UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Hugo Marcondes

Um Sistema de Arquivos para o EPOS

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Dr. Antônio Augusto Medeiros Fröhlich

Florianópolis, Novembro de 2004

Um Sistema de Arquivos para o EPOS

Hugo Marcondes

Esta dissertação foi julgada adequada para a obtenção do título de Bacharel em Ciência da Computação, e aprovada em sua forma final pela Coordenadoria do Curso de Bacharelado em Ciência da Computação.

____
Prof. Dr. José Mazzucco Junior

Banca Examinadora

____
Prof. Dr. Antônio Augusto Medeiros Fröhlich

____
Prof. Dr. Rômulo Silva de Oliveira

____
M. Sc. Marcelo T. Pereira

"O dicionário é o único lugar onde
a palavra sucesso vem antes de trabalho."
- Cyrus K. Curtis

"A todos que acreditaram
em meu trabalho..."

Agradecimentos

Primeiramente gostaria de agradecer ao meu pai, que fomentou não apenas o meu interesse pelos computadores, mas acima de tudo ensinou-me o valor do conhecimento e a minha mãe por todo apoio que me oferece nos momentos de alegria e tristeza. A minha namorada que sempre me incetivou a nunca desistir de meus sonhos, e que para mim é um exemplo de determinação. Agradeço ao meu orientador que sempre acreditou na minha competência para a realização deste trabalho e sempre me deu a plena liberdade na execução do mesmo. A Carl Sagan, que através de seus livros, soube inspirar o desejo e fascínio pela busca do conhecimento científico. Deixo aqui também um enorme obrigado a todos os meus amigos, principalmente ao Luiz Felipe, que sempre me apoiou durante os últimos anos da universidade, ao Burny por tornar os meus dias mais felizes, aos meus chefes que me deram a flexibilidade necessária para conciliar trabalho e estudo, principalmente ao Renato, que sempre ter modificado seus horários para que eu pudesse cumprir com os horários na universidade, a todos vocês o meu muito obrigado de coração. Não posso aqui, neste momento de agradecimentos, esquecer do Marcelo e de todos os companheiros do LISHA que sempre compartilharam suas experiências e conhecimentos comigo, tornando-se peça fundamental para o sucesso deste trabalho. Gostaria também de agradecer toda a comunidade de software livre, não só por compartilhar o conhecimento através da distribuição de código fonte, mas também por viabilizar a execução deste projeto, através do uso de ferramentas livres, como por exemplo o LaTeX, que esta sendo utilizado para gerar este documento, e finalmente agradeço a Deus por estar sempre iluminando o meu caminho...

Resumo

O mercado de sistemas embutidos anda em crescente ascensão. Nos dias de hoje é quase impossível não encontrarmos equipamentos eletrônicos que utilizem sistemas de controle para implementar suas funcionalidades. Com o crescimento deste mercado, suas aplicações tem-se tornado cada vez mais complexas. Tendo em vista isto, é crescente a necessidade da utilização de Sistemas de Arquivos para armazenar de forma persistente uma grande quantidade de dados. Este trabalho se concentra na modelagem, seguindo a metodologia de Application Oriented System Design, das abstrações necessárias para que o sistema operacional EPOS ofereça suporte a sistemas de arquivos em tal ambiente. Um sistemas de arquivos criado na RAM do sistema foi implementado para validar a modelagem proposta.

Keywords: sistemas operacionais, sistemas embutidos, sistema de arquivos.

Abstract

The market of embedded systems is increasingly growing. Nowadays, it is almost impossible not to find electronic equipment that use control systems to implement its functions. Given this market growth, its applications have become more and more complex. In view of this, the need to utilize File Systems to consistently store a large quantity of data is increasing. This study, following the Application Oriented System Design methodology, focus on the modeling of necessary abstractions in such a way that the operational system EPOS provides support to files systems in such an environment. A file system created in the RAM of the system was implemented to validate the proposed modeling.

Keywords: operating systems, embedded systems, file systems.

Sumário

Resumo **vi**

Abstract **vii**

Lista de Figuras **3**

1 Introdução **4**

1.1 Motivação 4

1.2 Justificativa 5

1.3 Objetivos 6

2 Sistemas de Arquivos **7**

2.1 Discos 8

2.2 Gerenciamento de blocos 9

2.3 Arquivos 10

2.3.1 Estrutura de arquivos 11

2.3.2 Tipo de arquivos 11

2.3.3 Acesso a arquivos 13

2.3.4 Atributos de arquivos 13

2.3.5 Descritores de arquivos 15

2.3.6 Operações em arquivos 19

2.4 Diretórios 20

2.4.1 Diretório de nível único 20

2.4.2 Diretório de nível duplo 20

2.4.3 Árvores de diretórios 21

2.4.4 Grafos acíclicos de diretórios 21

2.4.5 Grafos de diretórios 22

2.4.6 Operações em diretórios 22

2.5 Sistemas de arquivos estruturados em Log 23

3 Sistemas de arquivos no EPOS 25

3.1 Metodologia e ambiente de desenvolvimento 25

3.1.1 Sistemas Operacionais Orientados a Aplicação 25

3.1.2 EPOS 26

3.2 Famílias de um sistema de arquivos 27

3.2.1 Visão do sistema operacional 28

3.2.2 Visão do usuário 29

3.2.3 A Família Storage 29

3.2.4 A Família Volume 31

3.2.5 A Família Allocator 34

3.2.6 A Família FileSystem 37

3.2.7 A Família File 42

3.2.8 A Família Directory 46

4 Protótipo do Sistema 48

5 Conclusões 51

5.1 Trabalhos Futuros 52

Referências Bibliográficas 53

A Código Fonte 55

B Artigo 87

Lista de Figuras

2.1 Estrutura de arquivos 12

2.2 Alocação contínua de arquivos 15

2.3 Alocação através de listas ligadas de arquivos 16

2.4 Alocação através de listas ligadas indexadas de arquivos 17

2.5 Alocação através de inodos 18

3.1 Decomposição de um domínio através da AOSD 26

3.2 Ferramentas de geração e configuração do sistema EPOS 27

3.3 Famílias do Sistemas de Arquivos 28

3.4 Interface Inflada e membros da família Storage 30

3.5 Interface Inflada e membros da família Volume 32

3.6 Interface dos dados de alocação. 35

3.7 Interface dos alocadores do sistema. 36

3.8 Interface inflada e membros da família FileSystem. 37

3.9 Interface dos descritores de arquivos. 39

3.10 Interface inflada e membros da família File. 43

3.11 Interface inflada e membros da família Directory. 46

4.1 Estrutura geral da especificação implementada 49

4.2 Composição de um sistema de arquivos 50

Capítulo 1

Introdução

Este trabalho consiste no estudo e implementação de componentes para que o EPOS ofereça a aplicação, um sistema de arquivos para armazenamento e leitura de dados em mídias persistentes.

1.1 Motivação

Embora não sejam muito evidentes aos olhos dos consumidores, os sistemas dedicados e embutidos praticamente dominam o mercado da computação [Fröhlich 2003], estando cada vez mais presentes nos automóveis, PDA's[1], microondas, celulares, aparelhos de imagem e áudio, etc. Os sistemas operacionais mais populares no mercado atendem com eficiência apenas os requisitos na área de computação de propósito geral, principalmente pelo fato de serem sistemas de grande extensibilidade [Fröhlich 2003], visto que as aplicações que irão rodar nele são desconhecidas de antemão.

Infelizmente essa abordagem é inadequada ao mundo de sistemas embutidos, visto que o custo de tamanha extensibilidade é desnecessária, uma vez que no mundo dos embutidos a aplicação é geralmente conhecida de antemão.

Visando isto, o sistema operacional EPOS[2], defini um sistema

[1] Personal Digital Assistant

[2] Embedded Parallel Operating System

altamente adaptável a aplicação para suportar sistemas de computação dedicados, mais especificamente, sistemas paralelos e embutidos [Fröhlich 2002].

”Toda aplicação necessita armazenar e recuperar informações”[Tanenbaum e Woodhull 1997], para isso o sistema operacional fornece a aplicação um sistema de arquivos, que define uma forma organizacional com que os dados são armazenados em um dispositivo de armazenamento persistente do ponto de vista físico, permitindo assim que diversos arquivos coexistam e se organizem de forma lógica para o ponto de vista do usuário do sistema operacional.

1.2 Justificativa

Com o avanço da tecnologia, as aplicações de sistemas embutidos estão ficando cada vez mais complexas, necessitando assim da utilização de sistemas de arquivos, para cumprirem seus requisitos. Alguns exemplos, atualmente presentes em nosso cotidiano são os tocadores de MP3, cameras digitais, celulares, entre outros. Assim, é fundamental que o sistema EPOS possa fornecer um sistema de arquivos cumprindo assim os requisitos de tais aplicações.

Adicionalmente este projeto contribui para a comunidade científica através do desenvolvimento de uma nova modelagem para sistemas de arquivos, através de metodologia *Application Oriented System Design* [Fröhlich 2001], compartilhando seus resultados na academia.

Pessoalmente, a realização deste trará uma carga adicional de conhecimento científico ao meu currículo acadêmico nas áreas de engenharia de software básico e sistemas embutidos.

O EPOS fornece um repositório de componentes para a geração e configuração automática de sistemas operacionais que atenda apenas as necessidades de uma determinada aplicação, eliminando assim possíveis *overheads* causados por componentes presentes, porém não necessários a aplicação. Com isso a necessidade de se reescrever um sistema operacional para que o mesmo seja talhado as necessidades específicas de uma aplicação é reduzida, favorecendo assim um menor tempo ao mercado

(*time-to-market*) e o seu custo final.

1.3 Objetivos

O principal objetivo deste projeto é adquirir conhecimentos amplos e sólidos na área de desenvolvimento de software básico e sistemas operacionais, modelando e implementando as abstrações necessárias para o EPOS oferecer a aplicação um sistema de arquivos. Para isso é necessário:

- Consolidar meus conhecimentos no desenvolvimento de software básico, adquirindo experiência em técnicas de engenharia de software para a concepção de software básico.
- Modelar o domínio de sistemas de arquivos segundo a metodologia exposta em [Fröhlich 2001] , identificando suas famílias, cenários e aspectos.
- Implementar um sistema de arquivos, comprovando assim a validade da modelagem proposta.

Capítulo 2

Sistemas de Arquivos

Os sistemas de arquivos visam solucionar três fatores que limitam a execução de aplicações em um sistema computacional. São elas:

- Quantidade limitada para armazenamento de informações no espaço de endereçamento de um processo
- Perda das informações contidas no espaço de endereçamento após a execução de um processo
- Compartilhamento de informações entre processos distintos em um sistema

Segundo [Tanenbaum e Woodhull 1997], a solução mais comum para esses fatores é o armazenamento de informações em discos ou outras mídias externas em unidades chamadas de arquivos. Assim os processos podem ler e escrever em arquivos as informações que necessitam ser compartilhadas, mantidas após a sua execução e as informações que são grandes o suficiente para não caberem na memória principal do sistema, dentro do espaço de endereçamento do processo.

Assim os arquivos são gerenciados pelo sistema operacional e a maneira como eles são estruturados, nomeados, acessados, utilizados, protegidos e implementados são tópicos importantes no desenvolvimento de um sistema operacional [Tanenbaum e Woodhull 1997]. Dá-se o nome de sistemas de arquivos a parte do sistema operacional que manipula os arquivos.

