Faculdade de Tecnologia de Americana
Curso de Processamento de dados

Design do sistema de arquivos UNIX UFS

VINÍCIUS COQUE

Americana, SP
2010

Faculdade de Tecnologia de Americana
Curso de Processamento de dados

Design do sistema de arquivos UNIX UFS

VINÍCIUS COQUE

vcoque@gmail.com

Trabalho de graduação apresentado à Faculdade de Tecnologia de Americana, como parte dos requisitos para obtenção do título de Tecnólogo em Processamento de Dados, sob orientação do Prof. Antônio Alfredo Lacerda.

Americana, SP
2010

BANCA EXAMINADORA

Prof. Antonio Alfredo Lacerda (Orientador)

Prof. Irineu Ambrozano Filho

Prof. Benedito Aparecido Cruz

AGRADECIMENTOS

Agradeço a minha família, por sempre me apoiar e ajudar a superar mais um grande desafio em minha vida. Aos meus amigos, que me deram apoio e sempre foram compreensivos nos vários momentos em que estive ausente durante a elaboração deste trabalho, e aos professores da FATEC Americana, que sempre nos auxiliaram durante o curso colaborando para a formação profissional de todos.

DEDICATÓRIA

Aos meus pais e ao meu irmão, por estarem ao meu lado para me apoiar em todos os momentos da minha vida e pelos momentos agradáveis que passamos juntos.

RESUMO

Este trabalho tem o objetivo de descrever a arquitetura do sistema de arquivos UNIX FFS/UFS, desenvolvida pela Universidade da Califórnia em Berkeley. Serão apresentados os motivos que levaram ao desenvolvimento de um novo sistema de arquivos, quais as diferenças entre o antigo e o novo sistema de arquivos e a estrutura de partes importantes para o armazenamento e recuperação de dados do disco. Inicialmente será apresentada a historia do sistema operacional UNIX, que provocou uma grande revolução, até o desenvolvimento do popular Berkeley Software Distribution (BSD) e as melhorias trazidas por ele.

Palavras Chave: BSD, sistema de arquivos, UNIX.

ABSTRACT

This paper aims to describe the architecture of the UNIX file system FFS / UFS, developed by the University of California at Berkeley. Will present the reasons that led to the development of a new file system, the differences between the old and the new file system and structure of important parts for storing and retrieving data from disk. Initially it will be presented the history of the UNIX operating system, which caused a great revolution, until the development of the popular Berkeley Software Distribution (BSD) and the improvements brought by him.

Keywords: BSD, filesystem, UNIX.

SUMÁRIO

LISTA DE FIGURAS E DE TABELAS

LISTA DE SIGLAS

MIT……. Massachusets Institute of Technology

BSD…….. Berkeley Software Distribution

FFS……..… Fast File System

UFS……. Unix File System

Inode..... Index Node

Vfs…… Virtual File System

Vnode…….. Virtual Node

INTRODUÇÃO

Antigamente os sistemas computacionais eram muito complexos sua intercomunicação não era realizada de maneira simples e eficiente. Na tentativa de desenvolver um ambiente computacional conveniente e multitarefa, a Bell Telephone Laboratorie, junto com a General Eletric Company e o MIT, deram inicio ao projeto do sistema operacional MULTICS. Devido a muitas incertezas com relação ao projeto, a Bell abandonou o projeto. Com o fracasso do MULTICS e sem um ambiente computacional que atendesse suas necessidades, Ken Thompson e Dennis Ritchie iniciaram, no Bell Laboratories, o desenvolvimento de um novo sistema operacional, que viria a se chamar UNIX.

A distribuição do código fonte do UNIX para as universidades o tornou bastante popular rapidamente e permitiu que outras versões fossem desenvolvidas, sendo a mais famosa delas, desenvolvida na Universidade da Califórnia em Berkeley, o Berkeley Software Distribution (BSD), que anos depois viria a se tornar o FreeBSD.

Insatisfeitos com o desempenho do sistema de arquivos desenvolvido para o UNIX, a equipe de Berkeley deu inicio ao desenvolvimento de um novo sistema de arquivos. Durante o desenvolvimento do novo sistema de arquivos foram feitas várias mudanças para melhorar o desempenho, como aumentar o tamanho dos blocos de dados para que mais dados fossem transferidos em uma única transação, a divisão do disco em grupos de cilindros e alocar os inodes mais próximos de seus blocos de dados. Com as mudanças nas políticas de alocação, algumas estratégias foram necessárias, como a fragmentação de blocos, para aproveitar espaços deixados por pedaços de arquivos que não ocupam um bloco inteiro.

Durante a evolução do sistema de arquivos, houve necessidade de implementar outras mudanças, como a adoção do vnode/vfs, desenvolvido pela Sun Microsystems. Com o aparecimento de novos sistemas de arquivos, surgiu a necessidade de adicionar suporte a estes sistemas no FreeBSD. O vnode/vfs é uma interface que trabalha entre o inode e o arquivo, permitindo que o kernel trabalhe

com sistemas de arquivos diferentes, sem ter de lidar com as diferenças de estrutura dos inodes.

Nos próximos capítulos este documento apresenta as novas implementações feitas durante o desenvolvimento do novo sistema de arquivos pela equipe de Berkeley e outros desenvolvedores do projeto, com o objetivo de obter um sistema de arquivos com bom desempenho e confiabilidade.

1 Histórico do UNIX

> “O UNIX é basicamente um sistema operacional simples, mas é preciso ser um gênio para entender a simplicidade.”
>
> (Dennis Ritchie)

No inicio os sistemas computacionais existentes não se comunicavam uns com os outros. Mesmo os sistemas desenvolvidos pela mesma companhia precisavam de interpretadores e não existia interoperabilidade entre sistemas de diferentes fabricantes. Os sistemas operacionais eram capazes de executar tarefas muito limitadas e apenas nas máquinas nas quais eles foram escritos. Se uma empresa adquirisse um novo e mais poderoso computador, o antigo sistema provavelmente não iria funcionar neste novo computador e, posteriormente, todos os dados da companhia deveriam ser cadastrados novamente na nova máquina.

Na tentativa de desenvolver um ambiente computacional conveniente, interativo e fosse multiusuário, em 1965, a Bell Telephone Laboratories (Bell Labs) juntou esforços com a General Eletric Company e o projeto MAC do Massachusets Institute of Technology (MIT) para desenvolver um novo sistema operacional chamado MULTICS (*MULTIplexed Information and Computing Service*). Baseado uma um sistema operacional desenvolvido em uma pesquisa do MIT chamado *Compatible Time Sharing System* (CTSS), o MULTICS resultou em uma série de novas abordagens. Até 1969 já havia uma versão primitiva do Multics rodando em um computador GE 645, porem ele não atendia os serviços computacionais pretendidos, nem estava claro quanto tempo levaria para atingir seus objetivos. Devido ao alto custo de desenvolvimento e ao tempo necessário para completar o projeto a Bell Labs encerrou sua participação no projeto.

Com a saída da Bell Labs do projeto, os membros do *Computing Science Research Center* (Centro de Pesquisas em Ciências da Computação) ficaram sem um serviço computacional conveniente. Um dos pesquisadores, Ken Thompson, propôs a Bell Labs que comprassem um novo computador e dessem inicio ao desenvolvimento de seu próprio sistema, sua proposta foi recusada, Porém não levou muito tempo para que Thompson encontrasse um PDP-7 usado e começasse a trabalhar nele. Na tentativa de melhorar seu ambiente de programação, Dennis

Ritchie, Ken Thompson e outros esboçaram o projeto de um sistema de arquivos, que depois viria a evoluir para uma primeira versão do sistema de arquivos UNIX. Thompson escreveu programas que simulavam o comportamento do sistema de arquivos, desenvolveu alguns utilitários (ls, cp, cat, rm) e um interpretador de comandos, ou shell..

Em 1970 Brian Kernighan sugeriu o nome UNIX para o novo sistema operacional como um trocadilho com o nome MULTICS. Em áreas onde o MULTICS tentava fazer várias tarefas, o Unix fazia apenas uma, porém bem feita.

