudos demonstraram que, na prática, é difícil superar o primeiro algoritmo a chegar, primeiro a ser servido.

Nesse modelo simples de divisão, uma thread somente pede algum número determinado de CPUs e as recebe todas, ou tem de esperar até que estejam disponíveis. Uma abordagem diferente é deixar que os threads gerenciem ativamente o grau de paralelismo. Um método para gerenciar o paralelismo é ter um servidor central que controle quais threads estão executando e querem executar e quais são as exigências de CPU mínima e máxima. Periodicamente, cada aplicação indaga o servidor central para saber quantas CPUs ela pode usar. A aplicação então ajusta o número de threads para mais ou para menos a fim de corresponder com o que há disponível.

Escalonamento em bando

Uma vantagem clara do compartilhamento de espaço é a eliminação da multiprogramação, que elimina a sobrecarga de chaveamento de contexto. No entanto, uma desvantagem igualmente clara é o tempo desperdiçado quando uma CPU bloqueia e não tem nada a fazer até encontrar-se pronta de novo. Em consequência, as pessoas procuraram por algoritmos que buscam escalonar tanto no tempo quanto no espaço juntos, especialmente para threads que criam múltiplos threads, e que em geral precisam comunicar-se uns com os outros.

O escalonamento em bando tem três partes:

1. Grupos de threads relacionados são escalonados como uma unidade, um bando.

2. Todos os membros do bando executam ao mesmo tempo em diferentes CPUs com tempo compartilhado.

3. Todos os membros do bando começam e terminam juntos suas faixas de tempo.

O truque que faz o escalonamento de bando funcionar é que todas as CPUs são escalonadas de maneira sincronizada. Isso significa que o tempo é dividido em quanta discretos.

No começo de cada quantum novo, todas as CPUs são escalonadas novamente, com um thread novo sendo iniciado em cada uma. No começo de cada quantum seguinte, outro evento de escalonamento acontece. Entre eles, não ocorre nenhum escalonamento. Se um thread bloqueia, a sua CPU permanece ociosa até o fim do quantum.

A ideia do escalonamento em bando é ter todos os threads de um processo executados juntos, ao mesmo tempo, em diferentes CPUs, de maneira que se um deles envia uma solicitação para outro, ele receberá a mensagem quase imediatamente e será capaz de responder da mesma forma.

Multicomputadores

Multiprocessadores são populares e atraentes porque eles oferecem um modelo de comunicação simples: todas as CPUs compartilham uma memória comum. Processos podem escrever mensagens para a memória que podem então ser lidas por outros processos. A sincronização pode ser feita usando mutexes, semáforos, monitores e outras técnicas bem estabelecidas. O único problema é que grandes multiprocessadores são difíceis de construir e, portanto, são caros. Então algo mais é necessário se formos aumentar para um grande número de CPUs.

Para contornar esses problemas, muita pesquisa foi feita sobre multicomputadores, que são CPUs estreitamente acopladas que não compartilham memória. Cada uma tem a sua própria memória. Esses sistemas também são conhecidos por uma série de outros nomes, incluindo aglomerados de computadores e COWS (Clusters Of Workstations — aglomerados de estações de trabalho). Serviços de computação na nuvem são sempre construídos em multicomputadores, pois eles precisam ser grandes.

Multicomputadores são fáceis de construir, pois o componente básico é apenas um PC "despido", sem teclado, mouse ou monitor, mas com uma placa de interface de rede de alto desempenho. É claro, o segredo para se atingir um alto desempenho é projetar a rede de interconexão e a placa de interface inteligentemente. Esse problema é completamente análogo a construir a memória compartilhada em um multiprocessador. No entanto, a meta é enviar mensagens em uma escala de tempo de microssegundos, em vez de acessar a memória em uma escala de tempo de nanossegundos, então é algo mais simples, barato e fácil de conseguir.

Hardware de multicomputadores

O nó básico de um multicomputador consiste em uma CPU, memória, uma interface de rede e às vezes um disco rígido. O nó pode ser empacotado em um gabinete padrão de PC, mas o monitor, teclado e mouse estão quase sempre ausentes. Às vezes essa configuração é chamada de estação de trabalho sem cabeça (headless workstation), pois não há um usuário com uma cabeça na frente dela.

Tecnologia de interconexão

Cada nó tem uma placa de interface de rede com um ou dois cabos (ou fibras) saindo dela. Esses cabos conectam-se a outros cabos ou a comutadores. Em um sistema pequeno, pode haver um comutador para o qual todos os nós estão conectados na topologia da estrela.

Como alternativa ao projeto de um comutador único, os nós podem formar um anel, com dois fios saindo da placa de interface da rede, um para o nó à esquerda e outro indo para o nó à direita.

A grade ou malha é um projeto bidimensional que foi usado em muitos sistemas comerciais. Ela é altamente regular e fácil de escalar para tamanhos grandes e tem um diâmetro, o

caminho mais longo entre quaisquer dois nós, que aumenta somente com a raiz quadrada do número de nós. Uma variante da grade é o toro duplo, que é uma grade com as margens conectadas. Não apenas ele é mais tolerante a falhas do que a grade, mas também o diâmetro é menor, pois os cantos opostos podem se comunicar agora em apenas dois passos.

O cubo da é uma topologia tridimensional regular. Ilustramos um cubo 2 × 2 × 2, mas no caso mais geral ele poderia ser um cubo k × k × k. Temos um cubo tetradimensional constituído de dois cubos tridimensionais com os nós correspondentes conectados. Poderíamos fazer um cubo de cinco dimensões. Um cubo n-dimensional formado dessa maneira é chamado de um hipercubo.

Muitos computadores paralelos usam uma topologia de hipercubo, pois o diâmetro cresce linearmente com a dimensionalidade. Colocada a questão em outras palavras, o diâmetro é o logaritmo na base 2 do número de nós.

Dois tipos de esquemas de comutação são usados em multicomputadores. No primeiro, cada mensagem é primeiro quebrada (seja pelo software do usuário, ou pela interface de rede) em um bloco de algum comprimento máximo chamado de pacote. O esquema de comutação, chamado de comutação de pacotes armazenar e encaminhar (store-and-forward packet switching), consiste no pacote sendo injetado no primeiro comutador pela placa de interface de rede do nó remetente. Os bits chegam um de cada vez, e quando o pacote inteiro chega a um buffer de entrada, ele é copiado para a linha levando ao próximo comutador ao longo do caminho. Quando chega ao comutador ligado ao nó de destino, o pacote é copiado para aquela placa de interface de rede daquele nó e eventualmente para sua RAM.

Embora a comutação de pacotes armazenar e encaminhar seja flexível e eficiente, ela tem o problema de aumentar a latência (atraso) através da rede de interconexão.

O outro esquema de comutação, comutação de circuito (circuit switching), consiste no primeiro comutador estabelecer um caminho através de todos os comutadores até o comutador-destino. Uma vez que o caminho tenha sido estabelecido, os bits são bombeados até o fim, da origem ao destino, sem parar e o mais rápido possível. Não há armazenamento em buffer intermediário nos comutadores intervenientes. A comutação de circuito exige uma fase de preparação, que leva algum tempo, mas é mais rápida, uma vez que a preparação tenha sido completa. Após o pacote ter sido enviado, o caminho precisa ser desfeito novamente. Uma variação da comutação de circuito, chamada roteamento buraco de minhoca (wormhole routing), divide cada pacote em subpacotes e permite que o primeiro subpacote comece a fluir mesmo antes de o caminho inteiro ter sido construído.

Interfaces de rede

Todos os nós em um multicomputador têm uma placa contendo a conexão do nó para a rede de interconexão que mantém o multicomputador unido. A maneira como essas placas são construídas e como elas se conectam à CPU principal e RAM tem implicações substanciais para o sistema operacional.

