Elementos de design do sistema de VM do FreeBSD

Resumo

O título é realmente apenas uma maneira extravagante de dizer que vou tentar descrever todo o grupo de itens de uma VM, espero que de uma forma que todos possam acompanhar. Pelo ultimo ano eu me concentrei em vários dos principais subsistemas do kernel dentro do FreeBSD, com os subsistemas VM e Swap sendo os mais interessantes e o NFS sendo "uma tarefa necessária". Eu reescrevi apenas pequenas partes do código. Na área de VM, a única grande reescrita que fiz foi para o subsistema de troca. A maior parte do meu trabalho foi de limpeza e manutenção, com apenas uma moderada reescrita de código e sem grandes ajustes nos algorítimos do subsistema VM. A maior parte da base teórica do subsistema VM permanece inalterada e muito do crédito pelo esforço de modernização nos últimos anos pertence a John Dyson e David Greenman. Não sendo um historiador como Kirk, eu não tentarei marcar todos os vários recursos com nomes de pessoas, já que invariavelmente vou errar.

Índice

1. Introdução

Antes de avançarmos para o design atual, vamos dedicar um pouco de tempo a necessidade de manter e modernizar qualquer base de código duradoura. No mundo da programação, os algoritmos tendem a ser mais importantes do que o código, e é precisamente devido as raízes acadêmicas do BSD que uma grande atenção foi dada ao design do algoritmo desde o início. Mais atenção ao design geralmente leva a uma base de código limpa e flexível que pode ser facilmente modificada, estendida ou substituída ao longo do tempo. Embora o BSD seja considerado um sistema operacional "antigo" por algumas pessoas, aqueles de nós que trabalham nele tendem a vê-lo mais como uma base de código "madura" que possui vários componentes modificados, estendidos, ou substituído por código moderno. Ele evoluiu e o FreeBSD está no topo, não importa

quantos anos tenha o código. Esta é uma distinção importante a ser feita e infelizmente perdida para muitas pessoas. O maior erro que um programador pode cometer é não aprender com a história, e esse é precisamente o erro que muitos outros sistemas operacionais modernos cometeram. Windows NT® é o melhor exemplo disso, e as conseqüências foram terríveis. O Linux também cometeu esse erro até certo ponto - o suficiente para que nós, do BSD, possamos fazer pequenas piadas sobre isso de vez em quando, entretanto. O problema do Linux é simplesmente a falta de experiência e histórico para comparar idéias, um problema que está sendo resolvido de forma fácil e rápida pela comunidade Linux, da mesma forma como foi abordado na comunidade BSD - pelo desenvolvimento contínuo de código. Por outro lado, o povo do Windows NT®, repetidamente comete os mesmos erros resolvidos no UNIX® décadas atrás e depois gasta anos corrigindo-os. De novo e de novo. Eles têm um caso grave de "não foi projetado aqui" e "estamos sempre certos porque nosso departamento de marketing diz que sim". Tenho pouca tolerância para quem não pode aprender com a história.

Grande parte da complexidade aparente do design do FreeBSD, especialmente no subsistema VM/Swap, é um resultado direto de ter que resolver sérios problemas de desempenho que ocorrem sob várias condições. Estes problemas não se devem ao mau design de algorítimo, mas sim a fatores ambientais. Em qualquer comparação direta entre plataformas, estes problemas tornam-se mais aparentes quando os recursos do sistema começam a ficar estressados. Como descrevo o subsistema VM/Swap do FreeBSD, o leitor deve sempre manter dois pontos em mente:

1. O aspecto mais importante do design de desempenho é o que é conhecido como "Otimizando o Caminho Crítico ". Muitas vezes, as otimizações de desempenho inflam um pouco o código, para que o caminho crítico tenha um melhor desempenho.
2. Um design sólido e generalizado supera um projeto altamente otimizado a longo prazo. Enquanto um design generalizado pode acabar sendo mais lento do que um projeto altamente otimizado quando eles são implementados pela primeira vez, o design generalizado tende a ser mais fácil de se adaptar as mudanças de condições e o projeto altamente otimizado acaba tendo que ser descartado.

Qualquer base de código que sobreviva e seja sustentável por anos deve, portanto, ser projetada adequadamente desde o início, mesmo que isso custe algum desempenho. Vinte anos atrás, as pessoas ainda argumentavam que programar em assembly era melhor do que programar em uma linguagem de alto nível porque produzia um código que era dez vezes mais rápido. Hoje, a queda desse argumento é óbvia - assim como os paralelos com o design de algorítimo e a generalização de código.