A primeira versão do UNIX foi escrita utilizando linguagem Assembly, mas a intenção de Thompson era que o sistema operacional fosse escrito em uma linguagem de alto nível, então ele escreveu uma linguagem chamada B. B era uma linguagem interpretada com os problemas de performance característico destas linguagens, então Ritchie evoluiu B para uma linguagem chamada C, permitindo criação de código de máquina e declaração de tipos e estruturas. Em 1973 o sistema operacional foi reescrito em linguagem C.

Naquele momento a AT&T, proprietária da Bell Telephone Laboratories, não podia comercializar computadores devido a um acordo assinado com o governo dos Estados Unidos da América, mas ela fornecia cópias do UNIX para universidades que tivessem interesse em utilizá-lo para fins educacionais e a popularidade do sistema começou a crescer. Somente em 1977 a AT&T começou a comercializar as licenças de uso do UNIX, o fato do sistema ser distribuído com seu código fonte, permitiu as organizações, além de utilizar o sistema, implementar suas próprias mudanças internas. Com o aumento da popularidade dos microprocessadores, não demorou muito para outras companhias portassem seus UNIX para estes novos computadores, o que resultou em algumas variantes do sistema básico. No período de 1977 e 1982 a Bell Labs combinou algumas variantes existentes na AT&T em um único sistema básico, chamado de UNIX System III e mais tarde, adicionando novas funcionalidades, evoluíram para o UNIX System V.

O grupo de desenvolvimento mais influente, fora do Bell Labs e AT&T, foi o da Universidade da Califórnia em Berkeley. Seguindo a compra de um Vax 11/70 em

1975, Ken Thompson iniciou um período sabático em Berkeley, periodo no qual ele trouxe à tona o Unix 6th Edition. Nesta mesma época, os estudantes de graduação Bill Joy e Chuck Haley chegaram na universidade e começaram a trabalhar no novo sistema instalado e em 1977 Bill Joy distribuiu 30 cópias do Berkeley Software Distribution (chamado de 1BSD).

Em 1979 o Berkeley distribution estava na sua terceira versão, 3BSD. Nesta época a DARPA (*Defense Advanced Research Projects Agency*) decidiu utilizar o UNIX como padrão para prover uma rede para ligar seus principais centros de pesquisa. Com nase na proposta do professor Bob Fabry a DARPA, Berkeley ganhou um contrato de dezoito meses, Fabry criou o *Computer Systems Research Group* (CSRG) e Bill Joy trabalhou no que viria a ser o 4BSD. Lançado em 1980 o sistema incluia um compilador Pascal, controle de jobs, auto reboot e um sistema de arquivos de 1KB.

Seguindo a renovação de contrato com a DARPA, o novo projeto produziu suporte a grandes espaços de endereçamento virtual, melhores mecanismos IPC, o que viria a se tornar o *Berkeley Fast File System* e a pilha TCP/IP foi integrada ao BSD. Após Bill Joy deixar a universidade em 1982 para fundar a Sun Microsystems, o 4.2BSD foi lançado em 1983. Devido a introdução do TCP/IP e do Fast File System, o número de instalações do 4.2BSD ultrapassou o System V da AT&T.

Para evitar que os interessados em ter acesso ao código fonte dos componentes de rede do BSD, tivessem de comprar uma licença da AT&T, a *Networking Release* do BSD foi lançada em 1989. Uma versão expandida, que envolveu reescrever todos, exceto seis, arquivos do kernel foi distribuída como *Networking Release 2* em 1991. Isto envolveu um grande esforço de muitas pessoas. Bill Jolitz continuou o trabalho de reescrever os seis arquivos para evitar problemas com códigos licenciados pela AT&T e portar o sistema para o Intel 386, resultando no 386/BSD, que foi distribuído pela internet.

Seguindo o trabalho de Jolitz no 386/BSD, Jordan Hubbard, Rod Grimes e Nate Williams lançaram o *Unofficial 386BSD Patchkit*. Após discussões Hubbard e Walnut Creek criaram um novo sistema, que chamara de FreeBSD, sua primeira

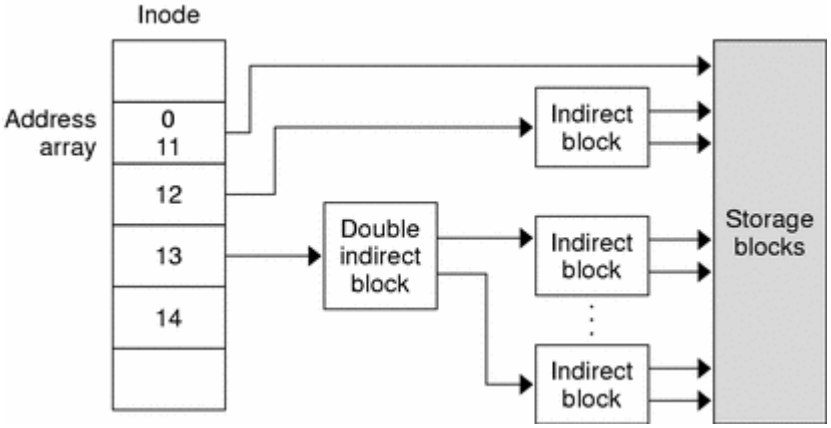
versão foi lançada em dezembro de 1993. Melhorias continuam sendo implementadas no FreeBSD, neste momento sua última versão é FreeBSD 8.0, lançada em novembro de 2009.

2 O sistema de arquivos UNIX

No sistema de arquivos desenvolvido no Bell Laboratories, chamado apenas de FS, cada disco é dividido em uma ou mais partições, cada uma destas partições do disco podem conter um sistema de arquivos. Um sistema de arquivos é descrito pelo superbloco, que contém seus parâmetros básicos, que inclui o número de blocos, o número máximo de arquivos e um ponteiro para a *free list*, uma lista encadeada de todos os blocos livres no sistema de arquivos.

Cada arquivo no sistema de arquivos contém um descritor associado, chamado *inode*, contendo informações sobre seu dono, data de modificação e do último acesso e um vetor de índices que aponta para os blocos de dados do arquivo. Um inode também pode conter referencias para blocos indiretos contendo mais dados, como mostrado abaixo na figura 1.

Figura 1: Ponteiros para blocos indiretos



Fonte: http://docs.sun.com/source/817-5093/images/fs_filesysappx.fig3184.gif

Em um sistema de arquivos UNIX tradicional, os inodes ficam separados dos dados, o que pode resultar em grandes tempos de busca. Por exemplo, um sistema de arquivos com 150MB tem 4MB reservado para seus inodes e 146MB de dados, isto separava os inodes dos blocos de dados e arquivos em um mesmo diretório não eram sempre alocados em blocos sequenciais de inodes. Esta organização faz com que ao acessar arquivos exista um grande intervalo entre a leitura do inode e a leitura do bloco de dados.

O antigo sistema de arquivos transferia não mais do que 512 bytes em cada transação e frequentemente descobria que o próximo bloco de dados não estava no mesmo cilindro, forçando buscas entre transações de 512 bytes. A combinação do tamanho de blocos pequeno com sua localização dispersa limitava severamente o throughput do sistema de arquivos.

Aumentando o tamanho do bloco de 512 bytes para 1024 bytes duplicou a performance do sistema de arquivos. O aumento acontece porque cada transferência acessa duas vezes mais dados e a maior parte dos arquivos podem ser descritos sem a necessidade de acessar blocos indiretos, já que os blocos contêm duas vezes mais dados. Isto indicou que aumentar o tamanho dos blocos era uma boa saída para melhorar o throughput. Embora tenha dobrado seu desempenho, o antigo sistema de arquivos ainda utilizava apenas 4% da capacidade de transferência do disco. O problema principal era que a free-list, embora ordenada para otimizar o acesso, ela se tornava embaralhada conforme arquivos era criados e excluídos. Algumas vezes ela se tornava totalmente aleatória, forçando uma busca a cada bloco acessado. Não havia outra maneira de restaurar o desempenho do antigo sistema de arquivos senão reconstruí-lo.