Em virtualmente todos os multicomputadores, a placa de interface contém uma RAM substancial para armazenar pacotes de entrada e saída. Em geral, um pacote de saída tem de ser copiado para a RAM da placa de interface antes que ele possa ser transmitido para o primeiro comutador. A razão para esse projeto é que muitas redes de interconexão são síncronas, então, assim que uma transmissão de pacote tenha começado, os bits devem continuar a fluir em uma taxa constante. Se o pacote está na RAM principal, esse fluxo contínuo saindo da rede não pode ser garantido devido a outro tráfego no barramento de memória. Usando uma RAM dedicada na placa de interface elimina esse problema.

O mesmo problema ocorre com os pacotes de entrada. Os bits chegam da rede a uma taxa constante e muitas vezes extremamente alta. Se a placa de interface de rede não puder armazená-los em tempo real à medida que eles chegam, dados serão perdidos. É mais seguro armazenar pacotes de entrada na RAM privada da placa de interface e então copiá-las para a RAM principal mais tarde.

A placa de interface pode ter um ou mais canais de DMA ou mesmo uma CPU completa (ou talvez mesmo CPUs completas) na placa. Os canais DMA podem copiar pacotes entre a placa de interface e a RAM principal a uma alta velocidade solicitando transferências de bloco no barramento do sistema, desse modo transferindo diversas palavras sem ter de solicitar o barramento separadamente para cada palavra. No entanto, é precisamente esse tipo de transferência de bloco, que interrompe o barramento de sistema para múltiplos ciclos de barramento, que torna a RAM da placa de interface necessária em primeiro lugar.

Muitas placas de interface têm uma CPU nelas, possivelmente em adição a um ou mais canais de DMA. Elas são chamadas de processadores de rede e estão se tornando cada dia mais poderosas. Esse projeto significa que a CPU principal pode descarregar algum trabalho para a placa da rede, como o tratamento de transmissão confiável (se o hardware subjacente puder perder pacotes), multicasting (enviar um pacote para mais de um destino), compactação/descompactação, criptografia/descriptografia, e cuidar da proteção em um sistema que tem múltiplos processos. No entanto, ter duas CPUs significa que elas precisam sincronizar-se para evitar condições de corrida, o que acrescenta uma sobrecarga extra e significa mais trabalho para o sistema operacional.

Copiar dados através de camadas é seguro, mas não necessariamente eficiente.

Software de comunicação de baixo nível

O inimigo da comunicação de alto desempenho em sistemas de multicomputadores é a cópia em excesso de pacotes. No melhor caso, haverá uma cópia da RAM para a placa de interface no nó fonte, uma cópia da placa de interface fonte para a placa de interface destinatária (se não ocorrer nenhum armazenamento e encaminhamento no caminho) e uma cópia dali para a RAM de destino, um total de três cópias. No entanto, em muitos sistemas é até pior. Em particular, se a placa de interface for mapeada no espaço de endereçamento virtual do núcleo e não no espaço de endereçamento virtual do usuário, um processo do usuário pode enviar um pacote somente emitindo uma chamada de sistema que é capturada para o núcleo. Os núcleos talvez tenham de copiar os pacotes para sua própria memória tanto na saída quanto na entrada, a fim de evitar, por exemplo, faltas de

páginas enquanto transmitem pela rede. Além disso, o núcleo receptor provavelmente não sabe onde colocar os pacotes que chegam até que ele tenha uma chance de examiná-los

Se cópias de e para a RAM são o gargalo, as cópias extras de e para o núcleo talvez dobrem o atraso de uma extremidade à outra e cortem a vazão pela metade. Para evitar esse impacto sobre o desempenho, muitos multicomputadores mapeiam a placa de interface diretamente no espaço do usuário para colocar pacotes na placa diretamente, sem o envolvimento do núcleo. Embora essa abordagem definitivamente ajude o desempenho, ela introduz dois problemas.

Primeiro, e se vários processos estão executando no nó e precisam de acesso de rede para enviar pacotes? Qual ficará com a placa de interface em seu espaço de endereçamento? Ter uma chamada de sistema para mapear a placa dentro e fora de um espaço de endereçamento é caro, mas se apenas um processo ficar com a placa, como os outros enviarão pacotes? E o que acontece se a placa for mapeada no espaço de endereçamento virtual de A e um pacote chegar para o processo B, especialmente se A e B tiverem proprietários diferentes e nenhum deles quiser fazer esforço para ajudar o outro?

Uma solução é mapear a placa de interface para todos os processos que precisam dela, mas então um mecanismo é necessário para evitar condições de corrida. Em um ambiente compartilhado com múltiplos usuários todos apressados para realizar o seu trabalho, um usuário pode simplesmente travar o mutex associado com a placa e nunca o liberar. A conclusão aqui é que mapear a placa da interface no espaço do usuário realmente funciona bem só quando há apenas um processo do usuário executando em cada nó, a não ser que precauções extras sejam tomadas.

O segundo problema é que o núcleo pode precisar realmente de acesso à própria rede de interconexão, por exemplo, a fim de acessar o sistema de arquivos em um nó remoto. Ter o núcleo compartilhando a placa de interface com qualquer usuário não é uma boa ideia. Suponha que enquanto a placa era mapeada no espaço do usuário, um pacote do núcleo chegasse. Ou ainda que o processo do usuário enviasse um pacote para uma máquina remota fingindo ser o núcleo. A conclusão é que o projeto mais simples é ter duas placas de interface de rede, uma mapeada no espaço do usuário para tráfego de aplicação e outra mapeada no espaço do núcleo pelo sistema operacional. Muitos multicomputadores fazem precisamente isso.

Por outro lado, interfaces de rede mais novas são frequentemente multifila, o que significa que elas têm mais de um buffer para dar suporte com eficiência a múltiplos usuários. Placas com afinidade de núcleo de processamento tem a sua própria lógica de espalhamento (hashing) para ajudar a direcionar cada pacote para um processo adequado. Como é mais rápido processar todos os segmentos no mesmo fluxo de TCP no mesmo processador (onde as caches estão quentes), a placa pode usar a lógica de espalhamento para organizar os campos de fluxo de TCP (endereços de IP e números de porta TCP) e adicionar todos os segmentos com o mesmo índice de espalhamento na mesma fila que é servida por um núcleo específico. Isso também é útil para a virtualização, à medida que ela nos permite dar a cada máquina virtual sua própria fila.

Comunicação entre o nó e a interface de rede

Outra questão é como copiar pacotes para a placa de interface. A maneira mais rápida é usar o chip de DMA na placa apenas para copiá-los da RAM. O problema com essa abordagem é que o DMA pode usar endereços físicos em vez de virtuais e executar independentemente da CPU, a não ser que uma MMU de E/S esteja presente. Para começo de conversa, embora um processo do usuário de certo saiba o endereço virtual de qualquer pacote que ele queira enviar, ele em geral não conhece o endereço físico. Fazer uma chamada de sistema para realizar todo o mapeamento virtual para físico é algo indesejável, pois o sentido de se colocar a placa de interface no espaço do usuário em primeiro lugar era evitar ter de fazer uma chamada de sistema para cada pacote a ser enviado.

Acesso direto à memória remota

Em alguns campos, altas latências de rede simplesmente não são aceitáveis. Por exemplo, para determinadas aplicações em computação de alto desempenho, o tempo de computação é fortemente dependente da latência de rede. De maneira semelhante, a negociação de alta frequência depende absolutamente de computadores desempenharem transações (compra e venda de ações) a velocidades altíssimas.

Nesses cenários, vale a pena reduzir a quantidade de cópias. Por essa razão, algumas interfaces de rede dão suporte ao RDMA (Remote Direct Memory Access — acesso direto à memória remota), uma técnica que permite que uma máquina desempenhe um acesso de memória direto de um computador para outro. O RDMA não envolve nenhum sistema operacional e os dados são buscados diretamente da — e escritos para a — memória de aplicação.