2.1 Discos

Afim de garantir a persistência dos dados, os sistemas de arquivos são armazenados em dispositivos de dados permanentes. Em sua grande maioria esses dispositivos são discos rígidos magnéticos ou ópticos[1], embora atualmente existam tecnologias de armazenamento em memórias não voláteis, como a memória FLASH, que se assemelham as memórias SDRAM e garantem a persistência dos dados mas infelizmente possuem um custo muito elevado, aumentando conseqüentemente o custo de produção em sistemas embutidos que requerem armazenar grandes quantidades de dados.

Independente disso, ambos trabalham como dispositivos orientados a blocos, isso por que o seu espaço para o armazenamento de dados é divido e acessado na forma de blocos físicos. O tamanho do bloco físico pode variar, conforme o seu projeto de hardware e sua geometria no caso de discos. Geralmente os discos magnéticos possuem blocos físicos de 512 bytes ou seja cada endereço físico acessa blocos de 512 bytes.

Atualmente é comum encontrarmos discos que variam a sua capacidade entre alguns megabytes a terabytes, no campo dos usuários domésticos os mais comuns são de 40Gb. Se considerarmos esses 40Gb divididos em blocos físicos de 512 bytes esse disco possuíria um espaço de endereçamento de 83.886.080 endereços. Um sistema de arquivos deve saber quais blocos estão ocupados com dados que devem ser mantidos e com dados que estão livres para serem alocados para a criação e redimensionamento de arquivos e para isso deve-se manter uma estrutura para esse controle.

Gerenciar essa estrutura e armazená-la no próprio disco pode se tornar uma tarefa de alto custo para o sistema caso o número de blocos físicos seja elevado. Por esse motivo existe um outro nível de blocos chamados de blocos lógicos, ou simplesmente blocos, que é utilizado pelo sistema de arquivos.

O tamanho do bloco definido pelo sistema de arquivos é de suma importância para a performance do sistema de arquivos. Blocos muito pequenos implicam em arquivos com muitos blocos, ocasionando em perda de performance devido ao

---

[1]CD-ROM, CD-RW, DVD

aumento do tempo de busca desses blocos no disco. Blocos muito grandes aumentam a fragmentação interna, ocasionando assim em um desperdício da capacidade de armazenamento. Um estudo de [Mullender e Tannembaum 1984] mostra que o tamanho médio de um arquivo no sistema UNIX é por volta de 1Kb.

Para contornar esses problemas pode-se utilizar blocos de tamanho variável, contudo sua implementação não é nada trivial e se torna uma tarefa bem complexa, sendo na maioria dos casos mais favorável encontrar um tamanho fixo de bloco ideal de acordo com a capacidade do disco.

Um disco pode ser particionado em diversos volumes, de capacidade inferior a capacidade do disco como um todo. Os sistemas de arquivos estão contidos nos volumes e só conhecem o volume a qual pertencem, sendo assim, cada volume é uma unidade independente para o sistema. Somente uma parte do sistema operacional diferencia um disco físico e um volume, sendo que o resto do sistema conhece apenas os volumes. [Fröhlich 1994]

As principais vantagens do uso de volumes, conforme [Fröhlich 1994] são uma maior tolerância a falhas, já que caso algum dado seja corrompido apenas o volume a qual ele pertence é corrompido, além de uma melhor gestão dos blocos, devido a menor capacidade de armazenamento dos volumes.

2.2 Gerenciamento de blocos

O gerenciamento de blocos consiste na tarefa de gerenciar, dentro de um volume, os blocos que estão sendo utilizados e os blocos que estão livres para serem alocados aos arquivos.

Existem duas formas principais para esse gerenciamento, a primeira delas é através de listas ligadas de blocos de disco. A lista armazena os endereços dos blocos livres no sistema. Em um bloco de 1Kb com endereçamento de 32bits é possível armazenar 255 blocos livres, uma das posições da lista ligada é utilizada para informar em qual bloco esta a próxima parte da lista.

A grande vantagem deste método é que o espaço gasto para o

armazenamento pode variar, diminuindo a medida que o disco é preenchido com arquivos, contudo um sistema de arquivos com pouco espaço utilizado implica em uma lista de blocos livres grandes, representando assim um gasto maior de memória principal do sistema.

Uma segunda alternativa é a utilização de bitmaps para representar a gestão de blocos, cada bit representa um bloco, caso o bit esteja em 1 o bloco representado por ele está ocupado, caso contrário o mesmo está livre. Ao contrário do método anterior o tamanho do bitmap é fixo e varia conforme o número de blocos disponíveis para a gestão. Um volume de 4Gb com blocos de 1.024 bytes possui 4.194.304 blocos que devem ser representados por um bitmap cujo tamanho é de 524.288 bytes (4.194.304 blocos / 8 bits = 524.288 bytes). Esse método é aconselhável quando todo o bitmap pode ser armazenado na memória do sistema sem o comprometimento do mesmo [Tanenbaum e Woodhull 1997].

2.3 Arquivos

Arquivos são um mecanismo de abstração [Tanenbaum e Woodhull 1997]. Eles provem uma maneira para o armazenamento de informações (dados) de forma persistente, para um acesso futuro. Embora [Tanenbaum e Woodhull 1997] cite como característica mais importante a um mecanismo de abstração a forma como estas são nomeadas, um arquivo não precisa necessariamente possuir um nome contextualizado, ou seja, um nome definido pelo seu criador e que geralmente possui alguma relação com seu conteúdo. Pensando em um Sistema de Arquivos como um conjunto de elementos chamados de arquivo, podemos fazer referência a eles, e conseqüentemente acessá-los, através da sua posição dentro deste conjunto. Um exemplo seria acessar o arquivo através do índice dentro da lista de descritores de arquivos, conceito este abordado na seção 2.3.5. Em um sistema interativo como o Windows, talvez seja inaceitável que o sistema de arquivos não possua um sistema de nomeação, contudo no mundo de sistemas embutidos é bastante razoável os arquivos não possuírem nomes.

2.3.1 Estrutura de arquivos

Os arquivos podem ser estruturados de diversas formas. A principal forma é considerar arquivos como uma seqüência de dados alinhados em 1 byte, enfim, um grande array de bytes. Esta forma de estrutura, vista na figura 2.1(a), prove ao usuário uma flexibilidade máxima, uma vez que os programas dos usuários podem inserir os dados da maneira que achar melhor, sendo assim a mais comum nos sistemas atuais.

Uma segunda forma de estruturar os arquivos é através de registros, conforme a figura 2.1(b), sendo um arquivo formado por uma coleção de registros. Cada registro possui uma tamanho fixo e alguma estrutura interna. Nesse tipo de estrutura de arquivos, as operações de leitura retornam um determinado registro e operações de escrita sobrescrevem ou adicionam registros ao arquivo. Segundo [Tanenbaum e Woodhull 1997] antigamente, quando os cartões perfurados ainda eram bastante utilizados, muitos sistema operacionais implementavam os seus arquivos nessa estrutura, sendo que cada registro possuía 80 caracteres, tamanho esse de cada cartão perfurado. Um exemplo de sistema que utilizava essa estrutura é o CP/M.

Uma terceira forma de estrutura, é através de uma árvore de registros, representada pela figura 2.1(c), onde para cada registro é associada uma chave. Esta árvore é ordenada através das chaves, favorecendo assim a busca de registros com uma determinada chave. Segundo [Tanenbaum e Woodhull 1997] essa estrutura de arquivo é amplamente utilizada em mainframes de larga escala e alguns processamentos de dados comerciais.

2.3.2 Tipo de arquivos

De fato, muitos sistemas operacionais podem definir tipos para os arquivos contidos em sua implementação. Com isso alguns arquivos podem possuir uma semântica distinta e serem tratados de forma diferente. Nos sistemas Windows ou Unix por exemplo, temos os tipos *arquivos regulares* e *diretórios*. Os arquivos regulares são os arquivos que contém dados do usuário, enquanto o arquivo do tipo diretório possuem informações do sistema de arquivos para a implementação de diretórios, sendo assim a sua

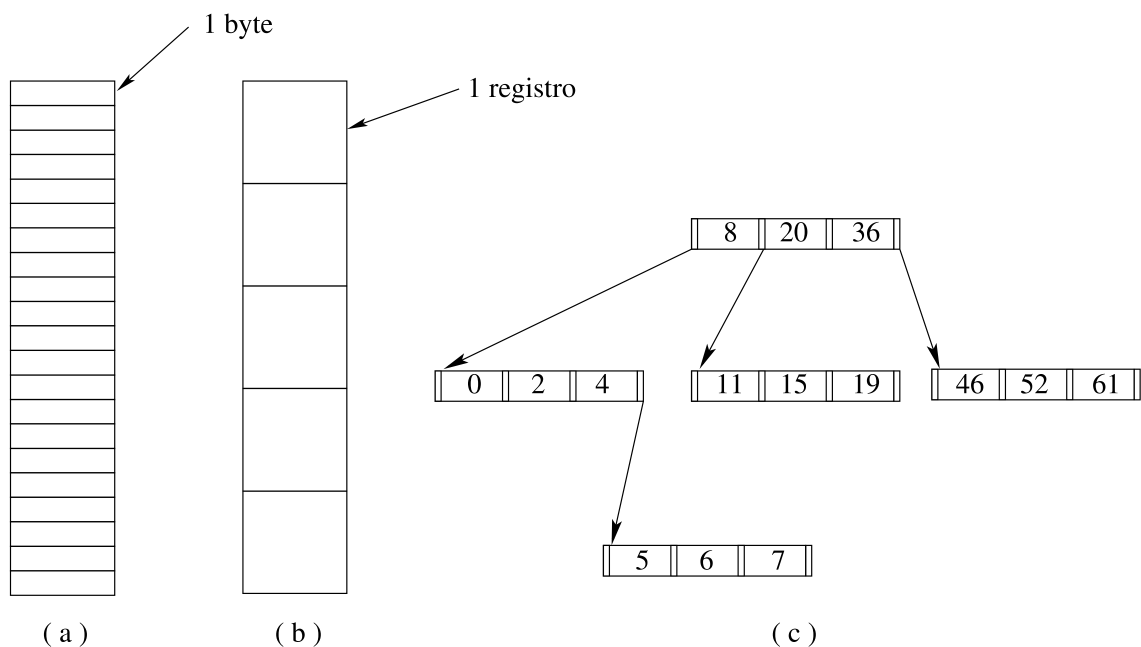

Figura 2.1: Estrutura de arquivos

(a) Seqüência de bytes. (b) Seqüência de registros. (c) Árvore de registros

existência como um arquivo, de certa forma, é transparente ao usuário, já que o conjunto de operações fornecidas sobre esses arquivos são distintas das oferecidas para arquivos regulares.

Os arquivos regulares basicamente se dividem em dois grupos. Os arquivos de texto ASCII [2], constituídos por linhas de puro texto no formato ASCII, e os arquivos binários, constituídos por bytes e que geralmente possuem uma estrutura interna que deve ser tratada, se for o caso, pela aplicação que o manipula. Existem diversas maneiras de identificar a aplicação que manipula determinado arquivo binário, evitando assim que determinado arquivo seja manipulado acidentalmente. Os sistemas UNIX utilizam o magic number [Tanenbaum e Woodhull 1997], constituído pelos primeiros 16 bits do arquivo binário que formam um código que pode ser checado pela aplicação, evitando que a mesma manipule os dados de forma incorreta. O sistema Windows utiliza o sufixo do nome do arquivo, também conhecido como extensão do arquivo, para tal

[2] American Standart Code for Information Interchange

identificação.