2.1 Melhorias e organização do Berkeley Fast File System

A primeira versão do atual sistema de arquivos BSD apareceu no 4.2BSD (McKusick et al., 1984), ela ainda está em uso até hoje como o nome de UFS2. Na organização do novo sistema de arquivos o superbloco é localizado no inicio da partição do disco. Como o superbloco contém dados críticos, ele é replicado quando o sistema de arquivos é criado, porém, como seus dados não se alteram suas cópias só precisam ser acessadas em caso de falha no disco, tornado assim o superbloco inutilizado.

Para suportar fragmentos tão pequenos quanto um único setor de 512 bytes, o tamanho mínimo de um bloco é de 4096 bytes. O tamanho dos blocos pode ser qualquer potência de dois, maior ou igual a 4096, e é gravado no superbloco, permitindo que sistema de arquivos com diferentes tamanhos de blocos sejam acessados no mesmo sistema.

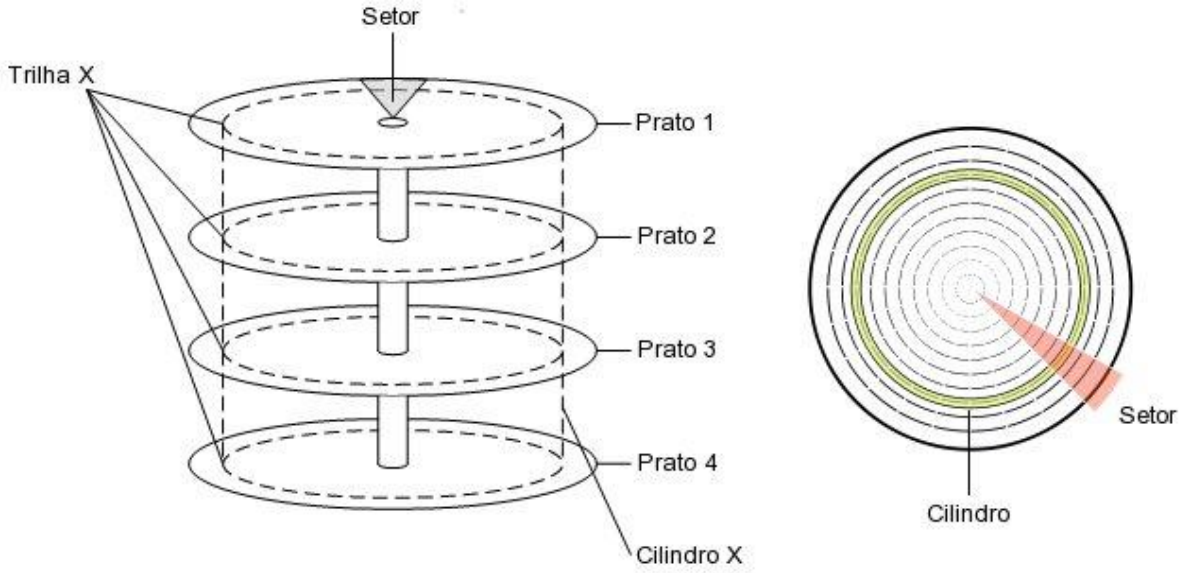
Na nova organização do sistema uma partição do disco é dividida em mais áreas chamadas grupos de cilindros. Historicamente grupos de cilindros eram compostos por um ou mais cilindros, embora o FreeBSD ainda use a mesma estrutura de dados para descrever grupos de cilindros, sua definição foi alterada. Quando o sistema foi projetado ele podia ter uma visão confiável da geometria do disco. Os discos mais novos fornecem um número fictício de blocos por trilha, trilhas por cilindro e cilindro por disco, sistemas que utilizam RAID, por exemplo, o “disco” utilizado pelo sistema pode ser formado, na verdade, por vários discos (McKusick, 2004). Discos mais modernos contêm um número maior de setores por trilha na parte externa que na parte interna do disco, o que faz com que a posição rotacional de um setor seja muito complexa para ser calculada. Segundo McKusick (2004), quando o UFS2 foi desenhado, decidiram se livrar do código de layout rotacional encontrado no UFS1 e assumiram que colocar os arquivos em blocos numericamente mais próximos daria o melhor desempenho. Assim os grupos de cilindros ainda existem no UFS2, mas apenas como uma forma conveniente de gerenciar grupos de blocos próximos.

Cada grupo de cilindros deve caber em um único bloco do sistema de arquivos. Quando um novo sistema de arquivos é criado o utilitário newfs calcula o número máximo de blocos que podem ser empacotados em um grupo de cilindros baseado no tamanho dos blocos. Então o número mínimo de grupos de cilindros necessários para descrever o sistema de arquivos é alocado. Um sistema de arquivos com blocos de 16KB, normalmente, tem seis grupos de cilindros por gigabyte.

Associado a cada grupo de cilindros está uma cópia redundante do superbloco, espaço para inodes um mapa de bits com os blocos disponíveis no grupo e um sumário de informações de uso dos blocos. O mapa de bits dos blocos disponíveis substitui a *free-list* do antigo sistema de arquivos. Cada grupo de cilindros contém um número fixo de inodes, que é definido quando o sistema de arquivos é criado. A política é alocar um inode para cada 2048 bytes, para que sempre existam mais inodes do que o necessário. Para garantir um nível de integridade os meta-dados de cada grupo de cilindros não são gravados no mesmo prato do disco, pois assim uma falha física no disco faria com que as cópias do

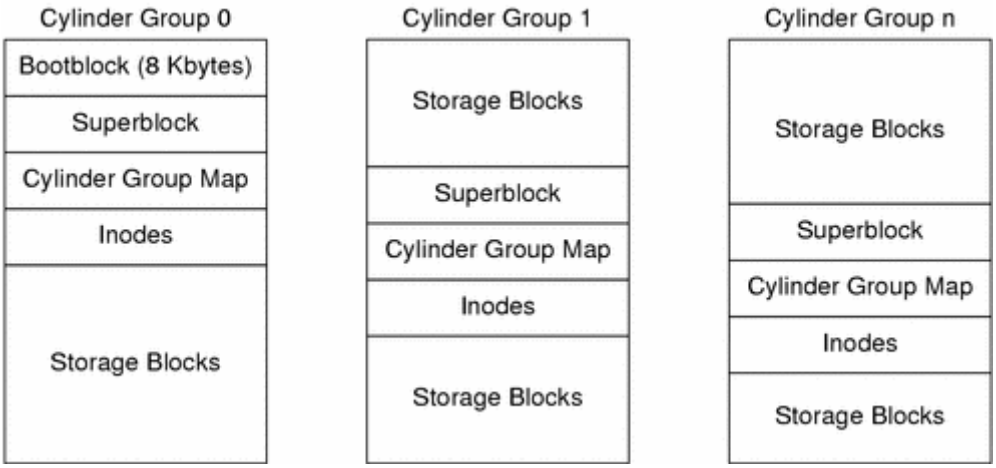
superbloco fossem destruídas. Para evitar isto, os meta-dados são deslocados uma trilha a mais a partir do início do grupo de cilindros em relação ao anterior. A seguir na figura 2 vemos como são formados os grupos de cilindros e na figura 3, a organização dos blocos de cada grupo de cilindros.

Figura 2: Grupos de cilindros em um disco



Fonte: https://cs.senecac.on.ca/~selmys/subjects/ops335-073/cylinder.png

Figura 3: Organização dos blocos de cada grupo de cilindros



Fonte: http://docs.sun.com/source/817-5093/images/fs_filesysappx.fig3185.gif

A razão para criar grupos de cilindros é para agrupar os inodes que são espalhados pelo disco próximos dos blocos referenciados por eles, ao invés de colocá-los no

inicio do sistema de arquivos. Isto evita longas buscas pelo disco e também diminui as chances de perda de todos os inodes em caso de falha no disco.