2. Objetos de VM

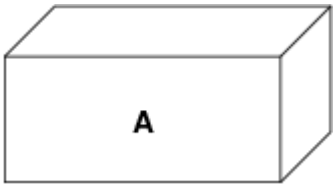
A melhor maneira de começar a descrever o sistema de VM do FreeBSD é examiná-lo da perspectiva de um processo em nível de usuário. Cada processo do usuário vê um espaço de endereço de VM único, privado e contíguo, contendo vários tipos de objetos de memória. Esses objetos possuem várias características. O código do programa e os dados do programa são efetivamente um único arquivo mapeado na memória (o arquivo binário sendo executado), mas o código do programa é read-only enquanto os dados do programa são copy-on-write. O programa BSS é apenas memória alocada e preenchida com zeros sob demanda, chamado de demanda de preenchimento de página com zero. Arquivos arbitrários também podem ser mapeados na

memória dentro do espaço de endereçamento como bem entender, que é como o mecanismo de biblioteca compartilhada funciona. Esses mapeamentos podem exigir modificações para permanecerem privados para o processo que os produz. A chamada do sistema de fork adiciona uma dimensão totalmente nova ao problema de gerenciamento de VMs além da complexidade já fornecida.

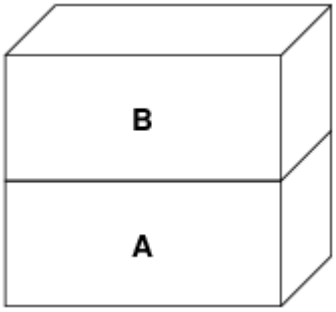
Uma página de dados binários do programa (que é uma página básica de copy-on-write) ilustra a complexidade. Um programa binário contém uma seção de dados pré-inicializada que é inicialmente mapeada diretamente a partir do arquivo de programa. Quando um programa é carregado no espaço de VM de um processo, esta área é inicialmente mapeada na memória e suportada pelo próprio binário do programa, permitindo que o sistema de VM liberte/reutilize a página e depois carregue-a de volta a partir do binário. No entanto, no momento em que um processo modifica esses dados, o sistema de VM deve fazer uma cópia privada da página para esse processo. A partir do momento que a cópia privada tenha sido modificada, o sistema de VM pode não mais liberá-la, porque não há mais como restaurá-la depois.

Você notará imediatamente que o que originalmente era um mapeamento de arquivo simples se tornou muito mais complexo. Os dados podem ser modificados página a página, enquanto o mapeamento de arquivos abrange muitas páginas de uma só vez. A complexidade aumenta ainda mais quando existe um fork do processo. Quando um processo se duplica, o resultado são dois processos - cada um com seus próprios espaços de endereçamento privados, incluindo quaisquer modificações feitas pelo processo original antes de chamar um `fork()`. Seria bobagem o sistema de VM fizesse uma cópia completa dos dados no momento do `fork()` porque é bem possível que pelo menos um dos dois processos precise apenas ler essa página a partir de então, permitindo que a página original continue a ser usada. O que era uma página privada é feito um copy-on-write novamente, já que cada processo (pai e filho) espera que suas próprias modificações pós-fork permaneçam privadas para si mesmas e não afetem a outra.

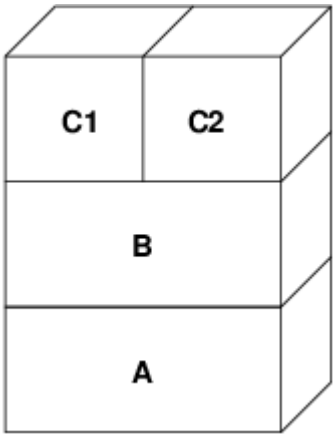
O FreeBSD gerencia tudo isso com um modelo de objetos de VM em camadas. O arquivo de programa binário original acaba sendo a camada de objeto de VM mais baixa. Uma camada copy-on-write é colocada acima dela para conter as páginas que tiveram que ser copiadas do arquivo original. Se o programa modificar uma página de dados pertencente ao arquivo original, o sistema de VM assumirá que existe uma falha e fará uma cópia da página na camada superior. Quando existe um fork do processo, as camadas adicionais de objetos de VM são ativadas. Isso pode fazer um pouco mais de sentido com um exemplo bastante básico. Um `fork()` é uma operação comum para qualquer sistema *BSD, então este exemplo irá considerar um programa que inicia e é feito um fork. Quando o processo é iniciado, o sistema de VM cria uma camada de objeto, vamos chamar isso de A:



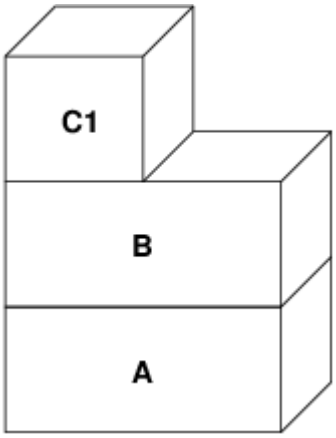
A representa o arquivo - as páginas podem ser paginadas dentro e fora da mídia física do arquivo, conforme necessário. Paginar a partir do disco é razoável para um programa, mas nós realmente não queremos voltar atrás e sobrescrever o executável. O sistema de VM, portanto, cria uma segunda camada, B, que será fisicamente suportada pelo espaço de troca:



Na primeira escrita em uma página depois disso, uma nova página é criada em B e seu conteúdo é inicializado a partir de A. Todas as páginas em B podem ser paginadas para dentro ou para fora por um dispositivo de troca. Quando é feito o fork do programa, o sistema de VM cria duas novas camadas de objetos - C1 para o processo pai e C2 para o filho - que ficam no topo de B:



Neste caso, digamos que uma página em B seja modificada pelo processo pai original. O processo terá uma falha de copy-on-write e duplicará a página em C1, deixando a página original em B intocada. Agora, digamos que a mesma página em B seja modificada pelo processo filho. O processo assumirá uma falha de copy-on-write e duplicará a página em C2. A página original em B agora está completamente oculta, já que C1 e C2 têm uma cópia e B poderia, teoricamente, ser destruído se não representasse um arquivo "real"; no entanto, esse tipo de otimização não é trivial de se fazer, porque é muito refinado. O FreeBSD não faz essa otimização. Agora, suponha (como é frequentemente o caso) que o processo filho execute um `exec()`. Seu espaço de endereço atual é geralmente substituído por um novo espaço de endereço representando um novo arquivo. Nesse caso, a camada C2 é destruída:



Neste caso, o número de filhos de B cai para um, e todos os acessos para B passam agora por C1. Isso significa que B e C1 podem ser unidas. Todas as páginas em B que também existem em C1 são

excluídas de B durante a união. Assim, mesmo que a otimização na etapa anterior não possa ser feita, podemos recuperar as páginas mortas quando um dos processos finalizar ou executar um `exec()`.

Este modelo cria vários problemas potenciais. O primeiro é que você pode acabar com uma pilha relativamente profunda de objetos de VM em camadas, que pode custar tempo de varredura e memória quando ocorrer uma falha. Camadas profundas podem ocorrer quando houver forks dos processos e, em seguida, houver um fork novamente (do processo pai ou filho). O segundo problema é que você pode acabar com páginas profundas inacessíveis e mortas no meio da pilha de objetos de VM. Em nosso último exemplo, se os processos pai e filho modificarem a mesma página, ambos receberão suas próprias cópias privadas da página e a página original em B não poderá mais ser acessada por ninguém. Essa página em B pode ser liberada.

O FreeBSD resolve o problema de camadas profundas com uma otimização especial chamada "All Shadowed Case". Este caso ocorre se C1 ou C2 tiverem falhas de COW suficientes para fazer uma copia de sombra completa de todas as páginas em B. Digamos que C1 consiga isso. C1 agora pode ignorar B completamente, então, em vez de temos C1→B→A e C2→B→A temos agora C1→A e C2→B→A. Mas veja o que também aconteceu - agora B tem apenas uma referência (C2), então podemos unir B e C2. O resultado final é que B é deletado inteiramente e temos C1→A e C2→A. É comum que B contenha um grande número de páginas e nem C1 nem C2 possam ofuscar completamente. Se nós forçarmos novamente e criarmos um conjunto de camadas D, no entanto, é muito mais provável que uma das camadas D eventualmente seja capaz de ofuscar completamente o conjunto de dados muito menor representado por C1 ou C2. A mesma otimização funcionará em qualquer ponto do gráfico e o grande resultado disso é que, mesmo em uma máquina diversos forks, pilhas de objetos da VM tendem a não ficar muito mais profundas do que 4. Isso é verdade tanto para o processo pai quanto para os filhos e verdadeiro quer seja o processo pai fazendo o fork ou os processos filhos fazendo forks em cascata.

O problema da página morta ainda existe no caso em que C1 ou C2 não ofuscaram completamente as páginas de B. Devido as nossas outras otimizações, este caso não representa um grande problema e simplesmente permitimos que as páginas fiquem inativas. Se o sistema ficar com pouca memória, ele irá trocá-las, comendo uma pequena parte da swap, mas é isso.

A vantagem do modelo de objetos de VM é que o `fork()` é extremamente rápido, já que não é necessária nenhuma cópia de dados real. A desvantagem é que você pode criar uma camada de Objetos de VM relativamente complexa que reduz um pouco o tratamento de falhas de página e gasta memória gerenciando as estruturas de Objetos de VM. As otimizações que o FreeBSD faz prova reduzir os problemas o suficiente para que as falhas possam ser ignoradas, não deixando nenhuma desvantagem real.

3. Camadas de SWAP

As páginas de dados privadas são inicialmente páginas copy-on-write ou zero-fill. Quando uma alteração e, portanto, uma cópia, é feita, o objeto de apoio original (geralmente um arquivo) não pode mais ser usado para salvar uma cópia da página quando o sistema da VM precisar reutilizá-lo para outras finalidades. É aí que o SWAP entra. O SWAP é alocado para criar um suporte de armazenamento para a memória que não o possui. O FreeBSD aloca a estrutura de gerenciamento de troca para um objeto de VM somente quando for realmente necessário. No entanto,

historicamente, a estrutura de gerenciamento de troca teve problemas:

- Sob o FreeBSD 3.X, a estrutura de gerenciamento de swap pré-aloca uma matriz que engloba todo o objeto que requer suporte para armazenamento da swap - mesmo que apenas algumas páginas desse objeto sejam suportadas por swap. Isto cria um problema de fragmentação de memória do kernel quando grandes objetos são mapeados ou processos com fork de grandes runsizes (RSS).
- Além disso, para manter o controle do espaço de swap, uma "lista de espaços vazios" é mantida na memória do kernel, e isso tende a ficar severamente fragmentado também. Como a lista "de espaços vazios" é uma lista linear, o desempenho de alocação e liberação de swap é uma troca O(n)-per-page (Uma por página) não ideal.
- Requer que as alocações de memória do kernel ocorram durante o processo de troca de swap, e isto cria problemas de deadlock de pouca memória.
- O problema é ainda mais exacerbado por buracos criados devido ao algoritmo de intercalação.
- Além disso, o mapa de blocos da swap pode se fragmentar com bastante facilidade, resultando em alocações não contíguas.
- A memória do kernel também deve ser alocada dinamicamente para estruturas adicionais de gerenciamento da swap quando ocorre uma troca.