2.3.3 Acesso a arquivos

Os arquivos podem ser acessados de duas formas diferentes: Seqüencialmente ou Randômicamente. Devido as próprias tecnologias de dispositivos de armazenamento existentes na época, os sistemas antigos proviam apenas uma forma de acesso: seqüencial. O **acesso seqüencial** permite a leitura dos bytes em ordem, não podendo efetuar operações de deslocamento de bytes e lê-los de forma não ordenada, contudo, arquivos seqüenciais podem ser ”rebobinados”para serem lidos ou escritos quantas vezes forem necessário. Arquivos seqüenciais são inerentes a mídias do tipo fita magnética, devido suas características.

Com o surgimento dos discos magnéticos, tornou-se possível ler os bytes ou registros de arquivos fora de ordem, ou no caso de arquivos estruturados em árvores de registros, acessar seus registros por chaves ao invés de sua posição. Este tipo de acesso, fora de ordem, ao conteúdo de um arquivo é denominado **acesso randômico**.

2.3.4 Atributos de arquivos

Todo arquivo possui o seus dados e possivelmente um nome contextualizado, conforme visto na seção 2.3. Um sistema operacional, pode contudo, agregar aos arquivos outras informações que achar conveniente, adicionando assim funcionalidades ao sistema de arquivos, como proteção ao acesso de seus dados, bloqueio de acessos, entre outros. Essas informações adicionais são conhecidas como **atributos do arquivo**. O tamanho do arquivo se inclui dentro desses atributos e talvez seja o mais fundamental para um implementação concisa de um sistema de arquivos, já que tal informação é de suma importância para a execução de diversas operações sobre arquivos.

Outros atributos estão ligados a fatores como segurança (proprietário do arquivo, acessos permitidos aos arquivos, senha para acesso), controle de idade (dia e hora de criação, dia e hora da ultima alteração), etc.

Na tabela 2.1 podemos verificar alguns desses atributos.

Atributo	**Significado**
Proteção	Quem pode acessar o arquivo e de que maneira
Senha	Senha necessária para o acesso
Criador	Identificador do criador
Proprietário	Identificar do proprietário
Apenas Leitura	Informa se o arquivo permite apenas a leitura
Oculto	Informa se o arquivo não é visível
Sistema	Informa se o arquivo é de sistema
Arquivar	Informa se deve ser efetuado um backup do arquivo
ASCII/Binário	Informa o tipo do arquivo (ASCII ou binário)
Acesso Randômico	Informa o tipo de acesso
Temporário	Informa se o arquivo é temporário
Bloqueado	Informa se o arquivo esta bloqueado para o acesso
Tamanho do registro	Número de bytes em cada registro
Posição da chave	Posição da chave dentro de cada registro
Tamanho da chave	Tamanho da chave
Tempo de criação	Data e horário de criação do arquivo
Tempo do último acesso	Data e horário do último acesso ao arquivo
Tempo da última modificação	Data e horário da última modificação ao arquivo
Tamanho	Tamanho do arquivo
Tamanho máximo	Tamanho máximo que o arquivo pode atingir

Tabela 2.1: Alguns possíveis atributos para arquivos

2.3.5 Descritores de arquivos

Descritores de arquivos são estruturas contidas no sistema de arquivos que visam descrever um arquivo no sistema. São transparentes para o usuário e são utilizadas apenas para controle do sistema de arquivos. [Tanenbaum e Woodhull 1997] apresenta 4 implementações diferentes para se descrever os dados de um arquivo.

2.3.5.1 Alocação Contínua

Esta é a forma mais simples para descrever a alocação dos dados de um arquivo. Como neste tipo de alocação os dados estão contidos em um conjunto de blocos seqüenciais, basta o seu descritor conter a posição do bloco inicial do arquivo, e os blocos seguintes podem ser acessados somando-se o seu deslocamento..

Este método é de fácil implementação e reduz o tempo de busca dos blocos no disco, já que estes estão seqüenciais. Contudo suas desvantagens são a necessidade de se estabelecer um tamanho inicial do arquivo para a sua alocação e conseqüente dificuldade em aumentar o tamanho do mesmo, necessitando alocar uma nova porção seqüencial do disco e copiar o arquivo para esta região.

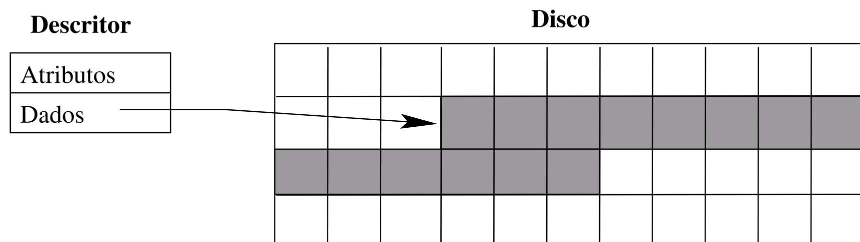

Figura 2.2: Alocação contínua de arquivos

2.3.5.2 Alocação em Lista Ligada

Semelhante a alocação contínua, o descritor de arquivo continua possuindo apenas uma referência ao primeiro bloco de dados do arquivo, contudo, visando anular as desvantagens do método anterior cada bloco de dados do arquivo contém uma

referência ao próximo bloco do arquivo, conforme mostrado na figura 2.3. Desta maneira novos blocos podem ser alocados, em qualquer posição do disco, bastando agora apenas colocar uma nova referência a este novo bloco no bloco anterior a ele.

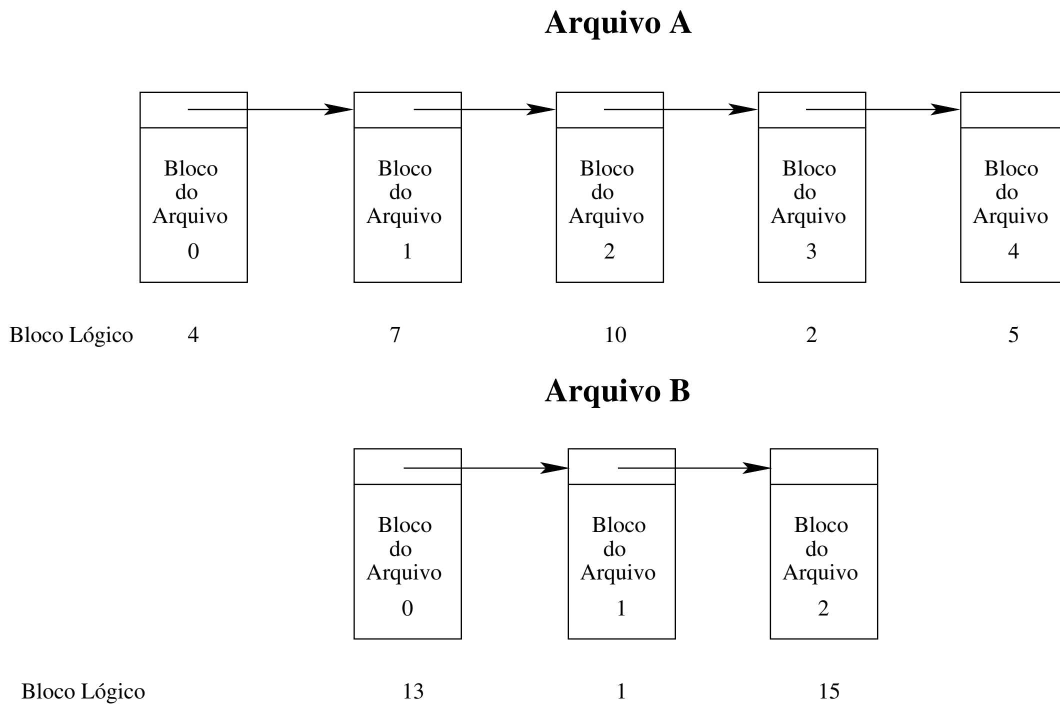

Figura 2.3: Alocação através de listas ligadas de arquivos

Sua principal desvantagem ocorre no momento que um acesso randômico é efetuado, sendo necessário ler toda a cadeia de blocos para se chegar ao bloco desejado. Além disso, a capacidade de cada bloco não e mais uma potência de dois, devido ao ponteiro que cada bloco deve possuir, embora não seja um grande problema, isto acarreta em perda de eficiência, já que a maioria dos programa lê e escreve em blocos cujo tamanho é uma potência de dois [Tanenbaum e Woodhull 1997]

2.3.5.3 Alocação em Lista Ligada Indexada

Visando eliminar as duas desvantagens da utilização de listas ligadas para a alocação dos dados de um arquivo, foi criado este método, que utiliza uma tabela indexada na memória principal para armazenar a seqüencia da lista ligada. Cada posição

da tabela representa um bloco lógico do disco e o seu conteúdo é o índice para o próximo bloco na cadeia. O último bloco possui o conteúdo nulo.

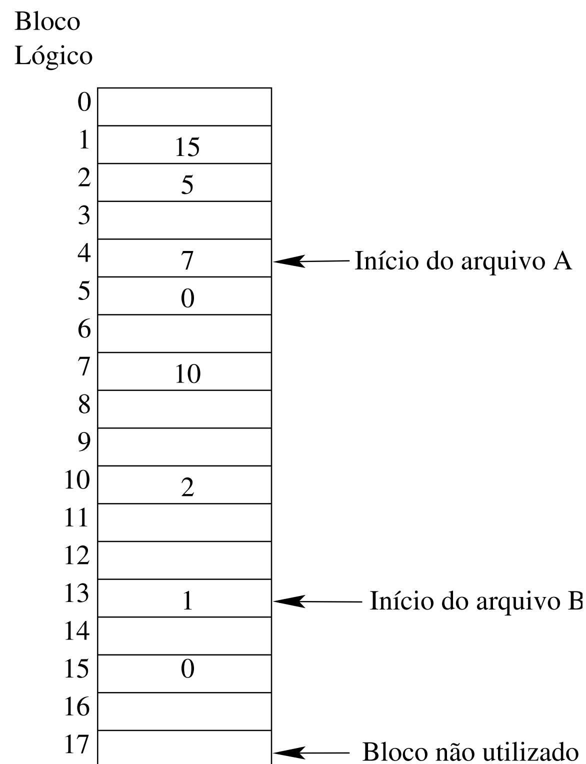

Figura 2.4: Alocação através de listas ligadas indexadas de arquivos

Esta tabela é armazenada no disco e precisa ser lida apenas no momento em que o sistema de arquivos é carregado na memória. Assim, seguir o encadeamento da lista deixa de ser uma tarefa cara devido a alta velocidade de acesso a memória principal do sistema.

Sua principal desvantagem é que a tabela toda deve estar presente na memória principal e nem sempre esse recurso pode estar presente em abundância, principalmente em sistemas embutidos. Além do fato que a tabela cresce de acordo com o tamanho do disco. Para um disco de 10 Gb ou 2.500.000 de blocos de 4 Kb, considerando que cada entrada da tabela consumiria 4 bytes, a tabela iria possuir aproximadamente 10 Mb. Certamente uma quantia preciosa em termos de memória principal.

2.3.5.4 Alocação em Inodos

Este método utiliza uma pequena tabela de tamanho fixo para descrever os arquivos, denominada inodo (nodo indexado), que possui os atributos de um arquivo, assim como os endereços dos blocos que compõe o mesmo.