2.2 Utilização do disco

Um arquivo, no novo sistema de arquivos, é composto de blocos de 8192 bytes de dados, comparado aos 1024 bytes do 3BSD, é possível transferir oito vezes mais dados por transação. O problema com blocos grandes é que os sistemas de arquivos contêm arquivos pequenos, menores que 8KB, fazendo com que sobre espaço não utilizado nos blocos. Na tabela 1 vemos a percentagem de perda de espaços em disco, quanto maior o tamanho dos blocos, o espaço reservado para inodes diminui, mas o espaço não utilizado nos blocos cresce rapidamente para intoleráveis 29,4% de perda.

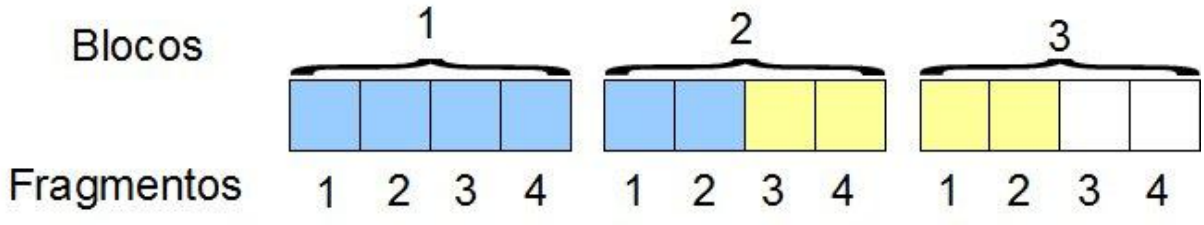
Tabela 1: Espaço não utilizado em função do tamanho do bloco

Total (%)	Dados (%)	Inodes (%)	Organização
0,0	0,0	0,0	Dados apenas, sem separação entre arquivos
1,1	1,1	0,0	Dados apenas, arquivos em limite de 512 bytes
7,4	1,1	6,3	Dados + inodes, blocos 512 bytes
8,8	2,5	6,3	Dados + inodes, blocos 1024 bytes
11,7	5,4	6,3	Dados + inodes, blocos 2048 bytes
15,4	12,3	3,1	Dados + inodes, blocos 4096 bytes
29,4	27,8	1,6	Dados + inodes, blocos 8192 bytes

Fonte: McKusick (1984)

Para não haver perdas significantes de espaço utilizando tamanhos maiores de blocos, arquivos pequenos precisam ser armazenados de maneira mais eficiente, para isso é possível fazer a divisão de um bloco em fragmentos. Cada bloco pode ser quebrado em dois, quatro ou oito fragmentos, cada um deles é endereçável. O menor tamanho de um fragmento é limitado ao tamanho do setor do disco, normalmente 512 bytes. O mapa de blocos associado com cada grupo de cilindros registra o espaço disponível no nível de fragmentação, para determinar quando um bloco está disponível o sistema examina os blocos alinhados.

Figura 4: Blocos de disco fragmentados



Em um sistema de arquivos com blocos de 4096 bytes e tamanho de fragmentos de 1024 bytes, um arquivo pode ser representado por zero, ou mais, blocos de dados e, possivelmente, um bloco fragmentado. Se um bloco precisa ser fragmentado para alocar dados pequenos, os fragmentos restantes são disponibilizados para alocar outros arquivos. Como exemplo, considere um arquivo de 11000 bytes armazenado em um sistema de arquivos de 4096/1024, o arquivo irá utilizar dois blocos inteiros e mais três fragmentos de outro bloco. Caso nenhum bloco com três fragmentos alinhados esteja disponível, um bloco inteiro é dividido produzindo os fragmentos necessários e um único fragmento não utilizado. Este fragmento que sobrou pode ser alocado para outros arquivos quando necessário. A figura 4 mostra um exemplo de fragmentação de blocos, onde um arquivo utilizou um bloco inteiro e mais dois fragmentos do um segundo bloco. Os blocos 2 e 3 contém fragmentos de outro arquivo, porém os fragmentos número 3 e 4 continuam disponíveis para alocação.

2.3 Políticas de layout e alocação de blocos

As políticas de layout são divididas em duas partes, em um nível mais alto estão as políticas globais, que utilizam um sumário de informações para tomar decisões quanto a localização de blocos e inodes. Estas políticas também decidem quando forçar uma busca longa de um novo grupo de cilindros porque não há espaço suficiente no atual. Abaixo das políticas globais estão as políticas de alocação local, que utilizam um esquema otimizado para distribuir os blocos de dados.

Há dois métodos para aumentar a performance de um sistema de arquivos, um é aumentar a localidade de referência para diminuir a latência das buscas, outra

é melhorar o layout dos dados para transferir a maior quantidade de dados possível. As políticas globais tentam melhorar o desempenho agrupando dados relacionados, elas não podem concentrar todas as referências a dados, mas precisam espalhar dados não relacionados por diferentes grupos de cilindros.

Inodes de arquivos em um mesmo diretório são frequentemente acessados juntos, a política de layout dos inodes tenta colocar os inodes dos arquivos em um diretório em um mesmo grupo de cilindros. Para garantir que os arquivos serão distribuídos por todo o sistema de arquivos, quando um diretório é criado na raiz ele é colocado em um cilindro com um número maior que a média de blocos livres e inodes próximo do diretório pai. A intenção deste método é que o agrupamento de inodes aconteça a maior parte das vezes e diminuir a distância percorrida no disco por aplicações que navegam pelos diretórios. A alocação de inodes é feita utilizando a estratégia do “próximo livre”, embora isso faça com que os blocos sejam alocados aleatoriamente, todos os inodes podem ser acessados utilizando de 10 a 20 transferências do disco. Em contraste o antigo sistema de arquivos precisava de uma transferência para acessar o inode para cada arquivo em um diretório.

Os blocos de dados de um arquivo são acessados juntos, portanto as políticas tentam alocar os blocos em um mesmo grupo de cilindros. O problema desta política é que arquivos grandes irão encher um grupo de cilindros rapidamente, forçando que dados sejam espalhados em outros cilindros. O ideal é que nenhum grupo de cilindros se torne completamente cheio, então a alocação de blocos é direcionada para outro grupo quando um arquivo ocupa aproximadamente 25% dos blocos existentes neste grupo. Embora arquivos grandes tendam a se espalhar pelo disco vários megabytes de dados estão disponíveis antes que uma nova busca seja necessária.

As rotinas de políticas globais chamam as políticas de alocação local requisitando um bloco específico. As rotinas de alocação irão alocar o bloco requisitado caso ele esteja livre, senão a seguinte estratégia de alocação será utilizada:

1. Usar o próximo bloco, mais próximo do bloco requisitado, dentro do mesmo grupo de cilindros;

2. Se o grupo de cilindros estiver cheio, aplica o hash quadrático no número do grupo para encontrar outro grupo de cilindros onde procurar por um bloco livre;

3. Se o hash quadrático falhar, é feita uma busca extensiva em todos os grupos de cilindros.

Na primeira verão do FFS existiam 4 passos para encontrar um bloco disponível, que consistia em procurar o bloco rotacionalmente mais próximo (McKusick, 1984), porém ele não existe mais nas versões mais recentes do sistema de arquivos.

Se um arquivo está sendo escrito no disco e o ponteiro do bloco é zero ou aponta para um fragmento muito pequeno para receber os dados adicionais, as rotinas de alocação irão obter um novo bloco. Se o arquivo precisa ser estendido uma das duas condições precisam existir:

1. O arquivo não contém blocos fragmentados e seu último bloco não contém espaço suficiente para armazenar os novos dados. Se existir espaço em um bloco já alocado, então o espaço será preenchido. Se o restante dos novos dados consistirem de mais de um bloco inteiro, o primeiro bloco completo de dados é escrito. Este processo é repetido até que menos que um bloco completo de dados sobre. Se os dados restantes se encaixarem em menos que um bloco completo, o número necessário de fragmentos é alocado. Para evitar problemas de desempenho, o sistema de arquivos permite que apenas blocos indiretos referenciem fragmentos.