É evidente a partir dessa lista que havia muito espaço para melhorias. Para o FreeBSD 4.X, eu reescrevi completamente o subsistema de swap:

- As estruturas de gerenciamento de swap são alocadas por meio de uma tabela de hash, em vez de um array linear, fornecendo um tamanho de alocação fixo e uma granularidade muito mais fina.
- Em vez de usar uma lista vinculada linearmente para acompanhar as reservas de espaço de troca, ele agora usa um bitmap de blocos de troca organizados em uma estrutura de árvores raiz com dicas de espaço livre nas estruturas do nó de origem. Isto efetivamente faz a alocação de swap e libera uma operação O(1).
- Todo o bitmap da árvore raiz também é pré-alocado para evitar ter que alocar a memória do kernel durante operações críticas de troca com memória baixa. Afinal de contas, o sistema tende a trocar quando está com pouca memória, por isso devemos evitar a alocação da memória do kernel nesses momentos para evitar possíveis deadlocks.
- Para reduzir a fragmentação, a árvore raiz é capaz de alocar grandes blocos contíguos de uma só vez, pulando pedaços menores e fragmentados.

Eu não dei o último passo de ter um "ponteiro de sugestão de alocação" que percorria uma porção da swap conforme as alocações eram feitas a fim de garantir alocações contíguas ou pelo menos a referência localmente, mas assegurei que tal adição poderia ser feita.

4. Quando libertar uma página

Como o sistema de VM usa toda a memória disponível para o cache em disco, geralmente há poucas páginas realmente livres. O sistema de VM depende de poder escolher corretamente as páginas que não estão em uso para reutilizar em novas alocações. Selecionar as páginas ideais para liberar é

possivelmente a função mais importante que qualquer sistema de VM pode executar, porque se fizer uma seleção ruim, o sistema de VM poderá ser desnecessariamente forçado a recuperar páginas do disco, prejudicando seriamente o desempenho do sistema.

Quanta sobrecarga estamos dispostos a sofrer no caminho crítico para evitar a liberação da página errada? Cada escolha errada que fazemos nos custará centenas de milhares de ciclos da CPU e uma paralisação notável dos processos afetados, por isto estamos dispostos a suportar uma quantidade significativa de sobrecarga, a fim de ter certeza de que a página certa é escolhida. É por isto que o FreeBSD tende a superar outros sistemas quando os recursos de memória ficam estressados.

O algoritmo de determinação de página livre é construído sobre um histórico do uso das páginas de memória. Para adquirir este histórico, o sistema tira proveito de um recurso de um bit usado pela página que a maioria das tabelas de página de hardware possui.

Em qualquer caso, o bit usado na página é desmarcado e, em algum momento posterior, o sistema de VM encontra a página novamente e vê que o bit usado na página foi definido. Isso indica que a página ainda está sendo usada ativamente. Se o bit ainda estiver desmarcado, é uma indicação de que a página não está sendo usada ativamente. Ao testar este bit periodicamente, é desenvolvido um histórico de uso (na forma de um contador) para a página física. Quando, posteriormente, o sistema de VM precisar liberar algumas páginas, a verificação desse histórico se tornará a base da determinação da melhor página candidata a ser reutilizada.

Para as plataformas que não possuem esse recurso, o sistema realmente emula um bit usado na página. Ele remove o mapeamento ou protege uma página, forçando uma falha de página se a página for acessada novamente. Quando a falha de página acontece, o sistema simplesmente marca a página como tendo sido usada e desprotege a página para que ela possa ser usada. Embora a tomada de tais falhas de página apenas para determinar se uma página está sendo usada pareça ser uma proposta cara, é muito menos dispendioso do que reutilizar a página para outra finalidade, apenas para descobrir que um processo precisa dela e depois ir para o disco .