Software de comunicação no nível do usuário

Processos em CPUs diferentes em um multicomputador comunicam-se enviando mensagens uns para os outros. Na forma mais simples, essa troca de mensagens é exposta aos processos do usuário. Em outras palavras, o sistema operacional proporciona uma maneira de enviar e receber mensagens, e rotinas de biblioteca tornam essas chamadas subjacentes disponíveis para os processos do usuário. Em uma forma mais sofisticada, a troca de mensagens real é escondida dos usuários ao fazer com que a comunicação remota pareça uma chamada de rotina.

Envio e recepção

No mínimo dos mínimos, os serviços de comunicação fornecidos podem ser reduzidos a duas chamadas (de biblioteca), uma para enviar mensagens e outra para recebê-las. A chamada para enviar uma mensagem poderia ser send(dest, &mptr); e a chamada para receber uma mensagem poderia ser receive(addr, &mptr);

A primeira envia a mensagem apontada por mptr para um processo identificado por dest e faz o processo que a chamou ser bloqueado até que a mensagem tenha sido enviada. A

segunda faz o processo que a chamou ser bloqueado até a chegada da mensagem. Quando isso ocorre, a mensagem é copiada para o buffer apontado por mptr e o processo chamado é desbloqueado. O parâmetro addr especifica o endereço para o qual o receptor está à espera. São possíveis muitas variantes dessas duas rotinas e seus parâmetros.

Chamadas bloqueantes versus não bloqueantes

As chamadas descritas são chamadas bloqueantes (às vezes conhecidas por chamadas síncronas). Quando um processo chama send, ele especifica um destino e um buffer para enviar àquele destino. Enquanto a mensagem está sendo enviada, o processo emissor é bloqueado (isto é, suspenso). A instrução seguindo a chamada para send não é executada até a mensagem ter sido completamente enviada. De modo similar, uma chamada para receive não retorna o controle até que a mensagem tenha sido realmente recebida e colocada no buffer de mensagem apontado pelo parâmetro. O processo segue suspenso em receive até a mensagem chegar, mesmo que isso leve horas. Em alguns sistemas, o receptor pode especificar de quem espera receber, neste caso ele permanece bloqueado até chegar uma mensagem daquele emissor.

Uma alternativa às chamadas bloqueantes é usar as chamadas não bloqueantes (às vezes conhecidas por chamadas assíncronas). Se send for não bloqueante, ele retorna o controle para o processo chamado imediatamente antes de a mensagem ser enviada. A vantagem desse esquema é que o processo emissor pode continuar a calcular em paralelo com a transmissão da mensagem, em vez de a CPU ter de ficar ociosa (presumindo que nenhum outro processo seja executável). A escolha entre primitivas bloqueantes e não bloqueantes é em geral feita pelos projetistas do sistema (isto é, ou uma primitiva ou outra está disponível), embora em alguns sistemas ambas estão disponíveis e os usuários podem escolher a sua favorita.

No entanto, a vantagem de desempenho oferecida por primitivas não bloqueantes é superada por uma séria desvantagem: o emissor não pode modificar o buffer da mensagem até que ela tenha sido enviada. As consequências de o processo sobrescrever a mensagem durante a transmissão são terríveis demais para serem contempladas. Pior ainda, o processo emissor não faz ideia de quando a transmissão terminou, então ele nunca sabe quando é seguro reutilizar o buffer. Ele mal pode evitar tocá-lo para sempre.

Há três saídas possíveis, a primeira solução é fazer o núcleo copiar a mensagem para um buffer de núcleo interno e então permitir que o processo continue.

A segunda solução é interromper (sinalizar) o emissor quando a mensagem tiver sido completamente enviada para informá-lo de que o buffer está mais uma vez disponível.

A terceira solução é fazer o buffer copiar na escrita, isto é, marcá-lo como somente de leitura até que a mensagem tenha sido enviada. Se o buffer for reutilizado antes de a mensagem ter sido enviada, uma cópia é feita.

Desse modo, as escolhas do lado emissor são:

1. Envio bloqueante (CPU ociosa durante a transmissão da mensagem).

2. Envio não bloqueante com cópia (tempo da CPU desperdiçado para cópia extra).
3. Envio não bloqueante com interrupção (torna a programação difícil).
4. Cópia na escrita (uma cópia extra provavelmente será necessária).

Assim como send pode ser bloqueante ou não bloqueante, da mesma forma receive pode ser os dois. Uma chamada bloqueante apenas suspende o processo que a chamou até a mensagem ter chegado. Se múltiplos threads estiverem disponíveis, esta é uma abordagem simples. De modo alternativo, um receive não bloqueante apenas diz ao núcleo onde está o buffer e retorna o controle quase imediatamente. Uma interrupção pode ser usada para sinalizar que uma mensagem chegou. No entanto, interrupções são difíceis de programar e também bastante lentas, então talvez seja preferível para o receptor testar mensagens que estejam chegando usando uma rotina, poll, que diz se há alguma mensagem esperando. Em caso afirmativo, o processo chamado pode chamar get_message, que retorna a primeira mensagem que chegou.

Outra opção ainda é um esquema no qual a chegada de uma mensagem faz com que um thread novo seja criado espontaneamente no espaço de endereçamento do processo receptor. Esse thread é chamado de thread pop-up. Ele executa uma rotina especificada antes e cujo parâmetro é um ponteiro para a mensagem que chega. Após processar a mensagem, ele apenas termina e é automaticamente destruído.

Uma variante dessa ideia é executar o código receptor diretamente no tratador da interrupção, sem passar o trabalho de criar um thread pop-up. Para tornar esse esquema ainda mais rápido, a mensagem em si contém o endereço do tratador, então quando uma mensagem chega, o tratador pode ser chamado com poucas instruções. A grande vantagem aqui é que nenhuma cópia é necessária. O tratador pega a mensagem da placa de interface e a processa no mesmo instante. Esse esquema é chamado de mensagens ativas.

Chamada de rotina remota

Embora o modelo de troca de mensagens proporcione uma maneira conveniente de estruturar um sistema operacional de multicomputadores, ele sofre de uma falha incurável: o paradigma básico em torno do qual toda a economia é construída é entrada/saída. As rotinas send e receive estão fundamentalmente engajadas em realizar E/S, e muitas pessoas acreditam que E/S é o modelo de programação errado.

Em suma, o que Birrell e Nelson sugeriram foi permitir que os programas chamassem rotinas localizadas em outras CPUs. Quando um processo na máquina 1 chama uma rotina na máquina 2, o processo chamador na 1 é suspenso, e a execução do processo chamado ocorre na 2. Informações podem ser transportadas do chamador para o chamado nos parâmetros e pode voltar no resultado da rotina. Nenhuma troca de mensagens ou E/S é visível ao programador. Essa técnica é conhecida como RPC (Remote Procedure Call — chamada de rotina remota) e tornou-se a base de uma grande quantidade de softwares de multicomputadores. Tradicionalmente o procedimento chamador é conhecido como o cliente

e o procedimento chamado é conhecido como o servidor, e usaremos esses nomes aqui também.

A ideia por trás do RPC é fazer com que uma chamada de rotina remota pareça o mais próxima possível de uma chamada a uma rotina local. Na forma mais simples, para chamar um procedimento remoto, o programa cliente deve ser ligado a uma rotina de biblioteca pequena chamada stub do cliente que representa a rotina do servidor no espaço de endereçamento do cliente. De modo similar, o servidor é ligado a uma rotina chamada stub do servidor. Essas rotinas escondem o fato de que a chamada de rotina do cliente para o servidor não é local.