2. O arquivo contém um ou mais fragmentos e eles não contém espaço suficiente para armazenar os novos dados. Se o tamanho dos novos dados, mais o tamanho dos dados já existentes nos fragmentos,

excederem o tamanho de um bloco, então um novo será alocado. O conteúdo dos fragmentos são copiados para o inicio do bloco, o restante do bloco é preenchido com os novos dados. O processo então continua como no primeiro passo. Caso houver espaço suficiente em um bloco, um novo conjunto de fragmentos grande o bastante para armazenar os dados é alocado.

Por causa da restrição que apenas o ultimo bloco de um arquivo pode ser fragmentado, é necessário garantir que todos os fragmentos anteriores foram atualizados para um bloco completo. Ao finalizar com sucesso a alocação, o número de blocos ou fragmentos utilizados é retornado e ponteiro para o bloco no inode é atualizado. Tendo o bloco alocado no disco, o sistema está pronto para alocar um buffer para receber o conteúdo deste bloco.

3 Virtual-Filesystem Interface

Nos sistemas UNIX anteriores, as entradas de arquivo referenciavam diretamente os inodes, isto funcionou bem quando havia apenas um sistema de arquivos implementado. Entretanto, com o surgimento de vários sistemas de arquivos, a arquitetura teve de ser generalizada, a nova arquitetura tinha de permitir o acesso a sistemas de arquivos de outras máquinas, incluindo máquinas que estavam executando diferentes sistemas operacionais.

A saída mais simples e lógica para este problema era adiciona uma camada, orientada a objetos, abaixo dos arquivos e acima dos inodes. Esta camada foi implementada pela Sun Microsystems, que a chamou de *virtual-node*, ou *vnode*. Interfaces de sistema que referenciavam inodes foram alteradas para referenciar os vnodes. Um vnode utilizado por um sistema de arquivos local deverá referenciar um inode, Um vnode utilizado por um sistema de arquivos remoto deverá referenciar um protocolo de controle que descreve a localização do arquivo e informações necessárias para acesso remoto ao arquivo.

Kleiman (1986), em sua publicação sobre os vnodes, descreve quatro objetivos para projeto:

1. Separar as implementações de funcionalidades independentes do sistema de arquivos e as dependentes do sistema de arquivos e prover uma interface melhor definida entre os dois;

2. A interface deverá suportar acesso a sistemas de arquivos locais, como o FFS do 4.2BSD, sistemas de arquivos remotos, como NFS da Sun ou o RFS da AT&T e sistemas de arquivos não-UNIX, como o MS-DOS

3. Deve suportar ser utilizada pelo servidor de sistema de arquivos remotos, para satisfazer requisições de clientes;

4. Operações do sistema de arquivos através da interface deverão ser atômicas, de modo que várias operações não precisem ser englobadas pelos locks do sistema de arquivos.

O vnode foi projetado como uma interface orientada a objetos, assim o kernel o manipula enviando requisições ao objeto oculto através de um conjunto de operações definidas. Devido a variedade de sistemas de arquivos suportados no FreeBSD, este conjunto de operações é grande e extensível. Ao contrário da implementação original dos vnodes, feita pela Sun Microsystems, a utilizada no FreeBSD permite que operações de vnodes sejam adicionadas ao sistema dinâmicamente, durante a inicialização do sistema ou quando um sistema de arquivos é carregado dinamicamente no kernel.

3.1 Tradução de nomes de caminho

A tradução de nomes exige uma série de interações entre o vnode e o sistema de arquivos. A tradução de nomes segue da seguinte maneira.

1. O nome a ser traduzido é copiado do processo do usuário ou do buffer de rede;

2. O ponto inicial do nome é determinado com o diretório raiz ou o diretório atual;

3. O vnode chama a operação de busca específica do sistema de arquivos. Esta operação de busca vai retornar um vnode ou um erro, caso o nome não exista;

4. Se houver um erro, a camada superior irá retorná-lo, se o nome foi esgotado, a busca é concluída o vnode retornado é o resultado da busca. Se não houver erros e a busca não estiver concluída, a camada superior verifica se o diretório é um ponto de montagem para outro sistema de arquivos. Se for um ponto de montagem, o diretório de busca torna-se o sistemas de arquivos montado, senão o diretório de busca torna-se o vnode retornado. A busca então retorna ao passo número 3.

Embora pareça possa parecer ineficaz a chamada através da interface vnode para cada componente do caminho, fazer isso normalmente é necessário. A razão para isso é que o sistema de arquivos não sabe quais diretórios são utilizados como pontos de montagem. Desde que irá redirecionar a busca para um novo sistema de arquivos, é importante que o sistema de arquivos atual não vá além do diretório montado. Embora possa ser possível para um sistema de arquivos local saber quais diretórios são pontos de montagem, é quase impossível para um servidor saber quais diretórios são utilizados como pontos de montagem por seus clientes. Consequentemente, abordagem conservadora de passar somente um componente do nome do caminho por busca é utilizado.

4 Gerenciamento de buffer

Historicamente os sistemas UNIX dividiam a memória em dois pools principais. O primeiro era a memória virtual, utilizada para fazer cache das páginas de processos. O segundo era o buffer pool e era utilizado para fazer cache dos dados dos sistemas de arquivos. A memória era dividida entre estes dois pools durante o boot do sistema e não havia migração de memória entre estes dois pools, uma vez que eles eram criados. Com a adição da chamada de sistema mmap, o kernel suportava o mapeamento de arquivos no espaço de endereçamento de processos. Deste modo o FreeBSD juntou o buffer cache e a memória virtual em um único cache.

A memória virtual é dividida em um pool de paginas de conteúdo de arquivos e um pool com páginas anônimas contendo partes de processos que não são apoiados por arquivos, assim como sua pilha. Ao invés de reescrever toda a busca de páginas no pool memória virtual, uma camada de emulação de buffer cache foi escrita. A camada de emulação tem a mesma interface que o antigo buffer cache, mas trabalha buscando as paginas requisitadas no cache de memória virtual. Quando o sistema de arquivos solicita um bloco de um arquivo, a camada de emulação chama o sistema de memória virtual para ver se ele está em memória, se não estiver em memória o sistema tenta obter a leitura. Normalmente as páginas do cache de memória virtual não são mapeadas no espaço de endereçamento do kernel. Entretanto, um sistema de arquivos frequentemente precisa inspecionar os blocos que ele requisitou, se é um arquivo ou meta-dados por exemplo, assim a camada de emulação do buffer cache precisa não apenas encontrar o bloco solicitado mas também alocar algum endereço no espaço do kernel e mapear o bloco nele.