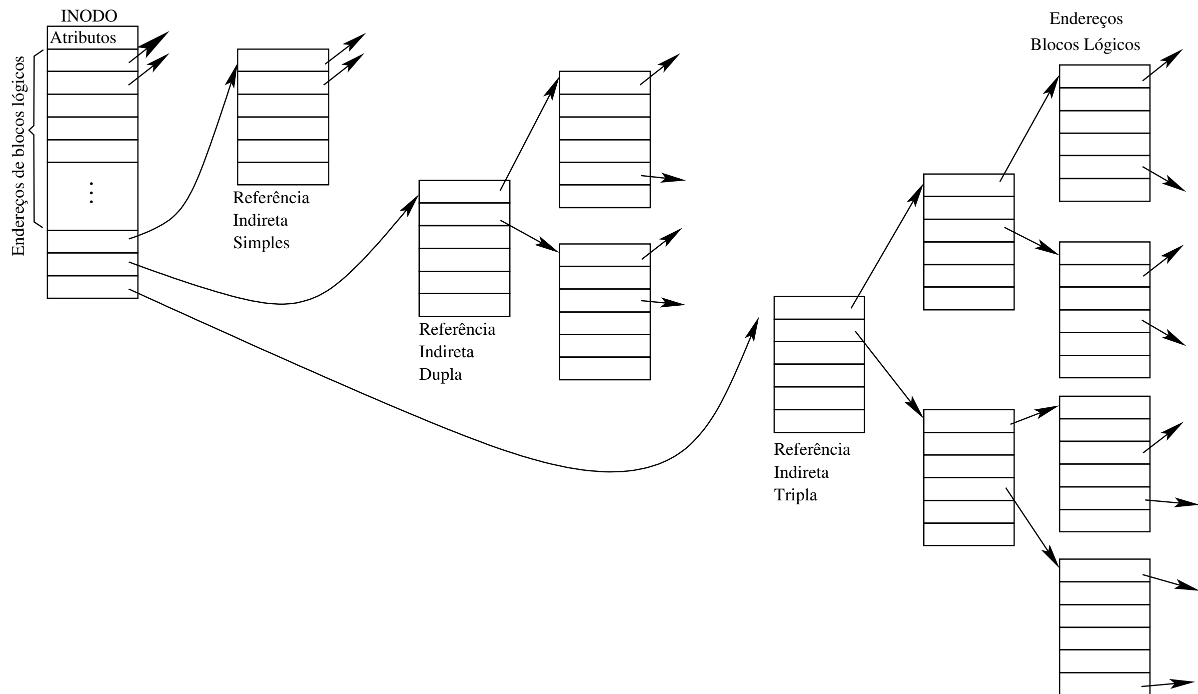

Figura 2.5: Alocação através de inodos

Os primeiros endereços de blocos são armazenados no próprio inodo, que é carregado do disco para a memória no momento que o arquivo é aberto favorecendo assim a performance de arquivos pequenos. Isso implica em um fator limitante quanto ao tamanho máximo que um arquivo pode ter, visando ampliar o tamanho máximo de um arquivo, cada inodo possui uma referência indireta simples. Esta referência indireta indica um bloco que possui endereços de blocos de dados adicionais. Caso isso não seja suficiente para endereçar os blocos de um arquivo, um segundo nível é criado, chamado de referência indireta dupla, que endereça um bloco que contém uma lista de blocos com referências indiretas simples. Um terceiro nível de referências indiretas também pode ser utilizado caso necessário.

2.3.6 Operações em arquivos

Um sistema de arquivos deve fornecer um conjunto de operações para a manipulação do arquivo por parte do usuário. Abaixo iremos apresentar as operações mais comuns:

- **Criação :** Inicializar e cria as estruturas iniciais de um arquivo
- **Remoção :** Apaga um arquivo, quando o mesmo não é mais necessário, removendo e marcando como disponíveis os recursos (blocos, descritores) que um arquivo utilizava.
- **Abertura :** carrega o descritor de um arquivo na memória, viabilizando assim o acesso aos seus dados.
- **Fechamento :** grava o estado atual do arquivo no disco e remove seu descritor da memória.
- **Leitura :** Efetua a leitura efetiva dos dados de um arquivo, retornando os mesmos para o usuário
- **Escrita :** Efetua a escrita de dados em um arquivo, podendo sobrescrever dados já existentes ou adicionar novos dados aos arquivos.
- **Anexar :** Efetua a escrita de dados concatenado-os no final do arquivo.
- **Procurar :** Efetua um deslocamento para um determinada região do arquivo, assim chamadas subseqüentes das operações de leitura ou escrita serão iniciadas nesta região
- **Recupera atributos :** Efetua a leitura dos atributos de um arquivo.
- **Modifica atributos :** Efetua a modificação dos atributos de um arquivo.
- **Renomear :** Efetua a troca no nome de um arquivo

2.4 Diretórios

Um sistema de arquivos pode possuir milhares de arquivos. O conceito de diretórios surge para criar uma forma de organização dentro de um sistema de arquivos.

O diretório pode ser visualizado como uma tabela de símbolos que traduz nomes de arquivos em suas entradas de diretório [Silberschatz, Galvin e Peterson 1998]. Ou seja, os diretórios são um sistema de tradução de nomes de arquivos. Cada arquivo possui a sua entrada, que pode ser o próprio descritor do mesmo, ou dependendo do sistema de arquivos conter apenas o seu nome e uma referência para o seu descritor. De fato a maneira como a entrada de diretório é implementada esta estritamente ligado a especificação de cada sistema de arquivos.

Dentro da concepção de diretórios, diversos modelos estruturais para os diretórios podem ser implementados. A seguir será descrito alguns deles:

2.4.1 Diretório de nível único

A maneira mais simples de organização através de diretórios e através da existência de apenas um único diretório. Todos os arquivos estão relacionados neste diretório em uma tabela de entradas de diretório.

Algumas limitações são evidentes nesta forma de organização, não sendo aconselhado para sistemas grandes que deverão armazenar um grande quantidade de arquivos, ou mesmo para sistemas multi usuários. Já que cada entrada deve possuir um identificador único, geralmente o nome do arquivo. Utilizar essa organização em sistemas que possuem diversos usuários significa que nenhum usuário poderá ter arquivos com o mesmo nome.

2.4.2 Diretório de nível duplo

A principal desvantagem do diretório de nível simples é a confusão causada entre arquivos de diversos usuário distintos [Silberschatz, Galvin e Peterson 1998]. Para contornar isso foi criado um segundo

nível de diretório. Cada usuário do sistema possui o seu próprio diretório de arquivos. Assim o sistema de arquivos possui um diretório central que aponta para os diretórios de arquivos do usuário. Novos diretórios de arquivos podem ser criados e removidos conforme necessário. Este modelo resolve o problema da coexistência de arquivos com o mesmo nome, bastando para isso que os arquivos estejam em diretórios de diferentes.

2.4.3 Árvores de diretórios

Pensando em uma generalização, podemos organizar os diretórios em árvores de altura indefinida. Neste caso os usuários podem criar novos diretórios dentro de seus diretórios, formando assim uma estrutura de árvore. De fato, esta é a estrutura mais comum.

Assim cada arquivo possui um único nome, chamado de caminho, formado pelo caminho da raiz da árvore de diretórios até a sua folha que é o próprio arquivo, e que também é conhecido como caminho absoluto.

O caminho também pode ser relativo a algum subdiretório do sistema, ou seja, se iniciar em algum nodo(subdiretório) da árvore de diretórios, ao invés da raiz.

2.4.4 Grafos acíclicos de diretórios

A estrutura em árvore não permite que arquivos e diretórios sejam compartilhados entre usuários em localizações distintas da árvore. Contudo é interessante que usuários que compartilham seus arquivos possam cria-los no caminho que acharem mais conveniente. Pensado nisso foi criada a estrutura de diretórios em grafos acíclicos, que é uma generalização do conceito de árvores. No grafo acíclico é possível compartilhar pastas e diretórios através de entradas especiais de diretório. [Silberschatz, Galvin e Peterson 1998] ressalta que arquivos compartilhados não é o mesmo que duas cópias de um arquivo, onde modificações efetuadas em uma das cópias não se refletem nas outras.

Em sistemas UNIX, essas entradas de diretórios são chamadas de link e no sistema Windows de atalhos (shortcuts).

Alguns problemas emergem ao adicionarmos esse conceito de atalhos em uma estrutura dessas. O principal é pelo fato que, quando a árvore de diretórios é atravessada (na busca de um arquivo por exemplo), um mesmo arquivo pode ser lido diversas vezes, caso o mesmo seja compartilhado. Além disso a exclusão de um arquivo se torna mais complexa, já que um arquivo pode ter diversas referências distintas.

2.4.5 Grafos de diretórios

Uma quarta estrutura pode ser obtida, caso um sistema de arquivos permita a criação de links diretos a diretórios, através destas e possível criar-se ciclos no grafo de diretórios de um sistema, Isso gera um agravante aos problemas dos grafos acíclicos já que se torna mais difícil identificar ciclos no momento que a árvore é atravessada e aumentando a complexidade de se identificar quando um arquivo pode ser marcado como excluído, sendo a solução para este último o uso de um coletor de lixo para identificar quando um arquivo não possui nenhuma referência.

2.4.6 Operações em diretórios

As operações que podem ser efetuadas em diretórios são descritas abaixo:

- **Criação :** Efetua a criação de um diretório vazio apenas com as referências a ele mesmo e ao seu pai.
- **Remoção :** Remove um diretório caso esteja vazio.
- **Abertura :** Inicializa as estruturas necessárias para a leitura das entradas do diretório.
- **Ler :** Efetua a leitura de uma entrada de diretório.
- **Renomear :** Modifica o nome (chave) de uma entrada de diretório
- **Link :** Cria um link para um outro caminho da estrutura de diretório.

- **Unlink :** Remove o link para um outro caminho da estrutura de diretório.

2.5 Sistemas de arquivos estruturados em Log

A *lei de Moore* [Moore 1965] vem trazendo problemas aos atuais sistemas de arquivos, uma vez que a cada dia os processadores se tornam mais rápidos, os discos maiores e mais baratos, e as memórias vem crescendo de forma exponencial. Enquanto isso o tempo de busca por blocos não vem acompanhando este ritmo [Tanenbaum e Woodhull 1997]. Assim um grande gargalo de performance vem surgindo nos atuais sistemas de arquivos. Estudos realizados em Berkeley tentam aliviar esse problema modelando uma especifícação distinta de sistemas de arquivos entitulada de *Log-Structured FileSystem* (LFS) [Rosenblum e Ousterhout 1992].

A idéia básica dessa modelagem é o aproveitamento do crescimento da cache de disco, favorecida pelos avanços dos processadores e memórias, satisfazendo assim que grande parte dos acessos de leitura do discos sejam cumpridos diretamente pelos dados da cache do sistema de arquivos. Dessa maneira, os acessos efetivos ao disco são dominados por acessos de escrita. Infelizmente grande parte dos acessos de escrita são efetuados em regiões pequenas do disco (atualização de um descritor de arquivos, atualização de uma pequena parte do bitmap de alocação) e em consequência disto a performance do disco diminui devido a latência ocasionada pelo posicionamento da cabeça do disco a efetiva região de escrita da operação. Para se ter uma idéia, uma simples criação de arquivos é necessario atualizar o bitmap de descritores de arquivos, inicializar o descriptor de arquivos com os dados iniciais do mesmo e gravar uma entrada de diretório do arquivo correspondente, sendo que cada uma dessas tarefas acarretam na escrita de dados em regiões distintas do disco, na maioria dos sistemas de arquivos. Segundo [Tanenbaum e Woodhull 1997], tipicamente uma escrita de 50 microsegundos, demora 10 microsegundos de atraso de busca e mais 6 microsegundos de atraso na rotação do disco, diminuindo assim a eficiência do disco a uma fração de 1 porcento.