Os passos reais na realização de uma RPC são: o passo 1 é o cliente chamando o stub do cliente. Essa chamada é uma chamada de rotina local, com os parâmetros empurrados para a pilha como sempre. O passo 2 é o stub do cliente empacotando os parâmetros em uma mensagem e fazendo uma chamada de sistema para enviar a mensagem. O empacotamento dos parâmetros é chamado de marshalling (preparação). O passo 3 é o núcleo enviando a mensagem da máquina do cliente para a máquina do servidor. O passo 4 é o núcleo passando o pacote que chega para o stub do servidor (que em geral teria chamado receive antes). Por fim, o passo 5 é o stub do servidor chamando a rotina do servidor. A resposta segue o mesmo caminho na outra direção.

O item fundamental a ser observado aqui é que a rotina do cliente, escrita pelo usuário, apenas faz uma chamada de rotina normal (isto é, local) para o stub do cliente, que tem o mesmo nome que a rotina do servidor. Dado que a rotina do cliente e o stub do cliente estão no mesmo espaço de endereçamento, os parâmetros são passados da maneira usual. De modo similar, a rotina do servidor é chamada por uma rotina em seu espaço de endereçamento com os parâmetros que ela espera. Para a rotina do servidor, nada é incomum. Dessa maneira, em vez de realizar E/S usando send e receive, a comunicação remota é feita simulando uma chamada de rotina local.

Memória compartilhada distribuída

Embora a RPC tenha os seus atrativos, muitos programadores ainda preferem um modelo de memória compartilhada e gostariam de usá-lo, mesmo em um multicomputador. De maneira bastante surpreendente, é possível preservar a ilusão da memória compartilhada razoavelmente bem, mesmo que ela não exista de fato, usando uma técnica chamada DSM (Distributed Shared Memory — memória compartilhada distribuída). Com DSM, cada página é localizada em uma das memórias. Cada máquina tem a sua própria memória virtual e tabelas de página. Quando uma CPU realiza um LOAD ou STORE em uma página que ela não tem, ocorre uma captura para o sistema operacional. O sistema operacional então localiza a página e pede à CPU que a detém no momento para removê-la de seu mapeamento e enviá-la através da rede de interconexão. Quando ela chega, a página é mapeada e a instrução faltante é reiniciada. Na realidade, o sistema operacional está apenas satisfazendo faltas de páginas da RAM remota em vez do disco local. Para o usuário, é como se a máquina tivesse a memória compartilhada.

Em um sistema DSM, o espaço de endereçamento é dividido em páginas, com as páginas sendo disseminadas através de todos os nós no sistema. Quando uma CPU referencia um endereço que não é local, ocorre uma captura, e o software DSM busca a página contendo o endereço e reinicializa a instrução com a falta, que agora completa de maneira bem-sucedida.

Replicação

Uma melhoria para o sistema básico que pode melhorar o desempenho consideravelmente é replicar páginas que são somente de leitura, por exemplo, código do programa, constantes somente de leitura, ou outras estruturas de dados somente de leitura.

Outra possibilidade é replicar não apenas páginas somente de leitura, mas também todas as páginas. Enquanto as leituras estão sendo feitas, não há de fato diferença alguma entre replicar uma página somente de leitura e replicar uma página de leitura e escrita. No entanto, se uma página replicada for subitamente modificada, uma ação especial precisa ser tomada para evitar que ocorram múltiplas cópias inconsistentes.

Falso compartilhamento

Os sistemas DSM são similares a multiprocessadores em aspectos chave. Em ambos os sistemas, quando uma palavra de memória não local é referenciada, um bloco de memória contendo a palavra é buscado da sua localização atual e colocado na máquina fazendo a referência (memória principal ou cache, respectivamente). Uma questão de projeto importante é: qual o tamanho que deve ter esse bloco? Em sistemas DSM, a unidade tem de ser um múltiplo do tamanho da página (porque o MMU funciona com páginas), mas pode ser 1, 2, 4, ou mais páginas. Na realidade, realizar isso simula um tamanho de página maior.

Há vantagens e desvantagens para um tamanho de página maior para DSM. A maior vantagem é que como o tempo de inicialização para uma transferência de rede é substancial, não leva realmente muito mais tempo transferir 4.096 bytes do que 1.024 bytes. Ao transferir dados em grandes unidades, quando uma parte grande do espaço de endereçamento precisa ser movida, o número de transferências muitas vezes pode ser reduzido. Essa propriedade é especialmente importante porque muitos programas apresentam localidade de referência, significando que se um programa referenciou uma palavra em uma página, é provável que ele referencie outras palavras na mesma página em um futuro imediato.

Por outro lado, a rede estará presa mais tempo com uma transferência maior, bloqueando outras faltas causadas por outros processos. Também, uma página efetiva grande demais introduz um novo problema, chamado de falso compartilhamento.

Aqui temos uma página contendo duas variáveis compartilhadas não relacionadas, A e B. O processador 1 faz um uso pesado de A, lendo-o e escrevendo-o. De modo similar, o processo 2 usa B frequentemente. Nessas circunstâncias, a página contendo ambas as variáveis estará constantemente se deslocando para lá e para cá entre as duas máquinas.

O problema aqui é que, embora as variáveis não sejam relacionadas, elas aparecem por acidente na mesma página, então quando um processo utiliza uma delas, ele também recebe a outra. Quanto maior o tamanho da página efetiva, maior a frequência da ocorrência de falso compartilhamento e, de maneira inversa, quanto menor o tamanho da página efetiva, menor a frequência dessa ocorrência.

Obtendo consistência sequencial

Se as páginas que podem ser escritas não são replicadas, atingir a consistência não é problema. Há exatamente uma cópia de cada página que pode ser escrita, e ela é movida para lá e pra cá dinamicamente conforme a necessidade. Sabendo que nem sempre é possível ver antecipadamente quais páginas podem ser escritas, em muitos sistemas DSM, quando um processo tenta ler uma página remota, uma cópia local é feita e tanto a cópia local quanto a remota são configuradas em suas respectivas MMUs como somente de leitura. Enquanto as referências forem de leitura, está tudo bem.

No entanto, se qualquer processo tentar escrever em uma página replicada, surgirá um problema de consistência potencial, pois mudar uma cópia e deixar as outras sozinhas é algo inaceitável. Antes que uma página compartilhada possa ser escrita, uma mensagem é enviada para todas as outras CPUs que detêm uma cópia da página solicitando a elas para removerem o mapeamento e descartarem a página. Após todas elas terem respondido que o mapeamento foi removido, a CPU original pode então realizar a escrita.

Também é possível tolerar múltiplas páginas que podem ser escritas em circunstâncias cuidadosamente restritas. Uma maneira é permitir que um processo adquira uma variável de travamento em uma porção do espaço de endereçamento virtual, e então desempenhar múltiplas operações de leitura e escrita na memória travada. No momento em que a variável de travamento for liberada, mudanças podem ser propagadas para outras cópias. Desde que somente uma CPU possa travar uma página em um dado momento, esse esquema preserva a consistência.

Escalonamento em multicomputadores

Em um multicomputador, a situação é bastante diferente. Cada nó tem a sua própria memória e o seu próprio conjunto de processos. A CPU 1 não pode subitamente decidir executar um processo localizado no nó 4 sem primeiro trabalhar bastante para consegui-lo. Essa diferença significa que o escalonamento em multicomputadores é mais fácil, mas a alocação de processos para os nós é mais importante.

O escalonamento em multicomputador é de certa maneira similar ao escalonamento em multiprocessador, mas nem todos os algoritmos do primeiro aplicam-se ao segundo. O algoritmo de multiprocessador mais simples — manter uma única lista central de processos prontos — não funciona, no entanto, dado que cada processo só pode executar na CPU em que ele está localizado no momento. No entanto, quando um novo processo é criado, uma escolha pode ser feita, por exemplo, onde colocá-lo para balancear a carga.