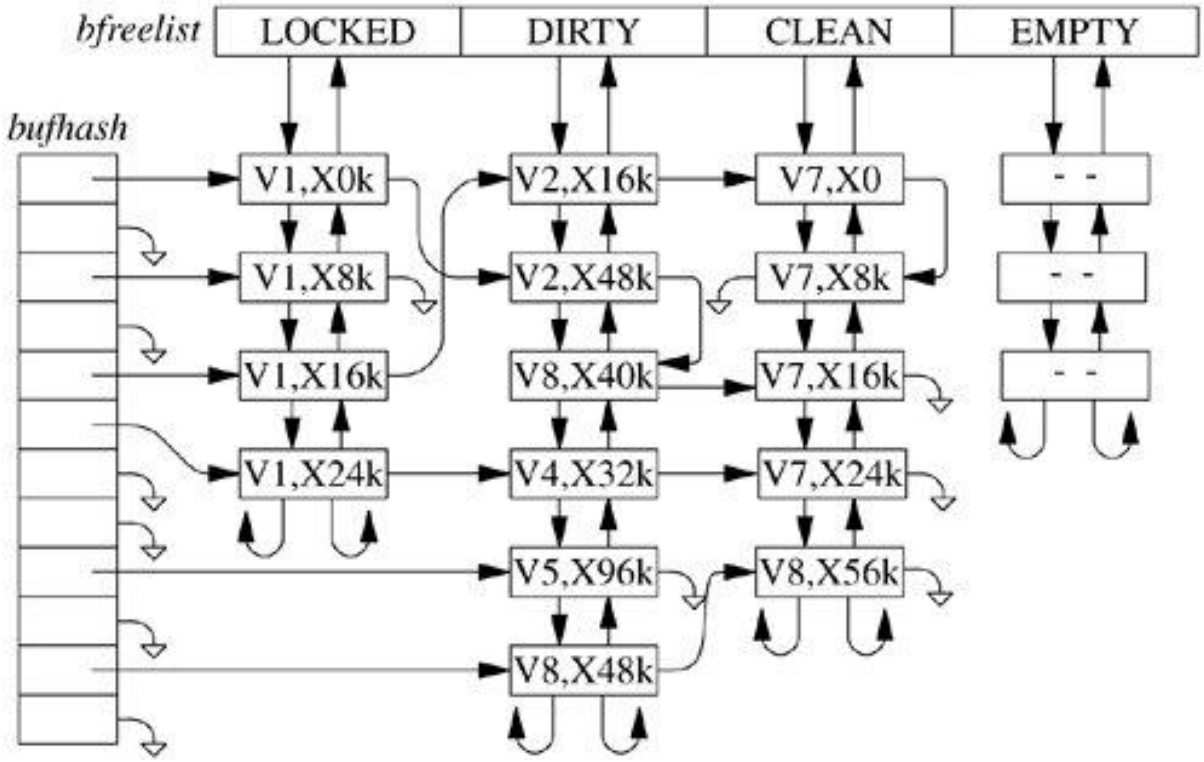
O sistema de memória virtual não tem como descrever blocos identificados como blocos associadas a um disco, uma pequena parte do buffer cache são mantidos para armazenar estes blocos usados pelos meta-dados, como superbloco ou inodes.

Se um buffer é modificado, ele é marcado como *dirty*, estes buffers precisam ser escritos de volta a seus sistemas de arquivos, três rotinas fazem o trabalho de escrever os buffers no disco. Uma vez que o buffer pode ser modificado novamente, ele é marcado como dirty, mas não deve ser escrito imediatamente. Depois de ser marcado como dirty o buffer é enviado para a lista de buffers dirty. Se um buffer for completamente utilizado, provavelmente ele não será modificado em breve, então ele deverá ser liberado pela rotina *bawrite()*. Esta rotina agenda a entrada/saída do buffer, mas permite que quem a chamou continue executando enquanto a saída termina.

O último caso é da chamada de sistema *bwrite()*, que garante que a escrita está completa antes de prosseguir. Como a chamada bwrite() pode inserir uma latência grande no sistema, ela é utilizada somente quando um processo explicitamente solicita este comportamento, quando a operação é crítica para garantir a consistência do sistema de arquivos após uma falha do sistema. Quando a operação de saída termina, todos os threads que estão aguardando são ativados ou, se não houver necessidade imediata, desfazer o buffer.

Alguns buffers, embora limpos, podem ser necessários novamente, para evitar a sobrecarga de criar e desfazer repetidamente um buffer, a camada de emulação do buffer cache provê rotinas que permite que o sistema de arquivos notifique se o buffer será necessário novamente. O buffer é colocado em uma lista de buffers limpos (clean), ao invés de desfazê-los.

Figura 5:Funcionamento do buffer pool. V – vnode, X – pedaço do arquivo



Fonte: McKusick (2004)

Na figura 5 podemos ver a representação interna de um buffer pool. Além de aparecer no hash, cada buffer não travado aparece em exatamente uma free list. A primeira delas é a *LOCKED*. Buffers nesta lista não podem ser retirados do cache. Esta lista originalmente mantinha dados do superbloco, no FreeBSD ela contém buffers que são escritos em segundo plano. Na escrita em segundo plano, o conteúdo de um buffer marcado como dirty são copiados para um buffer anônimo, o buffer anônimo então é escrito no disco. O buffer original pode continuar sendo utilizado enquanto o buffer anônimo é escrito.

A segunda lista é a *DIRTY*, com buffers que foram modificados, mas ainda não foram escritos no disco. Esta lista é gerenciada usando o algoritmo do buffer *menos usado recentemente*. Quando um buffer é encontrado na lista DIRTY, ele é removido e usado, depois é retornado ao fim da lista. Quando muitos buffers são utilizados, o kernel inicia o *buffer daemon*, que escrevem os buffers começando pelo topo da lista, assim buffers escritos repetidamente sempre estarão no fim da lista e não serão reutilizados prematuramente por novos blocos.

A terceira lista é a *CLEAN*, esta lista contém blocos que um sistema de arquivos não está utilizando atualmente, mas pode utilizar em breve. A lista CLEAN é gerenciada utilizando o mesmo algoritmo usado pela lista DIRTY.

A última lista é a de buffers vazios, EMPTY. Os buffers vazios são apenas cabeçalhos e não tem memória associada com eles. Eles são mantidos nesta lista aguardando por requisições de mapeamento de buffer.

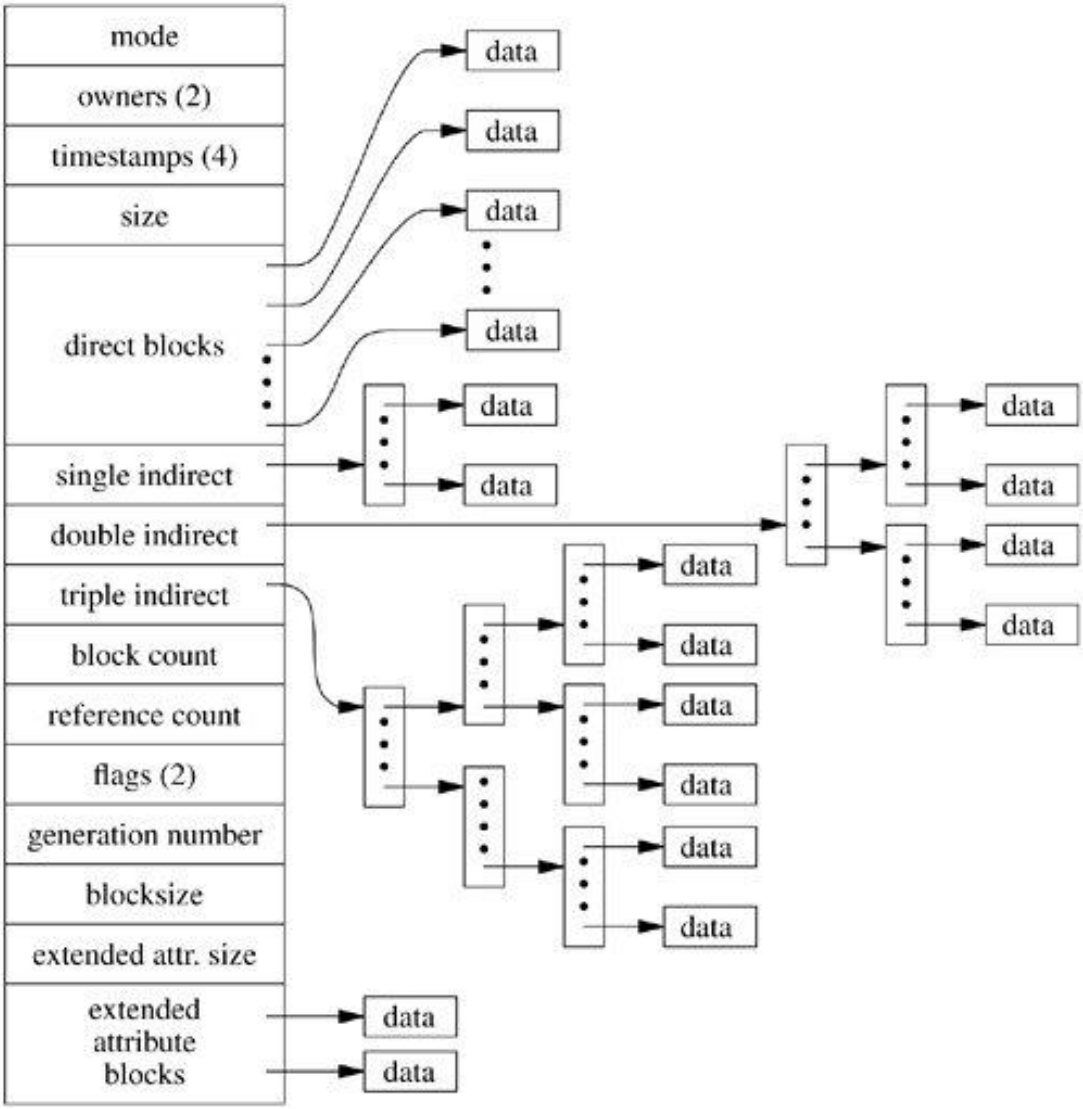
Quando um novo buffer é necessário, o kernel primeiro verifica a quantidade de memória que está dedicada aos buffers existentes. Se a memória em uso é abaixo do seu limite admissível, um novo buffer é criado a partir da lista EMPTY, caso contrário, o buffer mais antigo é removido do inicio da lista CLEAN. Se a lista CLEAN estiver vazia, o buffer daemon é iniciado para limpar e liberar um buffer da lista DIRTY.