O FreeBSD faz uso de várias filas de páginas para refinar ainda mais a seleção de páginas para reutilização, bem como para determinar quando páginas inativas devem ser liberadas para o suporte ao armazenamento. Como as tabelas de páginas são entidades dinâmicas sob o FreeBSD, não custa virtualmente nada desmapear uma página do espaço de endereço de qualquer processo que a utilize. Quando uma página cadidata ser escolhida com base no contador de uso de página, isso é precisamente o que é feito. O sistema deve fazer uma distinção entre páginas limpas que teoricamente podem ser liberadas a qualquer momento, e páginas inativas que devem primeiro ser escritas em seu repositório de armazenamento antes de serem reutilizáveis. Quando uma página candidata for encontrada, ela será movida para a fila inativa, se estiver inativas, ou para a fila de cache, se estiver limpa. Um algoritmo separado baseado na proporção de páginas inativas para limpas determina quando páginas inativas na fila inativa devem ser liberadas para o disco. Depois que isso for feito, as páginas liberadas serão movidas da fila inativa para a fila de cache. Neste ponto, as páginas na fila de cache ainda podem ser reativadas por uma falha de VM a um custo relativamente baixo. No entanto, as páginas na fila de cache são consideradas "imediatamente livres" e serão reutilizadas em uma forma LRU (usada menos recentemente) quando o sistema precisar alocar nova memória.

É importante notar que o sistema de VM do FreeBSD tenta separar páginas limpas e inativas pelo motivo expresso de evitar descargas desnecessárias de páginas inativas (que consomem largura de

banda de I/O), nem move páginas entre as várias filas de páginas gratuitamente quando o subsistema de memória não está sendo enfatizado. É por isto que você verá alguns sistemas com contagens de fila de cache muito baixas e contagens alta de fila ativa ao executar um comando `systat -vm`. À medida que o sistema de VM se torna mais estressado, ele faz um esforço maior para manter as várias filas de páginas nos níveis determinados para serem mais eficazes.

Uma lenda urbana circulou durante anos que o Linux fez um trabalho melhor evitando trocas do que o FreeBSD, mas isso de fato não é verdade. O que estava realmente ocorrendo era que o FreeBSD estava proativamente numerando páginas não usadas a fim de abrir espaço para mais cache de disco enquanto o Linux mantinha páginas não utilizadas no núcleo e deixando menos memória disponível para páginas de cache e processo. Eu não sei se isso ainda é verdade hoje.

5. Otimizações de Pré-Falhas ou para Zerar

Pegar uma falha de VM não é caro se a página subjacente já estiver no núcleo e puder simplesmente ser mapeada no processo, mas pode se tornar cara se você pegar muitas delas regularmente. Um bom exemplo disso é executar um programa como ls(1) ou ps(1) várias vezes. Se o programa binário é mapeado na memória, mas não mapeado na tabela de páginas, então todas as páginas que serão acessadas pelo programa irão estar com falha toda vez que o programa for executado. Isso é desnecessário quando as páginas em questão já estão no cache de VM, então o FreeBSD tentará preencher previamente as tabelas de páginas de um processo com as páginas que já estão no cache de VM. Uma coisa que o FreeBSD ainda não faz é pré-copiar-durante-escrita certas páginas no exec. Por exemplo, se você executar o programa ls(1) ao executar o `vmstat 1`, notará que sempre pega um determinado número de falhas de página, mesmo quando você o executa várias vezes. Estas são falhas de preenchimento com zero, não falhas de código de programa (que já foram pré-falhas). A pré-cópia de páginas em exec ou fork é uma área que poderia se utilizar de mais estudos.

Uma grande porcentagem de falhas de página que ocorrem são falhas de preenchimento com zero. Geralmente, você pode ver isso observando a saída de `vmstat -s`. Estas falhas ocorrem quando um processo acessa páginas em sua área BSS. Espera-se que a área BSS seja inicialmente zero, mas o sistema de VM não se preocupa em alocar memória alguma até que o processo realmente a acesse. Quando ocorre uma falha, o sistema de VM deve alocar não apenas uma nova página, mas deve zerá-la também. Para otimizar a operação de zeramento, o sistema de VM tem a capacidade de pré-zerar páginas e marcá-las como tal, e solicitar páginas pré-zeradas quando ocorrem falhas de preenchimento com zero. O pré-zeramento ocorre sempre que a CPU está inativa, mas o número de páginas que o sistema pre-zeros é limitado, a fim de evitar que os caches de memória sejam dissipados. Este é um excelente exemplo de adição de complexidade ao sistema de VM para otimizar o caminho crítico.

6. Otimizações da Tabela de Páginas

As otimizações da tabela de páginas constituem a parte mais contenciosa do design de VM do FreeBSD e mostraram alguma tensão com o advento do uso sério de `mmap()`. Eu acho que isso é realmente uma característica da maioria dos BSDs, embora eu não tenha certeza de quando foi introduzido pela primeira vez. Existem duas otimizações principais. A primeira é que as tabelas de páginas de hardware não contêm estado persistente, mas podem ser descartadas a qualquer

momento com apenas uma pequena quantidade de sobrecarga de gerenciamento. A segunda é que cada entrada ativa da tabela de páginas no sistema tem uma estrutura governante `pv_entry` que é amarrada na estrutura `vm_page`. O FreeBSD pode simplesmente iterar através desses mapeamentos que são conhecidos, enquanto o Linux deve verificar todas as tabelas de páginas que *possam* conter um mapeamento específico para ver se ele o faz, o que pode alcançar O(n^2) situações. É por isso que o FreeBSD tende a fazer melhores escolhas em quais páginas reutilizar ou trocar quando a memória é estressada, dando-lhe melhor desempenho em sobrecarga. No entanto, o FreeBSD requer o ajuste do kernel para acomodar situações de grandes espaços de endereços compartilhados, como aquelas que podem ocorrer em um sistema de notícias, porque ele pode rodar sem estruturas `pv_entry`.