Já que cada nó tem os seus próprios processos, qualquer algoritmo de escalonamento pode ser usado. No entanto, também é possível usar o escalonamento em bando de multiprocessadores, já que isso exige meramente um acordo inicial sobre qual processo executar em qual intervalo de tempo, e alguma maneira de coordenar o início dos intervalos de tempo.

Balanceamento de carga

Há relativamente pouco a ser dito a respeito do escalonamento de multicomputadores, pois uma vez que um processo tenha sido alocado para um nó, qualquer algoritmo de escalonamento local dará conta do recado, a não ser que o escalonamento em bando esteja sendo usado. No entanto, justamente porque há tão pouco controle sobre um processo uma vez que ele tenha sido alocado para um nó, a decisão sobre qual processo deve ir com qual nó é importante. Isso contrasta com sistemas de multiprocessadores, nos quais todos os processos vivem na mesma memória e podem ser escalonados em qualquer CPU de acordo com sua vontade. Em consequência, vale a pena observar como os processos podem ser alocados para os nós de uma maneira eficiente. Os algoritmos e as heurísticas para fazer isso são conhecidos como algoritmos de alocação de processador.

Um algoritmo determinístico teórico de grafos

Uma classe de algoritmos amplamente estudada é para sistemas consistindo de processos com exigências conhecidas de CPU e memória, e uma matriz conhecida dando a quantidade média de tráfego entre cada par de processos. Se o número de processos for maior do que o número de CPUs, k, vários processos terão de ser alocados para cada CPU. A ideia é executar essa alocação a fim de minimizar o tráfego de rede.

O sistema pode ser representado como um grafo ponderado, com cada vértice sendo um processo e cada arco representando o fluxo de mensagens entre dois processos. Matematicamente, o problema então se reduz a encontrar uma maneira de dividir (isto é, cortar) o gráfico em k subgrafos disjuntos, sujeitos a determinadas restrições (por exemplo, exigências de memória e CPU totais inferiores a determinados limites para cada subgrafo). A meta é então encontrar a divisão que minimize o tráfego de rede enquanto atende todas as restrições.

Intuitivamente, o que estamos fazendo é procurar por aglomerados fortemente acoplados (alto fluxo de tráfego intragrupo), mas que interajam pouco com outros aglomerados (baixo fluxo de tráfego intergrupo).

Um algoritmo heurístico distribuído iniciado pelo emissor

Um algoritmo diz que quando um processo é criado, ele executa no nó que o criou, a não ser que o nó esteja sobrecarregado. A métrica usada para comprovar a sobrecarga pode envolver um número de processos grande demais, um conjunto de trabalho grande demais, ou alguma outra. Se ele estiver sobrecarregado, o nó seleciona outro nó ao acaso e pergunta a ele qual é a sua carga (usando a mesma métrica). Se a carga do nó sondado

estiver abaixo do valor limite, o novo processo é enviado para lá (EAGER et al, 1986). Se não estiver, outra máquina é escolhida para a sondagem. A sondagem não segue para sempre.

Se nenhum anfitrião adequado for encontrado dentro de N sondagens, o algoritmo termina e o processo executa na máquina de origem. A ideia é para os nós pesadamente carregados tentar se livrar do trabalho em excesso.

Algoritmo heurístico distribuído iniciado pelo receptor

Um algoritmo complementar ao discutido anteriormente, que é iniciado por um emissor sobrecarregado, é iniciado por um receptor com pouca carga. Com esse algoritmo, sempre que um processo termina, o sistema confere para ver se ele tem trabalho suficiente. Se não tiver, ele escolhe alguma máquina ao acaso e solicita trabalho a ela. Se essa máquina não tem nada a oferecer, uma segunda, e então uma terceira máquina são solicitadas. Se nenhum trabalho for encontrado com N sondagens, o nó para temporariamente de pedir, realiza qualquer trabalho que ele tenha em fila e tenta novamente quando o próximo processo terminar. Se nenhum trabalho estiver disponível, a máquina fica ociosa. Após algum intervalo de tempo fixo, ela começa a sondar novamente.

Uma vantagem desse algoritmo é que ele não coloca uma carga extra sobre o sistema em momentos críticos. O algoritmo iniciado pelo emissor faz um grande número de sondagens precisamente quando o sistema menos pode tolerá-las — isto é, quando ele está pesadamente carregado. Com o algoritmo iniciado pelo receptor, quando o sistema estiver pesadamente carregado, a chance de uma máquina ter trabalho insuficiente é pequena. No entanto, quando isso acontecer, será fácil encontrar trabalho para fazer. É claro, quando há pouco trabalho a fazer, o algoritmo iniciado pelo receptor cria um tráfego de sondagem considerável à medida que todas as máquinas sem trabalho caçam desesperadamente por trabalho para fazer. No entanto, é muito melhor ter uma sobrecarga extra quando o sistema não está sobrecarregado do que quando ele está.

Também é possível combinar ambos os algoritmos e fazer as máquinas tentarem se livrar do trabalho quando elas têm demais, e tentarem adquirir trabalho quando elas não têm o suficiente. Além disso, máquinas podem às vezes fazer melhor do que sondagens aleatórias mantendo um histórico de sondagens passadas para determinar se alguma máquina vive cronicamente subcarregada ou sobrecarregada. Uma dessas pode ser tentada primeiro, dependendo de o iniciador estar tentando livrar-se do trabalho ou adquiri-lo.

Sistemas distribuídos

Esses sistemas são similares a multicomputadores pelo fato de que cada nó tem sua própria memória privada, sem uma memória física compartilhada no sistema. No entanto, sistemas distribuídos são ainda mais fracamente acoplados do que multicomputadores.

Para começo de conversa, cada nó de um multicomputador geralmente tem uma CPU, RAM, uma interface de rede e possivelmente um disco para paginação. Em comparação, cada nó em um sistema distribuído é um computador completo, com um complemento completo de periféricos. Em seguida, os nós de um multicomputador estão em geral em uma única sala, de maneira que eles podem se comunicar através de uma rede de alta velocidade dedicada, enquanto os nós de um sistema distribuído podem estar espalhados pelo mundo todo. Finalmente, todos os nós de um multicomputador executam o mesmo sistema operacional, compartilhando um único sistema de arquivos, e estão sob uma administração comum, enquanto os nós de um sistema distribuído podem cada um executar um sistema operacional diferente, cada um dos quais tendo seu próprio sistema de arquivos, e estar sob uma administração diferente. Um exemplo típico de um multicomputador são 1.024 nós em uma única sala em uma empresa ou universidade trabalhando com, digamos, modelos farmacêuticos, enquanto um sistema distribuído típico consiste em milhares de máquinas cooperando de maneira desagregada através da internet.

Usando essas métricas, multicomputadores estão claramente no meio. Uma questão interessante é: “multicomputadores são mais parecidos com multiprocessadores ou com sistemas distribuídos?”. Estranhamente, a resposta depende muito de sua perspectiva. Do ponto de vista técnico, multiprocessadores têm memória compartilhada e os outros dois não. Essa diferença leva a diferentes modelos de programação e diferentes maneiras de ver as coisas. No entanto, do ponto de vista das aplicações, multiprocessadores e multicomputadores são apenas grandes estantes com equipamentos em uma sala de máquinas. Ambos são usados para solucionar problemas computacionalmente intensivos, enquanto um sistema distribuído conectando computadores por toda a internet em geral está muito mais envolvido na comunicação do que na computação e é usado de maneira diferente.

Até certo ponto, o acoplamento fraco dos computadores em um sistema distribuído é ao mesmo tempo uma vantagem e uma desvantagem. É uma vantagem porque os computadores podem ser usados para uma ampla variedade de aplicações, mas também é uma desvantagem, porque a programação dessas aplicações é difícil por causa da falta de qualquer modelo subjacente comum.