A idéia básica do LFS é estruturar o disco como um log. Periodicamente, todas as modificações que são temporariamente armazenadas na

memória do sistema, são juntadas em um único segmento de tamanho fixo e gravadas no disco de forma sequencial no final do log. Com isso um único segmento pode conter diversos meta-dados distintos misturados no mesmo. O início de cada segmento há uma tabela descrevendo todos os meta-dados e dados presentes no mesmo. Esta abordagem favorece então um melhor aproveitamento da largura de banda de tráfego de dados do disco, uma vez que esta junta um conjunto de pequenas escritas no disco em uma única operação de escrita contínua do segmento no disco.

Contudo, desta abordagem também emergem problemas, principalmente no gerenciamento dessas estruturas. Agora os descritores de arquivos não possuem uma posição fixa no disco, sendo que sua posição também muda a cada vez que são atualizados. Com isso estruturas adicionais como uma tabela de descritores de arquivos mantida na memória devem ser utilizadas para poder localizar a posição atual de um determinado descritor. Esse sistema também implica na implementação de um coletor de lixo (*Garbage Collector*), responsável em atualizar e restruturar o log, liberando novos segmentos de dados que possuem dados muitos desatualizados. A idéia, uma vez que cada segmento possue um tamanho fixo, é ter um processo, que ao identificar um segmento com muitos dados desatualizados, copie os dados atuais para o *buffer* do sistema (que futuramente ao atigir o tamanho de um segmento será armazenado em um novo segmento do log) e marque este como livre, para receber um novo segmento futuramente. A implementação e concepção de algoritmos para realizarem a função de coletores de lixo é a principal área de estudo desse tipo de sistema de arquivos.

Capítulo 3

Sistemas de arquivos no EPOS

3.1 Metodologia e ambiente de desenvolvimento

Este trabalho utiliza conceitos de modelagem desenvolvidos por Fröhlich em sua tese de doutorado [Fröhlich 2001] entitulada de *Application Oriented System Design* (AOSD) ou em português, ”Projeto de sistemas orientados à aplicação”.

A apresentação da modelagem proposta é realizada utilizando-se diagramas UML [Larman 2001, OMG 1999]. A fim de simplificar os diagramas, foi utilizado uma convenção para a definição de tipos de dados não sinalizados, para evitar que as assinaturas de métodos nos diagramas ficassem muito extensas. Assim os dados são iniciados pela letra *u* seguido pela quantidade de bits que o dado utiliza, assim o tipo de dado *u8* representa um dado do tipo *unsigned char*.

3.1.1 Sistemas Operacionais Orientados a Aplicação

Introduzido em [Fröhlich 2001], os sistemas operacionais orientados a aplicação (*Application-Oriented Operating Systems*) propõe uma nova metodologia para projetos de sistemas, buscando atingir um alto grau de configurabilidade, permitindo assim a geração de sistemas específicos para uma determinada aplicação.

Para isto, a metodologia de projeto multi-paradigma *Application-Oriented System Design* é apresentada pelo mesmo. Tal metodologia

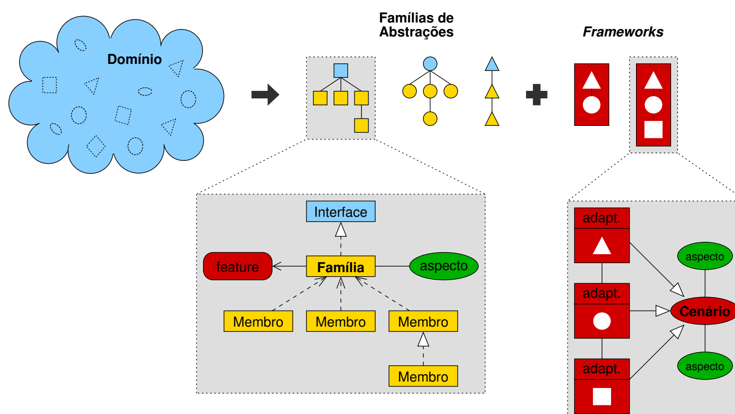

Figura 3.1: Decomposição de um domínio através da AOSD

decompõe um determinado domínio de conhecimento em famílias de abstrações independentes de cenário que, através de re-usabilidade, podem ser instâncias do mesmo sistema, conforme mostra a figura 3.1. Através de "configurable features"e aspectos de cenário, a configurabilidade do sistema pode ser alcançada. Tais famílias de abstrações constituem componentes de software que unidos através de um framework formam o sistema adaptado a aplicação. Os componentes de tal framework são acessados através de Interfaces Infladas que garantem a portabilidade do sistema e são o principal objeto de análise dos requisitos da aplicação.

3.1.2 EPOS

O EPOS é um sistema operacional desenvolvido por Fröhlich, inicialmente para validar os conceitos introduzidos em sua tese e foi concebido para a utilização em sistema embutidos e ambientes paralelos.

O EPOS e composto de três tipos de famílias de componentes: Abstração, Mediadores e Aspectos. As Abstrações são famílias que englobam as características, funcionalidades e estruturas independentes de arquitetura. Possuem grande reusabilidade, compondo uma grande parte das necessidades de determinada aplicação. Os Mediadores são responsáveis por implementar as características dependentes do hardware, implementando as funcionalidades que as abstrações e

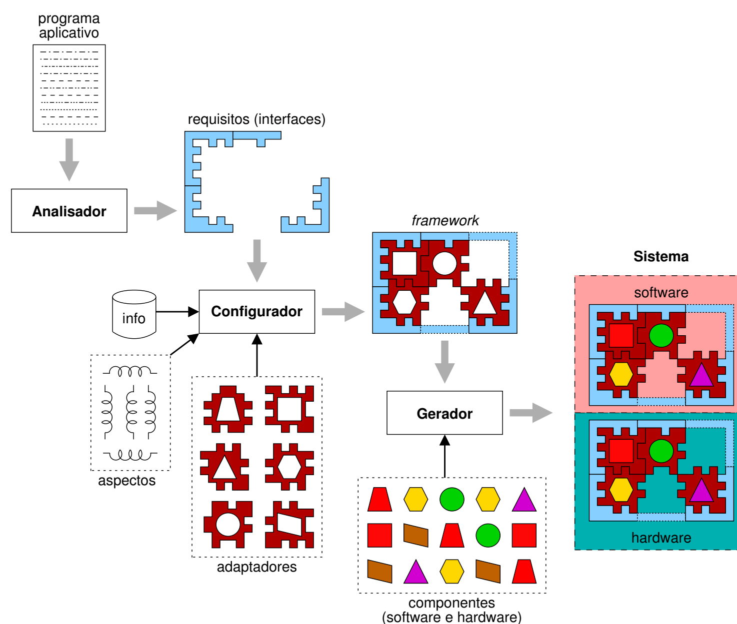

Figura 3.2: Ferramentas de geração e configuração do sistema EPOS

aplicações necessitam do hardware do sistema. As famílias de Aspectos implementam a configurabilidade do sistema.

Além dos componentes, o EPOS oferece um conjunto de ferramentas para a análise dos requisitos da aplicação e para a geração do framework, adaptado a aplicação, configurando o mesmo de acordo com as necessidades do sistema, conforme a figura 3.2.

3.2 Famílias de um sistema de arquivos

O trabalho de projeto iniciou-se com a análise de domínio de sistemas de arquivos. Através dessa análise, buscou-se identificar as **famílias de componentes de software** [Fröhlich 2001] (ver figura 3.3) que integram um sistema de arquivos.

Este domínio foi inicialmente dividido em duas partes, uma composta pela visão do sistema de arquivos pelo sistema operacional, ou seja, um conjunto de meta-dados que oferecem ao usuário uma abstração denominada arquivo. Existe uma outra visão relacionada ao usuário que apenas enxerga um sistema de arquivos como

um conjunto de arquivos, organizado em diretórios. A figura 3.3 mostra os famílias de abstrações, mediadores e aspectos de um sistema de arquivos.

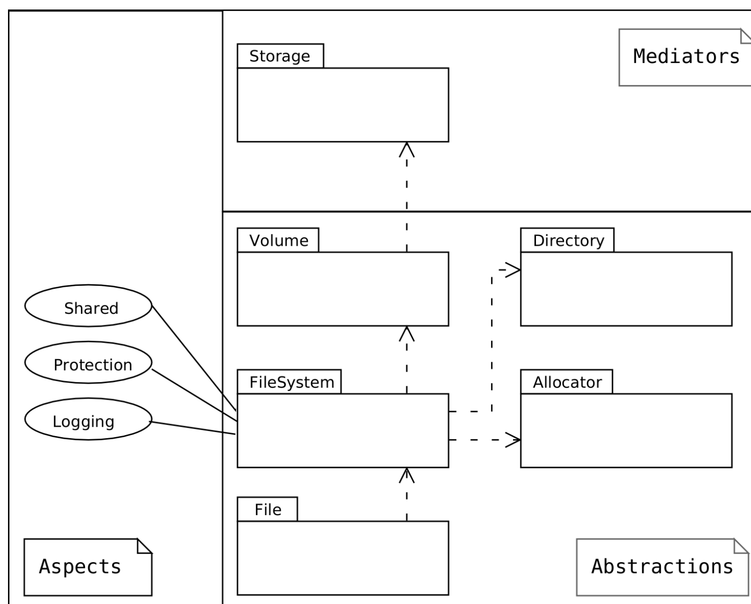

Figura 3.3: Famílias do Sistemas de Arquivos

3.2.1 Visão do sistema operacional

Pensando-se na visão do sistema operacional, um sistema de arquivo é uma coleção de estruturas que contém meta-dados e um dispositivo de armazenamento utilizado para gravar os dados. Dentro desta visão foram identificadas as seguintes famílias:

- **Storage :** Responsável por implementar as funcionalidades de determinado hardware de armazenamento de dados. Esta é uma família de mediadores
- **Volume :** Responsável pela interface entre o sistema de arquivos e o hardware de armazenamento de dados, implementando o conceito de volumes através de uma tabela de volumes.
- **Allocator :** Responsável pelo gerenciamento dos blocos livres no sistema e utilização da tabela de descritores de arquivos, implementando algoritmos de

alocamento de espaço sobre as estruturas que fazem esse controle.

- **FileSystem :** Responsável por gerenciar os meta-dados específicos de um sistema de arquivos e algumas das operações que são realizadas no sistema de forma global, como por exemplo a exclusão de arquivos.

3.2.2 Visão do usuário

A visão do usuário consiste em duas abstrações: Diretórios e Arquivos

- **File :** Implementa as chamadas de sistemas necessários para criar, acessar e modificar arquivos do sistema.
- **Directory :** Implementa as chamadas de sistema responsáveis por acessar diretórios no sistema de arquivo além de incluir, modificar e excluir entradas no diretório.

3.2.3 A Família Storage

A família storage visa implementar os mediadores para acesso a um dispositivo de armazenamento de dados persistentes, entre eles, dispositivos ATA, SCSI, memórias Flash, e todos os dispositivos baseados em blocos. A figura 3.4 mostra alguns membros desta família.

Neste trabalho, o storage foi implementado ”emulando”um dispositivo orientado a blocos, sobre a memória RAM do sistema, um vez que o foco deste trabalho se concentra na modelagem de um sistema de arquivos.

Esta família possui um interface inflada dissociativa [Fröhlich 2001],ou seja, seus membros possuem tanto métodos em comum, como métodos exclusivos aos mesmos. É o caso de membro FlashStorage, que necessita implementar métodos para apagar setores de uma memória flash, e que não fazem sentido em dispositivos ATA.

- **u32 get_block(u32 block, u8 * buffer)** : método responsável em efetuar a leitura de um bloco do dispositivo, seus parâmetros são o indexador do bloco e um ponteiro para um buffer que possui o tamanho do bloco deste dispositivo.