Tanto o Linux quanto o FreeBSD precisam funcionar nesta área. O FreeBSD está tentando maximizar a vantagem de um modelo de mapeamento ativo potencialmente esparso (nem todos os processos precisam mapear todas as páginas de uma biblioteca compartilhada, por exemplo), enquanto o Linux está tentando simplificar seus algoritmos. O FreeBSD geralmente tem a vantagem de desempenho aqui, ao custo de desperdiçar um pouco de memória extra, mas o FreeBSD quebra no caso em que um arquivo grande é massivamente compartilhado em centenas de processos. O Linux, por outro lado, se quebra no caso em que muitos processos mapeiam esparsamente a mesma biblioteca compartilhada e também são executados de maneira não ideal ao tentar determinar se uma página pode ser reutilizada ou não.

7. Page Coloring

Terminaremos com as otimizações de page coloring. Page coloring é uma otimização de desempenho projetada para garantir que acessos a páginas contíguas na memória virtual façam o melhor uso do cache do processador. Nos tempos antigos (isto é, há mais de 10 anos), os caches de processador tendiam a mapear a memória virtual em vez da memória física. Isso levou a um grande número de problemas, incluindo a necessidade de limpar o cache em cada troca de contexto em alguns casos e problemas com o alias de dados no cache. Caches de processador modernos mapeiam a memória física com precisão para resolver esses problemas. Isto significa que duas páginas lado a lado em um espaço de endereço de processos podem não corresponder a duas páginas lado a lado no cache. Na verdade, se você não for cuidadoso, as páginas lado a lado na memória virtual podem acabar usando a mesma página no cache do processador - conduzindo para que dados em cache sejam descartados prematuramente e reduzindo o desempenho da CPU. Isto é verdade mesmo com caches auto associativos de múltiplas vias (embora o efeito seja um pouco mitigado).

O código de alocação de memória do FreeBSD implementa otimizações de page coloring, o que significa que o código de alocação de memória tentará localizar páginas livres contíguas do ponto de vista do cache. Por exemplo, se a página 16 da memória física for atribuída à página 0 da memória virtual de um processo e o cache puder conter 4 páginas, o código de page coloring não atribuirá a página 20 da memória física a página 1 da memória virtual de um processo. Em vez disso, atribui a página 21 da memória física. O código de page coloring tenta evitar assimilar a página 20, porque ela é mapeada sobre a mesma memória cache da página 16 e resultaria em um armazenamento não otimizado. Este código adiciona uma quantidade significativa de complexidade ao subsistema de alocação de memória de VM, como você pode imaginar, mas o resultado vale o esforço. Page coloring torna a memória de VM tão determinante quanto a memória

física em relação ao desempenho do cache.

8. Conclusão

A memória virtual em sistemas operacionais modernos deve abordar vários problemas diferentes de maneira eficiente e para muitos padrões de uso diferentes. A abordagem modular e algorítmica que o BSD historicamente teve nos permite estudar e entender a implementação atual, bem como substituir de forma relativamente limpa grandes seções do código. Houve uma série de melhorias no sistema de VM do FreeBSD nos últimos anos e o trabalho está em andamento.

9. Sessão bônus de QA por Allen Briggs briggs@ninthwonder.com

9.1. O que é o algoritmo de intercalação ao qual você se refere em sua listagem dos males dos arranjos de swap do FreeBSD 3.X?

O FreeBSD usa um intercalador de swap fixo, cujo padrão é 4. Isso significa que o FreeBSD reserva espaço para quatro áreas de swap, mesmo se você tiver apenas uma, duas ou três. Como a swap é intercalada, o espaço de endereçamento linear representando as "quatro áreas de troca" estará fragmentado se você não tiver quatro áreas de troca. Por exemplo, se você tiver duas áreas de swap, A e B, a representação do espaço de endereçamento do FreeBSD para esta área de troca será intercalada em blocos de 16 páginas:

```
A B C D A B C D A B C D A B C D
```

O FreeBSD 3.X usa uma abordagem de "lista sequencial de regiões livres" para contabilizar as áreas de swap livres. A ideia é que grandes blocos de espaço linear livre possam ser representados com um único nó da lista (kern/subr_rlist.c). Mas devido a fragmentação, a lista sequencial acaba sendo insanamente fragmentada. No exemplo acima, a swap completamente sem uso terá A e B mostrados como "livres" e C e D mostrados como "todos alocados". Cada sequência A-B requer um nó da lista para considerar porque C e D são buracos, portanto, o nó de lista não pode ser combinado com a próxima sequência A-B.

Por que nós intercalamos nosso espaço de swap em vez de apenas colocar as áreas de swap no final e fazer algo mais sofisticado? Porque é muito mais fácil alocar trechos lineares de um espaço de endereçamento e ter o resultado automaticamente intercalado em vários discos do que tentar colocar esta sofisticação em outro lugar.