O que os sistemas distribuídos acrescentam à rede subjacente é algum paradigma comum (modelo) que proporciona uma maneira uniforme de ver o sistema como um todo. A intenção do sistema distribuído é transformar um monte de máquinas conectadas de maneira desagregada em um sistema coerente baseado em um conceito. Às vezes o paradigma é simples e às vezes ele é mais elaborado, mas a ideia é sempre fornecer algo que unifique o sistema.

Um exemplo simples de um paradigma unificador em um contexto diferente é encontrado em UNIX, onde todos os dispositivos de E/S são feitos para parecerem arquivos. Ter teclados, impressoras e linhas seriais, todos operando da mesma maneira, com as mesmas primitivas, torna mais fácil lidar com eles do que tê-los todos conceitualmente diferentes.

Um método pelo qual um sistema distribuído pode alcançar alguma medida de uniformidade diante diferentes sistemas operacionais e hardware subjacente é ter uma camada de

software sobre o sistema operacional. A camada, chamada de middleware.Essa camada fornece determinadas estruturas de dados e operações que permitem que os processos e usuários em máquinas distantes operem entre si de uma maneira consistente.

De certa maneira, middleware é como o sistema operacional de um sistema distribuído. Essa é a razão de ele estar sendo discutido em um livro sobre sistemas operacionais. Por outro lado, ele não é realmente um sistema operacional, então a discussão não entrará muito em detalhes.

Hardware de rede

Sistemas distribuídos são construídos sobre redes de computadores. As redes existem em duas variedades principais, LANs (Local Area Networks — redes locais), que cobrem um prédio ou um campus e WANs (Wide Area Networks — redes de longa distância), que podem cobrir uma cidade inteira, um país inteiro, ou todo o mundo. O tipo mais importante de LAN é a Ethernet, então a examinaremos como um exemplo de LAN. Como nosso exemplo de WAN, examinaremos a internet.

Ethernet
A Ethernet clássica, que é descrita no padrão IEEE Standard 802.3, consiste em um cabo coaxial ao qual uma série de computadores está ligada.

Na primeiríssima versão da Ethernet, um computador era ligado ao cabo literalmente abrindo um buraco no meio do cabo e enfiando um fio que levava ao computador. Esse conector era chamado de conector vampiro.

Com muitos computadores conectados ao mesmo cabo, um protocolo é necessário para evitar o caos. Para enviar um pacote na Ethernet, um computador primeiro escuta o cabo para ver se algum outro computador está transmitindo no momento. Se não estiver, ele começa a transmitir um pacote.

Se dois computadores começam a transmitir simultaneamente, resulta em uma colisão, que ambos detectam. Ambos respondem terminando suas transmissões, esperando por um tempo aleatório entre 0 e T μs e então iniciando de novo. Se outra colisão ocorrer, todos os computadores esperam um tempo aleatório no intervalo de 0 a 2T μs, e então tentam novamente. Em cada colisão seguinte, o intervalo de espera máximo é dobrado, reduzindo a chance de mais colisões. Esse algoritmo é conhecido como recuo exponencial binário.

Uma rede Ethernet tem um comprimento de cabo máximo e também um número máximo de computadores que podem ser conectados a ela. Para superar qualquer um desses limites, um prédio grande ou campus podem ser ligados a várias Ethernets, que são então conectadas por dispositivos chamados de pontes. Uma ponte é um dispositivo que permite o tráfego passar de uma Ethernet para outra quando a fonte está de um lado e o destino do outro.

Para evitar o problema de colisões, Ethernets modernas usam comutadores. Cada comutador tem algum número de portas, às quais podem ser ligados computadores, uma

Ethernet, ou outro comutador. Quando um pacote evita com sucesso todas as colisões e chega ao comutador, ele é armazenado em um buffer e enviado pela porta que acessa a máquina destinatária. Ao dar a cada computador a sua própria porta, todas as colisões podem ser eliminadas, ao custo de comutadores maiores.

A internet

A internet evoluiu da ARPANET, uma rede experimental de comutação de pacotes fundada pela Agência de Projetos de Pesquisa Avançados do Departamento de Defesa dos Estados Unidos. Ela foi colocada em funcionamento em dezembro de 1969 com três computadores na Califórnia e um em Utah. Ela foi projetada no auge da Guerra Fria para ser uma rede altamente tolerante a falhas que continuaria a transmitir tráfego militar mesmo no evento de ataques nucleares diretos sobre múltiplas partes da rede ao automaticamente desviar o tráfego das máquinas atingidas.

A ARPANET cresceu rapidamente na década de 1970, por fim compreendendo centenas de computadores. Então uma rede de pacotes via rádio, uma rede de satélites e por fim milhares de Ethernets foram ligadas a ela, levando à federação de redes que conhecemos hoje como internet.

A internet consiste em dois tipos de computadores, hospedeiros (hosts) e roteadores. Hospedeiros são PCs, notebooks, smartphones, servidores, computadores de grande porte e outros computadores de propriedade de indivíduos ou companhias que querem conectar-se à internet. Roteadores são computadores de comutação especializados que aceitam pacotes que chegam de uma das muitas linhas de entrada e os enviam para fora ao longo das muitas linhas de saída.

No topo temos um dos backbones (“espinha dorsal”), normalmente operado por um operador de backbone. Ele consiste em uma série de roteadores conectados por fibras óticas de alta largura de banda, com backbones operados por outras companhias telefônicas (competidoras). Em geral, nenhum hospedeiro se conecta diretamente ao backbone, a não ser máquinas de testes e manutenção operadas pela companhia telefônica.

Ligados aos roteadores dos backbones por conexões de fibra ótica de velocidade média, estão as redes regionais e roteadores nos ISPs. As Ethernets corporativas, por sua vez, cada uma tem um roteador nela e esses estão conectados a roteadores de rede regionais. Roteadores em ISPs estão conectados a bancos modernos usados pelos clientes dos ISPs. Dessa maneira, cada hospedeiro na internet tem pelo menos um caminho, e muitas vezes muitos caminhos, para cada outro hospedeiro.

Todo tráfego na internet é enviado na forma de pacotes. Cada pacote carrega dentro de si seu endereço de destino, e esse endereço é usado para o roteamento. Quando um pacote chega a um roteador, este extrai o endereço de destino e o compara (parte dele) em uma tabela para encontrar para qual linha de saída enviar o pacote e assim para qual roteador.

Serviços de rede e protocolos

Todas as redes de computador fornecem determinados serviços para os seus usuários (hospedeiros e processos), que elas implementam usando determinadas regras a respeito de trocas de mensagens legais.

Serviços de rede

Redes de computadores fornecem serviços aos hospedeiros e processos utilizando-as. O serviço orientado à conexão utilizou o sistema telefônico como modelo. Para falar com alguém, você pega o telefone, tecla o número, fala e desliga. De modo similar, para usar um serviço de rede orientado à conexão, o usuário do serviço primeiro estabelece uma conexão, usa a conexão e então a libera. O aspecto essencial de uma conexão é que ela atua como uma tubulação: o emissor empurra objetos (bits) em uma extremidade, e o receptor os coleta na mesma ordem na outra extremidade.

Em comparação, o serviço sem conexão utilizou o sistema postal como modelo. Cada mensagem (carta) carrega o endereço de destino completo, e cada uma é roteada através do sistema independente de todas as outras. Em geral, quando duas mensagens são enviadas para o mesmo destino, a primeira enviada será a primeira a chegar. No entanto, é possível que a primeira enviada possa ser atrasada de maneira que a segunda chegue primeiro. Com um serviço orientado à conexão isso é impossível.