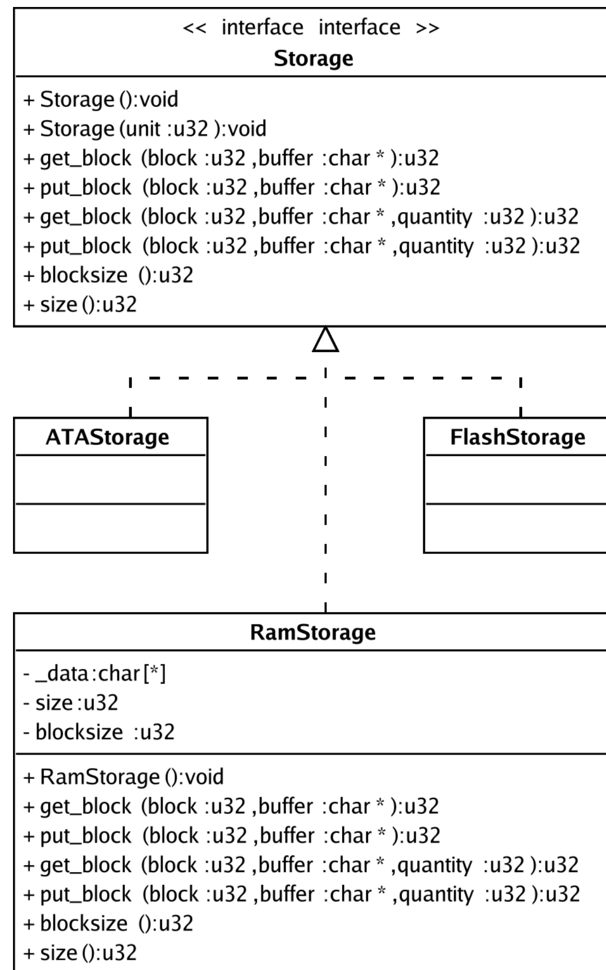

Figura 3.4: Interface Inflada e membros da família Storage

- **u32 put_block(u32 block, u8 * buffer)** : análogo ao anterior, porém este armazena os dados do parâmetro buffer, no bloco informado.

- **u32 get_block(u32 block, u8 * buffer, u32 quantity)** : este método efetua a leitura de um ou mais blocos consecutivos do dispositivo, favorecendo assim o uso de DMA, uma vez que geralmente o bloco lógico de um volume é constituído por diversos blocos físicos do dispositivo.

- **u32 put_block(u32 block, u8 * buffer, u32 quantity)** : efetua a escrita de um ou mais blocos consecutivos no dispositivo.

- **u32 size()** : retorna o tamanho do dispositivo, em número de blocos.

- **u32 blocksize()** : retorna o tamanho em bytes de cada bloco do dispositivo.

De fato a interface inflada acima não esta completa, como mencionado anteriormente, certos dispositivos necessitam de operações adicionais para tratar as suas peculiaridades, como no caso da Flash ou dispositivos de mídias removíveis. As mesmas não foram incluídas neste trabalho pois um estudo aprofundado destes dispositivos não foi realizado, uma vez que não são objetos específicos de estudo deste.

3.2.4 A Família Volume

Esta família implementa uma unidade lógica denominada VOLUME, também conhecida como partição em alguns sistemas. Cada membro desta família encapsula o conhecimento de uma determinada especificação de tabela de volumes. Esta família possui as seguintes funções no sistema como um todo:

- Garantir, caso necessário, a tradução de endereços de blocos lógicos para blocos físicos.

- Garantir o acesso de um determinado sistema de arquivos exclusivamente a região que lhe foi destinada.

- Encapsular o conceito de tabelas de volumes.

Na figura 3.5, podemos verificar os membros, inicialmente atribuídos a essa família. O membro FlatVolume implementa um volume que ocupa todo o espaço de armazenamento do componente Storage, ficando a ele atribuída apenas as funções de tradução de endereços lógicos para físicos. O membro EposVolume implementa uma especificação de tabela de volumes própria do EPOS, visando suprir algumas necessidades de se conhecer o tamanho do bloco lógico. Em grande parte dos sistemas, o tamanho do bloco lógico esta armazenado dentro do próprio sistema de arquivos, com isso é necessário inicialmente, efetuar uma busca no disco para localizar o descritor do sistema, geralmente chamado de superbloco, para se conhecer o real tamanho do bloco lógico, enquanto essa informação já poderia estar armazenada na própria tabela de volumes, simplificando assim o algoritmo de inicialização do sistema de arquivos. O membro DosVolume implementa a especificação de uma tabela de volumes do Sistema Operacional DOS.

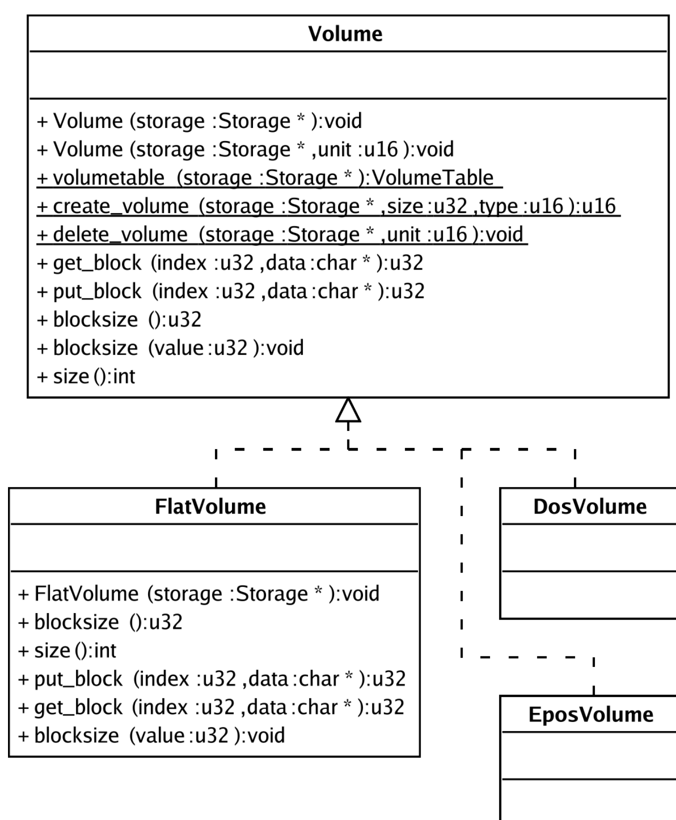

Figura 3.5: Interface Inflada e membros da família Volume

Esta família possui uma interface inflada uniforme, ou seja, seus membros implementam todos os métodos da interface.

- **Volume(Storage * storage)** : Constrói um objeto Volume, selecionando o primeiro volume do dispositivo.

- **Volume(Storage * storage, u16 unit)** : Constrói um objeto Volume, selecionando o volume indicado pelo parâmetro *unit*.

- **static VolumeTable volumetable(Storage * storage)** : Retorna a tabela de volumes do Storage passado como parâmetro, possibilitando assim consultas a tabela de partição.

- **static u16 create_volume(Storage * storage, u32 size, u16 type)** : Cria um volume no dispositivo indicado pelo parâmetro *storage*. O tamanho do volume, assim como o tipo do mesmo são específicados pelos parâmetros *size* e *type* respectivamente. Este método retorna o índice do volume criado.

- **static u16 delete_volume(Storage * storage, u16 unit)** : Remove o volume de índice específicado pelo parâmetro *unit* do dispositivo indicado.

- **u32 get_block(u32 index, u8 * data)** : Método responsável em efetuar a leitura de um bloco do volume, seus parâmetros são o indexador do bloco e um ponteiro para o buffer que irá armazenar os dados.

- **u32 put_block(u32 index, u8 * data)** : Análogo ao anterior, porém este armazena os dados do buffer, no bloco informado.

- **u32 blocksize()** : Retorna o tamanho de cada bloco do volume, em bytes.

- **void blocksize(u32 value)** : Configura o tamanho de cada bloco do volume, em bytes.

- **u32 size()** : Retorna o tamanho do volume, em número de blocos.

3.2.5 A Família Allocator

Esta família é implementada como uma classe utilitária do EPOS, ou seja, seus membros são genéricos para que outras partes do sistema operacional, assim como a própria aplicação, possam utiliza-lo para efetuarem alocação de recursos.

Nesta modelagem, buscou-se encontrar a separação entre o algoritmo de alocação e a estrutura de dados utilizada pelo mesmo para efetuar o gerenciamento dos blocos que estão livre. Através do paradigma de programação genérica, este resultado foi alcançado, utilizando-se o recurso de templates da linguagem C++.

Através da definição de interface comum, conforme a figura 3.6, foi possível modelar um conjunto de classes que são passadas a classe que implementa o algoritmo de alocação através de um template, gerando assim um alocador específico em tempo de compilação, conforme pode ser visualizado na figura 3.7.

Para a implementação do protótipo deste trabalho, foi criada uma classe de dados, chamada PersistentBitmap que é responsável por efetuar a leitura do Bitmap[1] do sistema de arquivos no disco e disponibilizá-lo ao algoritmo através de sua interface, assim como manter os dados atualizados no disco, conforme ocorrem modificações neste Bitmap.

Afim de validar esta modelagem, um algoritmo de alocação simples foi implementado e chamado de DummyAllocator. Este realiza a busca por espaço disponível de forma linear sobre a estrutura de dados utilizada pelo alocador.

Os métodos apresentados na figura 3.6 possuem a seguinte semântica:

- **u32 size()** : Retorna o número de elementos que esta estrutura possui.
- **void * set(void * entry, bool state)** : Altera o estado de determinado elemento referenciado pelo parâmetro *entry*. O novo estado do elemento é indicado pelo parâmetro *state* (livre ou não-livre).
- **bool state(void * entry)** : Retorna o estado do elemento referênciado pelo parâmetro *entry*.

[1] Mapa de bits - um array de bits, onde cada bit representa o estado de cada posição do alocador

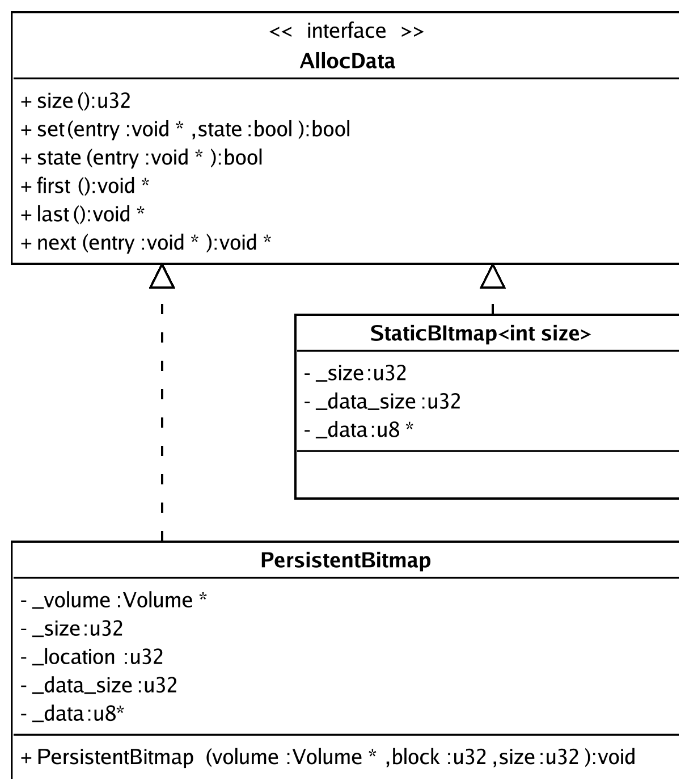

Figura 3.6: Interface dos dados de alocação.