A fragmentação causa outros problemas. Sendo uma lista linear sob 3.X, e tendo uma enorme quantidade de fragmentação inerente, alocando e liberando swap leva a ser um algoritmo O(N) ao invés de um algoritmo O(1). Combinado com outros fatores (troca pesada) e você começa a entrar em níveis de sobrecarga O(N^2) e O(N^3), o que é ruim. O sistema 3.X também pode precisar alocar

o KVM durante uma operação de troca para criar um novo nó da lista que pode levar a um impasse se o sistema estiver tentando fazer uma liberação de página em uma situação de pouca memória.

No 4.X, não usamos uma lista sequencial. Em vez disto, usamos uma árvore raiz e bitmaps de blocos de swap em vez de lista de nós variáveis. Aceitamos o sucesso de pré-alocar todos os bitmaps necessários para toda a área de swap na frente, mas acaba desperdiçando menos memória devido ao uso de um bitmap (um bit por bloco) em vez de uma lista encadeada de nós. O uso de uma árvore raiz em vez de uma lista sequencial nos dá quase o desempenho O(1), não importa o quão fragmentada a árvore se torne.

9.2. Como a separação de páginas limpas e sujas (inativas) está relacionada à situação em que você vê baixas contagens de filas de cache e altas contagens de filas ativas no systat -vm? As estatísticas do systat rolam as páginas ativa e inativas juntas para a contagem de filas ativas?

Sim, isto é confuso. A relação é "meta" versus "realidade". Nosso objetivo é separar as páginas, mas a realidade é que, se não estamos em uma crise de memória, não precisamos realmente fazer isso.

O que isto significa é que o FreeBSD não tentará muito separar páginas sujas (fila inativa) de páginas limpas (fila de cache) quando o sistema não está sendo estressado, nem vai tentar desativar páginas (fila ativa → fila inativa) quando o sistema não está sendo estressado, mesmo que não estejam sendo usados.

9.3. No exemplo ls1 / vmstat 1, algumas falhas de página não seriam falhas de página de dados (COW do arquivo executável para a página privada)? Ou seja, eu esperaria que as falhas de página fossem um preenchimento com zero e alguns dados do programa. Ou você está sugerindo que o FreeBSD faz pré-COW para os dados do programa?

Uma falha de COW pode ser preenchimento com zero ou dados de programa. O mecanismo é o mesmo dos dois modos, porque os dados do programa de apoio quase certamente já estão no cache. Eu estou realmente juntando os dois. O FreeBSD não faz o pré-COW dos dados do programa ou preenchimento com zero, mas *faz* pré-mapeamento de páginas que existem em seu cache.

9.4. Em sua seção sobre otimizações de tabela de páginas, você pode dar um pouco mais de detalhes sobre pv_entry e vm_page (ou vm_page deveria ser vm_pmap- como em 4.4, cf. pp. 180-181 of McKusick, Bostic, Karel, Quarterman)? Especificamente, que tipo de operação/reação exigiria a varredura dos mapeamentos?

Uma `vm_page` representa uma tupla (objeto,índice#). Um `pv_entry` representa uma entrada de tabela de página de hardware (pte). Se você tem cinco processos compartilhando a mesma página física, e três dessas tabelas de páginas atualmente mapeiam a página, esta página será representada por uma única estrutura `vm_page` e três estruturas `pv_entry`.

As estruturas `pv_entry` representam apenas as páginas mapeadas pela MMU (uma `pv_entry` representa uma pte). Isso significa que quando precisamos remover todas as referências de hardware para uma `vm_page` (para reutilizar a página para outra coisa, paginar, limpar, inativar e assim por diante), podemos simplesmente escanear a lista encadeada de `pv_entry` associada a essa `vm_page` para remover ou modificar os pte's de suas tabelas de páginas.

No Linux, não existe essa lista vinculada. Para remover todos os mapeamentos de tabelas de páginas de hardware para um `vm_page`, o linux deve indexar em todos os objetos de VM que *possam* ter mapeado a página. Por exemplo, se você tiver 50 processos, todos mapeando a mesma biblioteca compartilhada e quiser se livrar da página X nessa biblioteca, será necessário indexar na tabela de páginas para cada um desses 50 processos, mesmo se apenas 10 deles realmente tiverem mapeado a página. Então, o Linux está trocando a simplicidade de seu design com o desempenho. Muitos algoritmos de VM que são O(1) ou (pequeno N) no FreeBSD acabam sendo O(N), O(N^2), ou pior no Linux. Como os pte's que representam uma determinada página em um objeto tendem a estar no mesmo offset em todas as tabelas de páginas em que estão mapeados, reduzir o número de acessos nas tabelas de páginas no mesmo pte offset evitará a linha de cache L1 para esse deslocamento, o que pode levar a um melhor desempenho.

O FreeBSD adicionou complexidade (o esquema `pv_entry`) para aumentar o desempenho (para limitar os acessos da tabela de páginas a *somente* aqueles pte's que precisam ser modificados).