Cada serviço pode ser caracterizado por uma qualidade de serviço. Alguns serviços são confiáveis no sentido de que eles jamais perdem dados. Normalmente, um serviço confiável é implementado fazendo que o receptor confirme o recebimento de cada mensagem enviando de volta um pacote de confirmação para que o emissor tenha certeza de que ela chegou.

Uma situação típica na qual um serviço orientado à conexão confiável é apropriado é a transferência de arquivos.

Para algumas aplicações, os atrasos introduzidos pelas confirmações são inaceitáveis. Uma dessas aplicações é o tráfego de voz digitalizado. É preferível para os usuários de telefone ouvir um pouco de ruído na linha ou uma palavra embaralhada de vez em quando do que introduzir um atraso para esperar por confirmações.

Nem todas as aplicações exigem conexões. Por exemplo, para testar a rede, tudo o que é necessário é uma maneira de enviar um pacote que tenha uma alta probabilidade de chegada, mas nenhuma garantia. Um serviço não confiável (isto é, sem confirmação) sem conexão é muitas vezes chamado de um serviço de datagrama.

Em outras situações, a conveniência de não ter de estabelecer uma conexão para enviar uma mensagem curta é desejada, mas a confiabilidade é essencial. O serviço de datagrama com confirmação pode ser fornecido para essas aplicações. Ele funciona como enviar uma carta registrada e solicitar um recibo de retorno.

Ainda outro serviço é o serviço de solicitação-réplica. Nele o emissor transmite um único datagrama contendo uma solicitação, e a réplica contém a resposta. O cliente emite uma solicitação e o servidor responde a ela.

Protocolos de rede

Todas as redes têm regras altamente especializadas para quais mensagens podem ser enviadas e respostas podem ser retornadas em resposta a essas mensagens. O conjunto de regras pelo qual computadores particulares se comunicam é chamado de protocolo.

Todas as redes modernas usam o que é chamado de pilha de protocolos para empilhar diferentes protocolos no topo um do outro.

Tendo em vista que a maioria dos sistemas distribuídos usa a internet como base, os protocolos-chave que esses sistemas usam são os dois principais da internet: IP e TCP. IP (Internet Protocol — protocolo da internet) é um protocolo de datagrama no qual um emissor injeta um datagrama de até 64 KB na rede e espera que ele chegue. Nenhuma garantia é dada.

Duas versões de IP estão em uso, v4 e v6. Quando um pacote chega a um roteador, este extrai o endereço de destino IP e o usa para o roteamento.

Já que os datagramas de IP não recebem confirmações, o IP sozinho não é suficiente para uma comunicação confiável na internet. Para fornecer uma comunicação confiável, outro protocolo, TCP (Transmission Control Protocol — protocolo de controle de transmissão), geralmente é colocado sobre o IP. O TCP emprega o IP para fornecer fluxos orientados à conexão. Para usar o TCP, um processo primeiro estabelece uma conexão a um processo remoto. O processo que está sendo requisitado é especificado pelo endereço de IP de uma máquina e um número de porta naquela máquina, a quem os processos interessados em receber conexões ouvem. Uma vez que isso tenha sido feito, ele simplesmente envia bytes para a conexão, com garantia de que sairão do outro lado ilesos e na ordem correta. A implementação do TCP consegue essa garantia usando números de sequências, somas de verificação e retransmissões de pacotes incorretamente recebidos. Tudo isso é transparente para os processos enviando e recebendo dados. Eles simplesmente veem uma comunicação entre processos como confiável, como um pipe do UNIX.

Para estabelecer uma conexão com um hospedeiro remoto (ou mesmo para enviar um datagrama), é necessário saber o seu endereço IP. Visto que gerenciar listas de endereços IP de 32 bits é inconveniente para as pessoas, um esquema chamado DNS (Domain Name System — serviço de nomes de domínio) foi inventado como um banco de dados que mapeia nomes de hospedeiros em ASCII em seus endereços IP. Desse modo é possível usar o nome DNS star.cs.vu.nl em vez do endereço IP correspondente 130.37.24.6.

Middleware baseado em documentos

O paradigma original por trás da web era bastante simples: cada computador pode deter um ou mais documentos, chamados de páginas da web. Cada página da web consiste em

texto, imagens, ícones, sons, filmes e assim por diante, assim como hyperlinks (ponteiros) para outras páginas. Quando um usuário solicita uma página da web usando um programa chamado de navegador da web, a página é exibida na tela. O clique em um link faz que a página atual seja substituída na tela pela página apontada.

Cada página da web tem um endereço único, chamado de URL. O protocolo é geralmente o http. Depois, vem o nome no DNS do hospedeiro contendo o arquivo. Por fim, há um nome de arquivo local dizendo qual arquivo é necessário. Desse modo, um URL especifica apenas um único arquivo no mundo todo.

A maneira como o sistema todo se mantém unido funciona da seguinte forma: a web é em essência um sistema cliente-servidor, com o usuário como o cliente e o site da web como o servidor. Quando o usuário fornece o navegador com um URL, seja digitando-o ou clicando em um hyperlink na página atual, o navegador dá determinados passos para buscar a página solicitada.

Middleware baseado no sistema de arquivos

A ideia básica por trás da web é fazer com que um sistema distribuído pareça uma coleção gigante de documentos interligados por hyperlinks. Uma segunda abordagem é fazer um sistema distribuído parecer um enorme sistema de arquivos.

Usar um modelo de sistema de arquivos para um sistema distribuído significa que há um único sistema de arquivos global, com usuários mundo afora capazes de ler e escrever arquivos para os quais eles têm autorização. A comunicação é conseguida quando um processo escreve dados em um arquivo e outros o leem de volta. Muitas das questões dos sistemas de arquivos padrão surgem aqui, mas também algumas novas relacionadas à distribuição.

Modelo de transferência

A primeira questão é a escolha entre o modelo upload/download e o modelo de acesso remoto. No primeiro, um processo acessa um arquivo primeiramente copiando-o do servidor remoto onde ele vive. Se o arquivo é somente de leitura, ele é então lido localmente, em busca de alto desempenho. Se o arquivo deve ser escrito, é escrito localmente. Quando o processo termina de usá-lo, o arquivo atualizado é colocado de volta no servidor. Com o modelo de acesso remoto, o arquivo fica no servidor e o cliente envia comandos para que o trabalho seja feito no servidor.

As vantagens do modelo upload/download são a sua simplicidade, e o fato de que transferir arquivos inteiros ao mesmo tempo é mais eficiente do que transferi-los em pequenas partes. As desvantagens são que deve haver espaço suficiente para o armazenamento do arquivo inteiro localmente, mover o arquivo inteiro é um desperdício se apenas partes dele forem necessárias e surgirão problemas de consistência se houver múltiplos usuários concorrentes.

A hierarquia de diretórios

Os arquivos são apenas parte da história. A outra parte é o sistema de diretórios. Todos os sistemas de arquivos distribuídos suportam diretórios contendo múltiplos arquivos. A próxima questão do projeto é se todos os clientes têm a mesma visão da hierarquia de diretório.

Uma questão relacionada de perto diz respeito a haver ou não um diretório raiz global, que todas as máquinas reconhecem como a raiz. Uma maneira de se ter um diretório raiz global é fazer com que a raiz contenha uma entrada para cada servidor e nada mais. Nessas circunstâncias, os caminhos assumem a forma /server/path, o que tem suas próprias desvantagens, mas pelo menos é a mesma em toda parte no sistema.

Transparência de nomeação

O principal problema com essa forma de nomeação é que ela não é totalmente transparente. Duas formas de transparência são relevantes nesse contexto e valem a pena ser distinguidas. A primeira, transparência de localização, significa que o nome do caminho não dá dica alguma para onde o arquivo está localizado. Um caminho como /server1/dir1/dir2/x diz a todos que x está localizado no servidor 1, mas não diz onde esse servidor está localizado. O servidor é livre para se mover para qualquer parte que ele quiser sem que o nome do caminho precise ser modificado. Portanto, esse sistema tem transparência de localização.