5 Estrutura de um inode

Os inodes (*index node*) são descritores internos de um arquivo no UFS, ele é formado pelos seguintes campos:

- Tipo e modo de acesso
- Dono do arquivo e identificação do grupo
- Hora em que o arquivo foi criado, seu último acesso e última atualização de seu inode
- Tamanho do arquivo em bytes
- Número de blocos utilizados pelo arquivo
- Número de diretórios que referenciam o arquivo
- Flags definidas pelo usuário ou kernel
- Número de geração do arquivo
- Tamanho do bloco utilizado pelos atributos de acesso
- Tamanho dos atributos estendidos

Figura 6: Estrutura dos inodes no disco



Fonte: McKusick (2004)

Nos inodes não existe um campo para guardar os nomes dos arquivos. Nomes de arquivos são mantidos dentro de diretórios porque um arquivo pode ter vários nomes, links ou o nome pode ser grande (255 bytes). Diretórios serão tratados no próximo capítulo.

O inode contém um array de ponteiros para os blocos no arquivo. O tamanho dos inodes é fixo e a maior parte dos arquivos é pequena, então o array de ponteiros precisa ser pequeno para uso eficiente do espaço em disco. As primeiras 12 entradas estão alocadas no inode em si. Esta implementação permite os primeiros 96 ou 192 kbytes de dados possam ser localizados diretamente via uma simples busca indexada. Para arquivos grandes, a figura 5 mostra que o inode possui ponteiros indiretos para blocos. O *single indirect* aponta para um bloco indireto

simples de ponteiros para outros blocos. Quando o arquivo possui megabytes ou gigabytes de tamanho, mais blocos indiretos são necessários, então são utilizados os campos *double indirect* e *triple indirect*, que podem conter dois ou três níveis de ponteiros até acessar o bloco de dados.

Os atributos estendidos são dados auxiliares de armazenamento associados a um inode que pode ser utilizados para armazenar dados que são separados do conteúdo do arquivo. Estes atributos são utilizados para guardar dados referentes à ACL (*Access Control List*), que permite ter uma lista de usuários com permissão para acessar o arquivo, e MAC (*Mandatory Access Control*), que é um framework utilizado pelo kernel e suporta uma variedade de serviços de segurança.

O dono do arquivo ou o administrador do sistema podem definir as flags do arquivo. Estas flags permitem que o arquivo seja marcado como *append-only* (adicionar somente) e *immutable* (imutável). Um arquivo append-only é também imutável (*immutable*) exceto pelos dados que podem ser adicionados a ele, arquivos marcados como *immutable* podem ser modificados somente por alguém com acesso físico á máquina ou ao console.

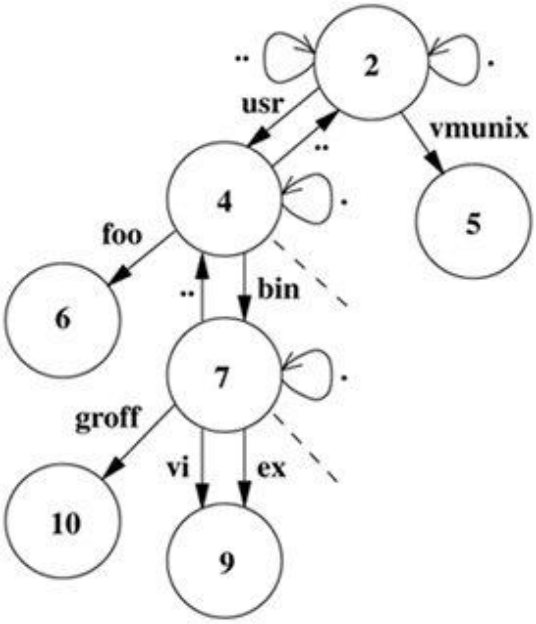
O primeiro passo ao abrir um arquivo é encontrar o inode associado a ele, a busca é feita pelo sistema de arquivos associado com o diretório sendo utilizado no momento. Quando o sistema de arquivos encontra o nome do arquivo no diretório ele retorna o inode associado a ele, então verifica se o inode já está na memória.

O próximo passo é localizar no disco o bloco contendo o inode e colocá-lo em um buffer na memória do sistema. Quando a última referência de um arquivo é fechada o sistema de arquivos é notificado que o arquivo tornou-se inativo então ele pode atualizar a data e hora do inode e escrevê-lo no disco.

6 Diretórios

Em um sistema de arquivos, alguns arquivos são identificados como diretórios e contém ponteiros para outros arquivos, que podem também ser diretórios. A hierarquia de um sistema de arquivos é organizada em uma estrutura de árvore. Na figura 6, cada circulo representa um inode, com seu número dentro, cada seta representa um nome em um diretório. Por exemplo, o inode 4 é o diretório /usr com a entrada “.”, apontando para ele mesmo, e a entrada “..”, que aponta para o diretório pai, inode 2, o diretório raiz.

Figura 7: Estrutura de diretórios em árvore



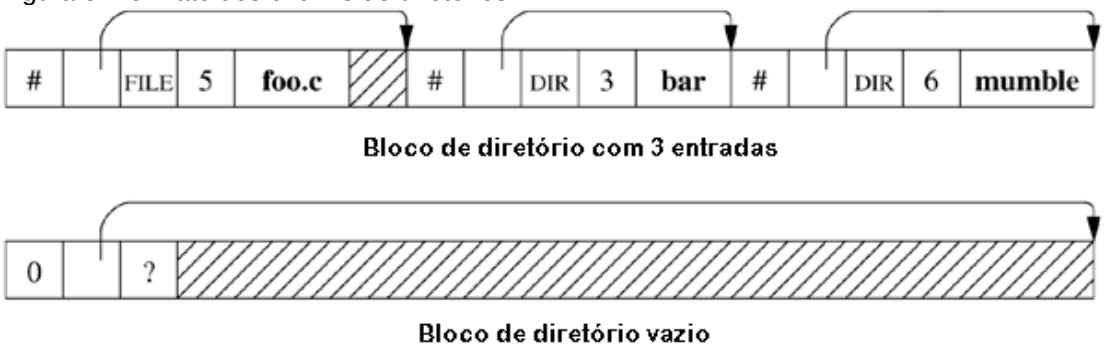
Fonte: McKusick (2004)

Diretórios são armazenados em uma unidade chamada *chunks* (pedaços), que são grupos de entradas de diretórios. Sistemas de arquivos antigos como o System V tinham um tamanho fixo para o registro de arquivos, o que significa que muito espaço seria desperdiçado se uma provisão para nomes longos fosse feita. No UFS, cada diretório pode ser de tamanho variável, fornecendo um mecanismo para nomes longos sem perder muito espaço. Nomes de arquivos no UFS podem ter até 255 caracteres. O tamanho de um chunk é definido de forma que cada alocação possa ser transferida para o disco em uma única operação, normalmente 512 bytes, tornando as atualizações atômicas. Os primeiros quatro campos de uma entrada de diretório têm tamanho fixo e contém o seguinte:

1. O número do inode;
2. Tamanho da entrada em bytes
3. O tipo da entrada
4. O tamanho do nome do arquivo contido na entrada em bytes

O restante da entrada é de tamanho variável e contém um nome do arquivo, terminado em null. O espaço alocado para um diretório deve ser contabilizado pelo total dos tamanhos das entradas existentes. Quando uma entrada é excluída de um diretório, o sistema de arquivos junta o espaço desta entrada na entrada do diretório anterior adicionando o tamanho da entrada excluída no chunk anterior. Se a primeira entrada de um chunk de diretório está livre o ponteiro do inode é definido como zero para mostrar que aquela entrada não está alocada.