Mas o FreeBSD tem um problema de escalonamento que o Linux não possui, pois há um número limitado de estruturas `pv_entry` e isso causa problemas quando você tem um compartilhamento massivo de dados. Nesse caso, você pode ficar sem estruturas `pv_entry`, mesmo que haja bastante memória livre disponível. Isto pode ser corrigido com bastante facilidade aumentando o número de estruturas `pv_entry` na configuração do kernel, mas realmente precisamos encontrar uma maneira melhor de fazê-lo.

Em relação à sobrecarga de memória de uma tabela de páginas verso do esquema `pv_entry`: o Linux usa tabelas "permanentes" que não são descartadas, mas não precisa de um `pv_entry` para cada pte potencialmente mapeado. O FreeBSD usa tabelas de páginas "throw away", mas adiciona em uma estrutura `pv_entry` para cada pte realmente mapeado. Eu acho que a utilização da memória acaba

sendo a mesma, dando ao FreeBSD uma vantagem algorítmica com sua capacidade de jogar fora tabelas de páginas a vontade com uma sobrecarga muito baixa.

9.5. Finalmente, na seção de page coloring, pode ser útil descrever um pouco mais o que você quer dizer aqui. Eu não segui bem isso.

Você sabe como funciona um cache de memória de hardware L1? Vou explicar: Considere uma máquina com 16MB de memória principal, mas apenas 128K de cache L1. Geralmente, a maneira como este cache funciona é que cada bloco de 128K de memória principal usa o *mesmo* 128K de cache. Se você acessar o offset 0 na memória principal e depois deslocar 128K na memória principal, você pode acabar jogando fora os dados em cache que você leu do offset 0!

Agora estou simplificando muito as coisas. O que acabei de descrever é o que é chamado de cache de memória de hardware "diretamente mapeado". A maioria dos caches modernos são chamados de definição de associações de 2 vias ou definição de associações de 4 vias. A definição de associacões permite acessar até N regiões de memória diferentes que se sobrepõem à mesma memória de cache sem destruir os dados armazenados em cache anteriormente. Mas apenas N.

Então, se eu tenho um cache associativo de 4-way, eu posso acessar o offset 0, offset 128K, 256K e offset 384K e ainda ser capaz de acessar o offset 0 novamente e tê-lo vindo do cache L1. Se eu, então, acessar o deslocamento 512K, no entanto, um dos quatro objetos de dados armazenados anteriormente em cache será descartado pelo cache.

É extremamente importante... *extremamente* importante para que a maioria dos acessos de memória de um processador possam vir do cache L1, porque o cache L1 opera na frequência do processador. No momento em que você tem uma falha de cache L1 e precisa ir para o cache L2 ou para a memória principal, o processador irá parar e potencialmente sentar-se por *centenas* de instruções aguardando uma leitura de memória principal para completar. A memória principal (o ram dinâmico que você coloca em um computador) é *lenta*, quando comparada à velocidade de um núcleo de processador moderno.

Ok, agora em page coloring: Todos os caches de memória modernos são conhecidos como caches *físicos*. Eles armazenam em cache endereços de memória física, não endereços de memória virtual. Isto permite que o cache seja deixado sozinho em uma opção de contexto de processo, o que é muito importante.

Mas no mundo UNIX® você está lidando com espaços de endereço virtual, não com espaços de endereço físico. Qualquer programa que você escreva verá o espaço de endereço virtual dado a ele. As páginas reais *físicas* subjacentes a este espaço de endereço virtual não são necessariamente contíguas fisicamente! De fato, você pode ter duas páginas que estão lado a lado em um espaço de endereço de processos que termina no offset 0 e desloca 128K na memória *física*.

Um programa normalmente pressupõe que duas páginas lado a lado serão armazenadas em cache de maneira ideal. Ou seja, você pode acessar objetos de dados em ambas as páginas sem que elas descartem a entrada de cache uma da outra. Mas isso só é verdadeiro se as páginas físicas subjacentes ao espaço de endereço virtual forem contíguas (no que se refere ao cache).

É isso que o disfarce de página faz. Em vez de atribuir páginas físicas *aleatórias* a endereços virtuais, o que pode resultar em desempenho de cache não ideal, o disfarce de página atribui páginas físicas *razoavelmente contíguas* a endereços virtuais. Assim, os programas podem ser escritos sob a suposição de que as características do cache de hardware subjacente são as mesmas para seu espaço de endereço virtual, como seriam se o programa tivesse sido executado diretamente em um espaço de endereço físico.

Note que eu digo "razoavelmente" contíguo ao invés de simplesmente "contíguo". Do ponto de vista de um cache mapeado direto de 128K, o endereço físico 0 é o mesmo que o endereço físico 128K. Assim, duas páginas lado a lado em seu espaço de endereço virtual podem acabar sendo compensadas em 128K e compensadas em 132K na memória física, mas também podem ser facilmente compensadas em 128K e compensadas em 4K na memória física e ainda manter as mesmas características de desempenho de cache. Portanto, disfarce de página *não* tem que atribuir páginas verdadeiramente contíguas de memória física a páginas contíguas de memória virtual, basta certificar-se de atribuir páginas contíguas do ponto de vista do desempenho e da operação do cache.