- **void * next(void * entry)** : Retorna uma referência ao próximo elemento do conjunto de dados, em relação ao parâmetro *entry*

- **void * first()** : Retorna uma referência ao primeiro elemento do conjunto de dados.

- **void * last()** : Retorna uma referência ao último elemento do conjunto de dados.

Os métodos apresentados na figura 3.7 possuem a seguinte semântica:

- **void Allocator(AllocData * data)** : Constrói um alocador utilizando os dados da estrutura passada através do parâmetro *data*

- **void * alloc()** : Aloca uma unidade do recurso, retornando um ponteiro para o mesmo

- **void * alloc(u32 size)** : Aloca a quantidade de unidades, específicadas pelo

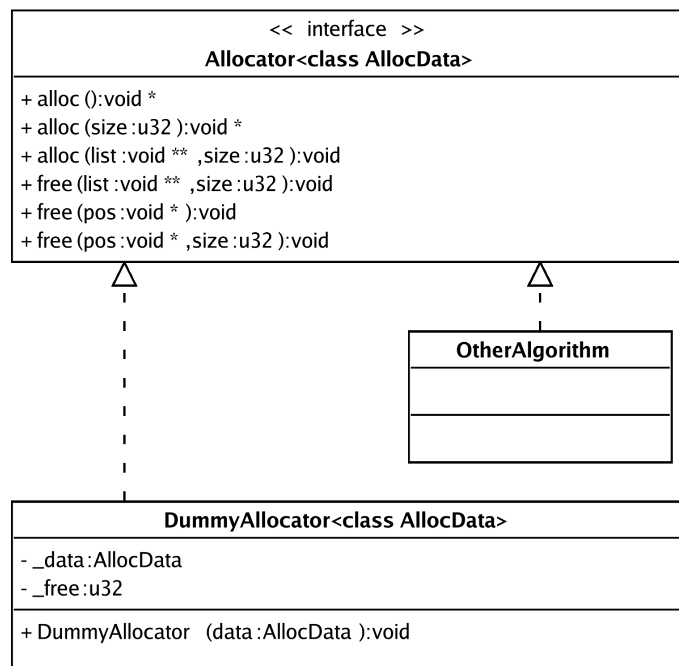

Figura 3.7: Interface dos alocadores do sistema.

parâmetro *size*, de forma contínua, retornando um ponteiro para a primeira unidade alocada

- **void alloc(void ** list, u32 size)** : Aloca a quantidade de unidades, específicadas pelo parâmetro *size*, retornando uma lista de ponteiros para as unidades alocadas

- **void free(void * pos)** : Libera o recurso alocado na posição específicada pelo parâmetro *pos*

- **void free(void * pos, u32 size)** : Libera a quantidade de unidades, específicadas pelo parâmetro *size*, de forma contínua, iniciando pelo recurso apontado pelo parâmetro *pos*

- **void free(void ** list, u32 size)** : Libera os recursos apontados pela lista de recursos *list*. O parâmetro *size* específica o tamanho da lista.

3.2.6 A Família FileSystem

Sendo esta a família central de um de sistemas de arquivos, cabe a esta efetuar grande parte da implementação de uma determinada especificação de um sistema de arquivos.

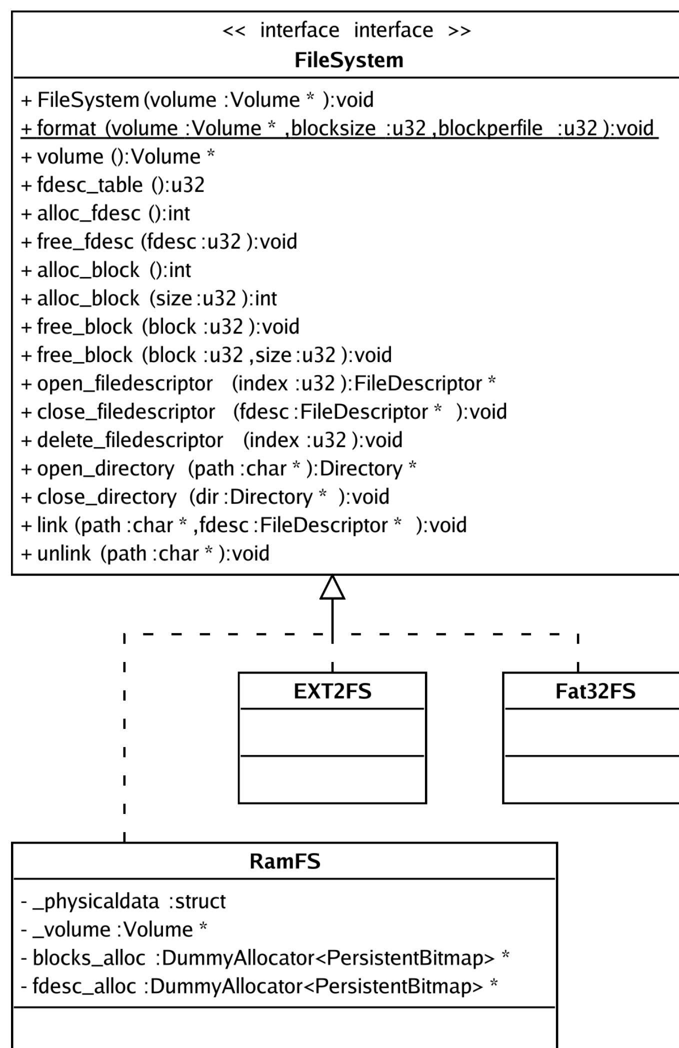

Figura 3.8: Interface inflada e membros da família FileSystem.

Todos os algoritmos necessários para acessar os meta-dados que compõem um sistema de arquivos são realizados nos membros desta. Dessa maneira a implementação dos membros da família File se torna genérica, podendo assim ser utilizada independente do sistemas de arquivos em questão.

Algumas das operações definidas para arquivos e diretórios são implementadas nesta família, uma vez que o contexto destas operações necessitam modificar diversas estruturas internas ao sistema de arquivos. As operações de exclusão de arquivos e diretórios é um caso típico. Ao excluir-se um arquivos do sistema de arquivos é necessária a atualização de diversos meta-dados no descritor do sistema de arquivos, assim como a desalocação dos blocos que este utiliza entre outros dados. Caso, esta operação fosse definida na interface da família arquivo, seria necessário garantir acesso a diversas informações (descritor do sistema de arquivos, alocadores de descriptores de arquivos e blocos) que não são pertinentes ao arquivo em si, quebrando-se o encapsulamento de dados da programação orientada a objetos.

Internamente aos membros do sistema de arquivos é definido uma classe denominada FileDescriptor, responsável por fornecer acesso aos meta-dados necessários a implementação dos algoritmos presentes na família File. A interface deste classe pode ser visualizada na figura 3.9, assim como alguns membros da mesma.

Sendo assim, a família FileSystem além de implementar diversos métodos para algumas das operações sobre arquivos e diretórios, ainda implementa uma ”fábrica”de FileDescriptor que são utilizados pela família File, afim desta poder fornecer o acesso de leitura e escrita aos dados do arquivo. Esta funcionalidade é implementada através do método da interface inflada ***FileDescriptor * open_filedescriptor(index: u32)***.

Adicionalmente, três aspectos foram identificados nesta família, sendo eles o aspecto *Shared*, *Protection* e *Logging*, conforme descrito por [Fröhlich 2001], aspectos compõem um cenário de execução do sistema, implementando diversas caracteríticas de configurabilidade do mesmo. Assim o aspecto *Shared*, diz respeito a todo o compartilhamento de arquivos entre processos implementando as necessidades em termos de algoritmo e estruturas para compartilhar um arquivo. O aspecto *Protection* implementa os algoritmos de proteção de acesso ao arquivo, verificando por exemplo, nas operações de abertura de arquivos, se determinado processo possui permissão de acesso ao mesmo e em que modo (leitura/escrita). O aspecto *Logging* executa os algoritmos de realocação de meta-dados no atual segmento de log do sistema de arquivos, além de atualizar as estruturas em memória como o mapa de descritores de arquivos com a nova

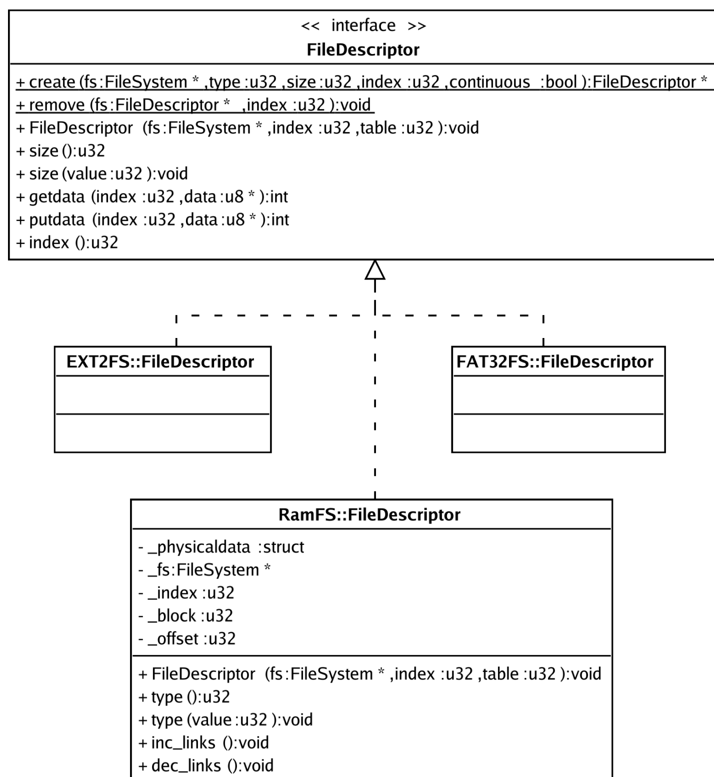

Figura 3.9: Interface dos descritores de arquivos.

localização destes elementos.

A interface inflada da família FileSystem, assim como os atributos específicos do protótipo implementado neste trabalho podem ser visualizadas na figura 3.8 e são seus métodos descritos abaixo:

- **FileSystem(Volume* volume)** : Contrói um sistema de arquivo, lendo as informações específicas presentes no volume. Neste método, os meta-dados são recuperados do volume, convertidos para a ordem de bytes correta na memória (little-endian ou big-endian) de acordo com a arquitetura onde o sistema esta sendo executado. Além disso outras operações que podem ser pertinentes em relação a especificação do sistema de arquivos são executadas (carregamento dos bitmaps de alocação, leitura do dirétorio raiz do sistema, etc...)

- **Volume * volume()** : Retorna um ponteiro para para o volume pertencente ao sistema de arquivos em questão.

- **u32 fdesc_table()** : Retorna o número do bloco no volume onde inicia-se a tabela de descritores de arquivos. Este método é utilizado pelos algoritmos presentes no FileDescriptor.

- **int alloc_fdesc()** : Efetua a alocação de um índice na tabela de descritores de arquivos, retornando o índice alocado ou -1 caso não existam descriptores livres.

- **void free_fdesc(u32 fdesc)** : Libera na estrutura de alocação o índice do descriptor de arquivos passado pelo parâmetro *fdesc*.

- **int alloc_block()** : Efetua a alocação de um bloco livre no volume do sistema de arquivos, retornando o número do bloco alocado ou -1 caso não seja possível alocar um bloco.