No entanto, suponha que o arquivo x seja extremamente grande e o espaço restrito no servidor 1. Além disso, suponha que exista espaço suficiente no servidor 2. O sistema pode muito bem mover x para o servidor 2 automaticamente. Infelizmente, quando o primeiro componente de todos os nomes de caminho está no servidor, o sistema não pode mover o arquivo para o outro servidor automaticamente, mesmo que dir1 e dir2 existam em ambos servidores. O problema é que mover o arquivo automaticamente muda o nome de caminho de /server1/dir1/dir2/x para /server2/dir1/dir2/x. Programas que têm a primeira cadeia de caracteres inserida cessarão de trabalhar se o caminho mudar. Um sistema no qual arquivos podem ser movidos sem que seus nomes sejam modificados possuem independência de localização. Um sistema distribuído que especifica os nomes de máquinas ou servidores em nomes de caminhos claramente não é independente de localização.

Existem três abordagens comuns para a nomeação de arquivos e diretórios em um sistema distribuído:

1. Nomeação de máquina + caminho, como /maquina/caminho ou maquina:caminho.

2. Montagem de sistemas de arquivos remotos na hierarquia de arquivos local.

3. Um único espaço de nome que parece o mesmo em todas as máquinas.

Semântica do compartilhamento de arquivos

Quando dois ou mais usuários compartilham o mesmo arquivo, é necessário definir a semântica da leitura e escrita precisamente para evitar problemas. Em sistemas de um único processador, a semântica normalmente afirma que, quando uma chamada de sistema read segue uma chamada de sistema write, a read retorna o valor recém-escrito. De modo similar, quando duas writes acontecem em rápida sucessão, seguidas de uma read, o valor lido é o valor armazenado na última escrita. Na realidade, o sistema obriga um ordenamento em todas as chamadas de sistema, e todos os processadores veem o mesmo ordenamento. Nós nos referiremos a esse modelo como consistência sequencial.

Em um sistema distribuído, a consistência sequencial pode ser conseguida facilmente desde que exista apenas um servidor de arquivos e os clientes não armazenem arquivos em cache. Todas as reads e writes vão diretamente para o servidor de arquivos, que as processa de maneira estritamente sequencial.

Na prática, no entanto, o desempenho de um sistema distribuído no qual todas as solicitações de arquivos devem ir para um único servidor é muitas vezes fraco. Esse problema é frequentemente solucionado permitindo que clientes mantenham cópias locais de arquivos pesadamente usados em suas caches privadas. No entanto, se o cliente 1 modificar um arquivo armazenado em cache localmente e logo em seguida o cliente 2 ler o arquivo do servidor, o segundo cliente receberá um arquivo obsoleto.

Uma saída para essa dificuldade é propagar todas as mudanças para arquivos em cache de volta para o servidor imediatamente. Quando o cliente 1 fecha o arquivo, ele envia uma cópia de volta para o servidor, então reads subsequentes recebem um novo valor, como solicitado. De fato, esse é o modelo upload/download. Essa semântica é amplamente implementada e é conhecida como semântica de sessão.

Middleware baseado em objetos

Agora vamos examinar um terceiro paradigma. Em vez de dizer que tudo é um documento ou tudo é um arquivo, dizemos que tudo é um objeto. Um objeto é uma coleção de variáveis que são colocadas juntas com um conjunto de rotinas de acesso, chamadas métodos. Processos não têm permissão para acessar as variáveis diretamente. Em vez disso, eles precisam invocar os métodos.

Um sistema bastante conhecido baseado em objetos em tempo de execução é o CORBA. CORBA é um sistema cliente-servidor, no qual os processos clientes podem invocar operações em objetos localizados em (possivelmente remotas) máquinas servidoras. CORBA foi projetado para um sistema heterogêneo executando uma série de plataformas de hardware e sistemas operacionais e programado em uma série de linguagens. Para tornar possível para um cliente em uma plataforma invocar um cliente em uma plataforma diferente, ORBs (Object Request Brokers — agentes de solicitação de objetos) são interpostos entre o cliente e o servidor para permitir que eles se correspondam. Os ORBs desempenham um papel importante no CORBA, fornecendo mesmo seu nome ao sistema.

Para invocar um método em um objeto, um processo cliente deve primeiro adquirir uma referência para aquele objeto. A referência pode vir diretamente do processo criador ou, de maneira mais provável, procurando-a pelo nome ou função em algum tipo de diretório. Uma vez que a referência do objeto está disponível, o processo cliente prepara os parâmetros para as chamadas dos métodos em uma estrutura conveniente e então contata o ORB cliente. Por sua vez, o ORB cliente envia uma mensagem para o ORB servidor, que na realidade invoca o método sobre o objeto. Todo o mecanismo é similar à RPC.

A função dos ORBs é esconder toda a distribuição de baixo nível e detalhes de comunicação dos códigos do cliente e do servidor. Em particular, os ORBs escondem do cliente a localização do servidor.

Para possibilitar o uso de objetos no CORBA que não foram escritos para o sistema, cada objeto pode ser equipado com um adaptador de objeto. Trata-se de um invólucro que realiza tarefas, como registrar o objeto, gerar referências do objeto e ativar o objeto, se ele for invocado quando não estiver ativo.

Um sério problema com o CORBA é que todos objetos estão localizados somente em um servidor, o que significa que o desempenho será terrível para objetos que são muito usados em máquinas clientes mundo afora. Na prática, o CORBA funciona de maneira aceitável somente em sistema de pequena escala, como para conectar processos em um computador, uma LAN, ou dentro de uma única empresa.

Middleware baseado em coordenação

Nosso último paradigma para um sistema distribuído é chamado de middleware baseado em coordenação.

Linda é um sistema moderno para comunicação e sincronização desenvolvido na Universidade de Yale por David Gelernter e seu estudante Nick Carriero.

No Linda, processos independentes comunicam-se por um espaço de tuplas abstrato. O espaço de tuplas é global para todo o sistema, e processos em qualquer máquina podem inserir tuplas no espaço de tuplas ou removê-las deste espaço sem levar em consideração como ou onde elas estão armazenadas. Para o usuário, o espaço de tuplas parece uma grande memória compartilhada global.

Uma tupla é como uma estrutura em C ou Java. Ela consiste em um ou mais campos, cada um dos quais é um valor de algum tipo suportado pela linguagem base.

Publicar/assinar

Nosso próximo exemplo de um modelo baseado em coordenação foi inspirado em Linda e é chamado de publicar/assinar (publish/subscribe) (OKI et al., 1993). Ele consiste em uma série de processos conectados por uma rede de difusão. Cada processo pode ser um produtor de informações, um consumidor de informações, ou ambos.

Quando um produtor de informações tem uma informação nova (por exemplo, um novo preço de uma ação), ele transmite a informação como uma tupla na rede. Essa ação é chamada de publicação. Cada tupla contém uma linha de assunto hierárquica contendo múltiplos campos separados por períodos. Processos que estão interessados em determinadas informações podem assinar para receber determinados assuntos, incluindo o uso de símbolos na linha do assunto. A assinatura é feita dizendo a um processo daemon de tuplas, na mesma máquina que monitora tuplas publicadas, quais assuntos procurar.

O modelo publicar/assinar desacopla inteiramente os produtores dos consumidores, assim como faz o Linda. Entretanto, às vezes é útil saber quem mais está lá. Essa informação pode ser adquirida publicando a tupla que basicamente pergunta: “Quem ai está interessado em x?” Respostas retornam em forma de tuplas que dizem: “Eu estou interessado em x”.