Figura 8: Formato dos chunks de diretórios



Fonte: McKusick (2004)

Quando é criada uma nova entrada em um diretório, o kernel precisa escanear todo diretório para garantir que o nome ainda não exista, enquanto ele faz este processo, também verifica se existe espaço suficiente para adicionar a nova entrada. O espaço precisa ser contíguo. O kernel irá compactar as entradas válidas dentro de um bloco do diretório para juntar pequenos espaços não utilizados em um único espaço grande o suficiente para receber a nova entrada. O kernel não irá compactar espaço através de dois blocos de diretório nem criar uma entrada que ocupe dois blocos, pois ele deve ser capaz de fazer atualizações no diretório

escrevendo um único bloco. Se não houver espaço suficiente, um novo bloco será alocado no final de diretório.

6.1 Encontrando nomes e caminhos de arquivos

Quando o kernel precisa procurar por um nome específico em um diretório, ele normalmente começa no inicio do diretório e segue comparando entrada por entrada. Primeiro, o tamanho do nome solicitado é comparado com o nome sendo verificado, se o tamanho for igual os nomes são comparados. Quando os nomes baterem, a busca está finalizada, senão ela continua com a próxima entrada. Sempre que um nome é encontrado, seu nome e diretório são colocados no cache. Quando uma busca não é bem sucedida, uma entrada é adicionada ao cache informando que aquele nome não existe no diretório em questão. Antes de iniciar uma busca o kernel verifica o cachê, assim a busca no diretório pode ser evitada.

Outra operação comum é buscar todas as entradas de um diretório, como no comando ls, por exemplo, que lista todas as entradas de um diretório. Para melhorar o desempenho destes programas o kernel mantém o deslocamento da última busca bem sucedida para cada diretório. Cada vez que uma nova busca é realizada, ela é iniciada do ponto onde o último nome foi encontrado. Para programas que percorrem sequencialmente um diretório com n arquivos, o tempo de busca cai de n^2 para n.

O cache oferece bom desempenho em buscas, mas é ineficaz para grandes diretórios com alta taxa de criação e exclusão de entradas. Sempre que uma nova entrada é criada, o kernel precisa escanear o diretório para garantir que não exista outra entrada igual. Quando uma entrada é excluída ele também precisa encontrar a entrada a ser excluída. Estas operações podem consumir muito tempo do sistema.

Para evitar buscas lineares em grandes diretórios, no FreeBSD 5.2 foi introduzido o hash dinâmico de diretórios. Este método constrói uma tabela *on the fly* quando um diretório é acessado pela primeira vez, isto evita buscas por todo o diretório após excluir ou criar nova entradas. O efeito do hash dinâmico de diretórios é que grandes diretórios causam problemas mínimos de desempenho.

Considerando o sistema de arquivos mostrado na figura 6, considere que o sistema precisa encontrar o arquivo **/usr/bin/vi**. O sistema precisa primeiro buscar o diretório **usr** dentro do diretório raiz. Ele encontra o inode que descreve o diretório raiz, por convenção o inode 2 é sempre reservado para o diretório raiz, e leva o inode para a memória. Este inode indica onde estão os blocos de dados do diretório raiz, eles também precisam ser levados para a memória para que seja possível buscar pelo diretório **usr**. Encontrando a entrada para **usr**, o sistema sabe que ele é descrito pelo inode 4, o sistema traz o inode 4 para saber onde estão seus blocos de dados. Buscando estes blocos é encontrada a entrada **bin**, que aponta para o inode 7. Buscando os blocos associados com o inode 7, a entrada **vi** é encontrada. Descobrindo que **vi** é descrito pelo inode 9, o sistema pode trazer os inodes e os blocos que contém o binário **vi**.

7 Conclusão

O Fast File System, desenvolvido pela equipe de Berkeley para o 4.2BSD, trouxe grandes melhorias em relação ao antigo sistema de arquivos utilizado no UNIX, eliminando vários problemas de performance e utilização do disco.

O aumento do tamanho dos blocos de disco utilizados, a organização do disco em grupos de cilindros, para manter dados e metadados mais próximos, a fragmentação de blocos e as novas políticas de utilização dos blocos de disco foram o primeiro passo para a grande evolução do sistema de arquivos UNIX.

O fato de o sistema BSD ser um sistema de código aberto foi um fator muito importante na evolução de todo os sistema operacional, permitindo que outras equipes de desenvolvimento colaborassem com o projeto. A Sun Microsystems foi uma das empresas que colaborou e ainda colabora com o desenvolvimento, não só do sistema de arquivos, mas de outras funcionalidades do sistema operacional.

O desenvolvimento da interface vnode/vfs pela Sun foi de grande importância, pois permitiu que outros sistemas de arquivos locais ou remotos fossem utilizados sem que grandes alterações no kernel fossem necessárias para suportar um novo sistema de arquivos. O vnode/vfs também fornece recursos independes do sistema de arquivos como o gerenciamento de buffers, que armazenam pedaços dos arquivos em memória, para evitar acessos ao disco toda vez que um processo de leitura ou escrita fosse realizado.

O sistema de arquivos UFS recebeu diversos recursos durante sua evolução como *snapshots*, logs, e o *Soft Updates.* Porém não foram detalhados por não serem o foco deste trabalho, mas é possível encontrar mais detalhes sobre estes recursos na bibliografia utilizada.

O desenvolvimento do FreeBSD mantém-se bastante ativo até hoje, contando com a contribuição de milhares de desenvolvedores, voluntários ou não. Atualmente o desenvolvedor Jeffery Roberson, com a supervisão do Dr. Marshall Kirk McKusick, estão desenvolvendo um sistema de Journaling para inodes, que estará disponível

no FreeBSD 8.1 e discutem o desenvolvimento do UFS3.

REFERÊNCIAS BIBLIOGRÁFICAS

LUCENT TECHNOLOGIES. Disponível em: http://www.bell-labs.com/history/unix/. Acesso em 25 abr. 2010. 14h30.

BACH, M. J. **The design of the UNIX Operating System**. Englewood Clifs: Prentice-Hall, 1986.

McKusick, M. K. [et al]. **A Fast File System for UNIX**. [s.n.], 1984.

Kleiman, S. R. **Vnodes: An Architecture for Multiple File System Types in Sun UNIX**. In: Proceedings of the Summer 1986 USENIX Conference. Atlanta. 1986. Disponível em: http://www.solarisinternals.com/si/reading/vnode.pdf. Acesso em 3 mai. 2010. 21h10.

McKusick, M. K., Neville-Neil, G. V. **The design and implementation of the FreeBSD Operating System**. [s.l.]: Addison-Wesley Professional, 2004.

PATE, S. D. **UNIX Filesystems – Evolution, design and implementation**. Indianapolis: Wiley Publishing, 2003.

Mauro, J., McDougall, R. **Solaris Internals**. [s.l.]: Sun Microsystems Press, 2000.