- **int alloc_block(u32 size)** : Efetua a alocação de blocos contínuos no volume do sistema de arquivos. A quantidade de blocos alocados é passado através do parâmetro *size*.

- **void free_block(u32 block)** : Libera o bloco identificado pelo parâmetro *block* no volume pertencente a este sistema de arquivos.

- **void free_block(u32 block, u32 size)** : Libera um conjunto de blocos contínuos do volume pertencentes a esse sistema de arquivos, a posição inicial, assim como o tamanho desse conjunto de blocos, são passados através do parâmetro *block* e *size*, respectivamente.

- **FileDescriptor * open_filedescriptor(u32 index)** : Recupera os meta-dados do descritor de arquivos referente ao índice na tabela de arquivos, de acordo com o parâmetro *index*, retornando um ponteiro para um objeto do tipo FileDescriptor, contendo os dados recuperados.

- **void close_filedescriptor(FileDescriptor * fdesc)** : Fecha o descriptor de arquivos, passado pelo parâmetro *fdesc*.

- **void delete_filedescriptor(u32 index)** : Modifica os dados presentes no descritor de arquivos referênciado pelo parâmetro *index*, liberando o mesmo para uso futuro.

- **Directory * open_directory(char * path)** : Abre o diretório especifícado pelo caminho do parâmetro *path*, retornando um ponteiro ao mesmo.

- **void close_directory(Directory * dir)** : Fecha o diretório passado através do parâmetro *dir*.

- **void link(char * path, FileDescriptor * fdesc)** : Inclui a entrada de diretório especifícada pelo paramêtro *path*, associando a mesma ao descritor de arquivos, referênciado pelo parâmetro *fdesc*.

- **void unlink(char * path)** : Remove a entrada de diretório referente ao caminho especifícado pelo parâmetro *path*.

- **static void format(Volume * volume, u32 blocksize, u32 blocksperfile)** : Este método estático é responsável pela formatação do volume passado através do parâmetro *volume*. O tamanho do bloco lógico do sistema é passado através do parâmetro *blocksize* e o parâmetro *blocksperfile* é utilizado no cálculo do número máximo de arquivos que o sistema pode comportar.

- **u32 FileDescriptor::index()** : Retorna o índice do descritor de arquivos representado por este objeto.

- **u32 FileDescriptor::size()** : Retorna o tamanho do arquivo referente a este descritor.

- **void FileDescriptor::size(u32 value)** : Altera o tamanho do arquivo referente a este descritor para o valor do parâmetro *value*.

- **int FileDescriptor::getdata(u32 index, u8 * data)** : Efetua a leitura no bloco do índice passado através do parâmetro *index* copiando estes para o endereço referenciado pelo parâmetro *data*.

- **int FileDescriptor::putdata(u32 index, u8 * data)** : Efetua a escrita no bloco do índice passado através do parâmetro *index* copiando os dados presentes no endereço referenciado pelo parâmetro *data* para o bloco.

- **static FileDescriptor * FileDescriptor::create(FileSystem * fs, u32 type, u32 size, u32 index, bool continuous)** : Efetua a inicialização do descritor de arquivos indicado pelo parâmetro *index*. Diversas informações adicionais são passadas que influenciam a inicialização dos dados do descritor. O parâmetro *continuous* por exemplo configura o algoritmo executado neste método para efetuar a alocação contínua de blocos a esse descritor.

- **static FileDescriptor * FileDescriptor::remove(FileSystem * fs, u32 index)** : Torna o descritor de arquivos livre para posterior utilização, desalocando os blocos que o mesmo possuia.

3.2.7 A Família File

Esta família é o principal componente desta modelagem, sendo esta responsável pelo acesso efetivo aos arquivos do sistema. Sua modelagem foi efetuada visando ser a mais genérica possível, sem que peculiaridades de uma determinada especifícação de sistema de arquivos influencie sua interface. Desta maneira foi possível criar componentes realmente genéricos e que podem ser aproveitados independentemente do sistema de arquivos que está sendo utilizado.

Foi modelado três membros distintos nesta família, são eles: *ContinuousFile*, *RandomFile* e *CircularFile*, conforme mostra a figura 3.10. A principal diferença entre estes membros, se dá na forma como o arquivo é acessado pelo usuário.

O membro ContinuousFile implementa um arquivos onde os dados são alocados de forma contínua no disco, seu acesso pode ser efetuado de forma randômica, contudo um tamanho inicial para o arquivo deve ser específicado no momento de sua criação e toda vez que o mesmo necessitar crescer em tamanho, uma tarefa custosa de realocar e copiar todo o arquivo para uma nova área contínua do disco pode ser efetuada.

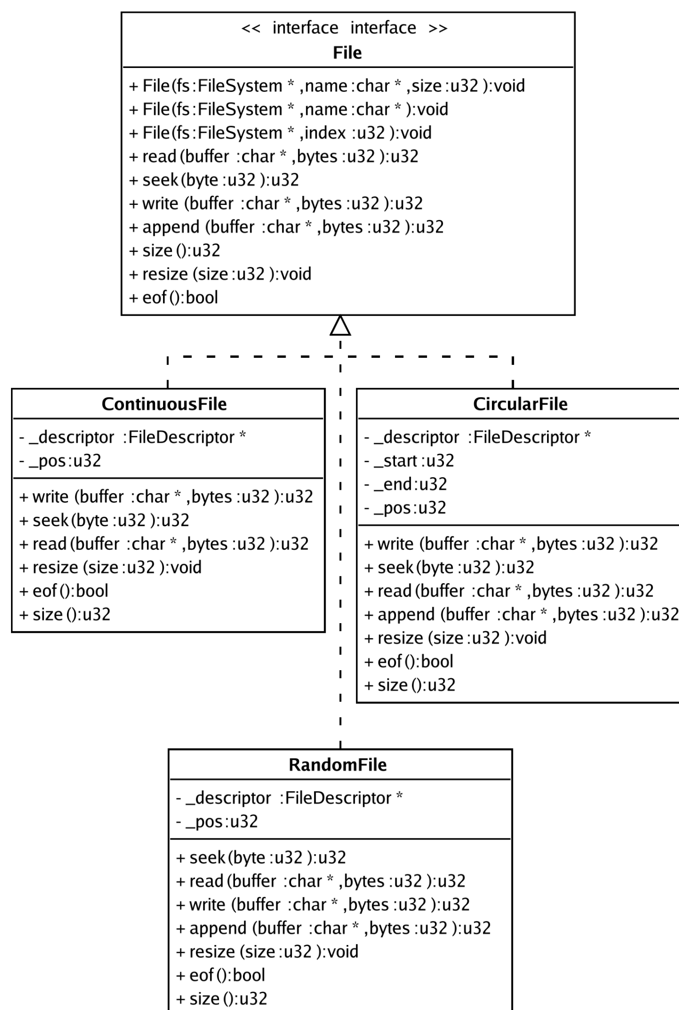

Figura 3.10: Interface inflada e membros da família File.

O membro RandomFile implementa a idéia mais comum de arquivos que temos atualmente, um arquivo que pode ser alocado por blocos não consecutivos no disco, possuindo acesso randômico a estes.

O membro CircularFile implementa um tipo de arquivo bem específico e geralmente utilizados em arquivos de log. A ídeia básica para este arquivo é que o mesmo possui um tamanho fixo, e da mesma maneira que no membro ContinuousFile, pode ser aumentado ou reduzido através de uma operação custosa ao sistema. Contudo este arquivo não possui um final e a partir do momento que é efetuada uma gravação no último bloco alocado a este, novos dados são armazenados no começo do arquivo,

sobrescrevendo os dados que estavam no início do arquivo. Essa é uma característica bastante interessante para ser adotada em arquivos de log de sistemas. Muitas vezes só existe a necessidade de manter armazenado as últimas informações de log de um sistema, não existindo a necessidade de armazenar dados antigos. Nestes casos um arquivo deste tipo pode ser utilizado com o objetivo de se manter apenas uma certa quantidade de logs recentes em um arquivo, garantindo também que o mesmo não irá crescer e ocupar um espaço maior que o estipulado pelo usuário ao arquivo durante sua criação, ou através do redimensionamento do mesmo explícitamente pela chamada do método ***u32 resize(u32 size)***.

Abaixo descrevemos os métodos da interface inflada da família File:

- **File(FileSystem * fs, u32 index)** : Abre o arquivo do sistema de arquivos passado através do parâmetro *fs* cujo indíce na tabela de arquivos é específicado pelo parâmetro *index*. Caso o índice na tabela não esteja sendo utilizado, o arquivo é criado e alocado neste indíce da tabela.

- **File(FileSystem * fs, char * name)** : Abre um arquivo do sistema de arquivos passado através do parâmetro *fs* cujo nome é passado através do parâmetro *name*. Caso o arquivo não exista, o mesmo é criado no sistema.

- **File(FileSystem * fs, u32 size)** : Cria um arquivo de tamanho inicial específicado pelo parâmetro *size* no sistema de arquivos referenciado pelo parâmetro *fs*.

- **File(FileSystem * fs, char * name, u32 size)** : Cria um arquivo no sistema referenciado pelo parâmetro *fs*, atribuindo o nome específicado pelo parâmetro *name*, de tamanho inicial atribuído pelo parâmetro *size*.

- **u32 read(u8 * buffer, u32 bytes)** : Efetua a leitura da quantidade de bytes passado através do parâmetro *bytes*, copiando os bytes lidos no buffer referenciado pelo parâmetro *buffer*, retornando a quantidade de bytes que foram efetivamente lidos. A leitura dos bytes é efetuada a partir de um ponteiro interno que aponta sempre para o ultimo byte lido ou escrito, pondendo este ser modificado pelo método ***u32 seek(u32 byte)***.

- **u32 write(u8 * buffer, u32 bytes)** : Grava a quantidade de bytes específicados pelo parâmetro *bytes*, copiando os dados presentes no buffer específicado pelo parâmetro *buffer*. Caso o tipo de arquivo permita o redimensionamento do mesmo, o arquivo tem o seu tamanho aumentado, caso todos os bytes não tenham sido gravados e o ponteiro interno do arquivo chegue ao final do mesmo. No caso de arquivos que não podem ser redimensionados automáticamente (este processo deve necessáriamente ser efetuado através do método ***u32 resize(u32 bytes)***), o processo de escrita termina ao se atingir o final do arquivo. Este método retorna o número de bytes que efetivamente foram escritos no arquivo.

- **u32 append(u8 * buffer, u32 bytes)** : Grava, no final do arquivo a quantidade de bytes específicado pelo parâmetro *bytes*, copiando os dados presentes no buffer referenciado pelo parâmetro *buffer*.

- **u32 seek(u32 byte)** : Modifica a posição do ponteiro interno do arquivo, para que a próxima escrita/leitura seja efetuada a partir do byte específicado pelo parâmetro *byte*. Caso esse parâmetro seja específicado para um byte maior que a quantidade de bytes do arquivo, o método posiciona os ponteiros no final do arquivo. Este método retorna a posição efetiva em que a próxima escrita/leitura irá ocorrer.

- **u32 size()** : Retorna o tamanho do arquivo em bytes.

- **u32 resize(u32 bytes)** : Modifica o tamanho do arquivo para o tamanho específicado pelo parâmetro *bytes*. Caso o arquivo seja um arquivo de dados contínuos, uma nova área de blocos do disco é alocado para comportar o novo tamanho do arquivo, e seu conteúdo é copiado para esta nova área, desalocando a área anterior. Retorna o novo tamanho do arquivo, ou inteiro ZERO, caso não seja possível redimensionar o arquivo.

- **bool eof()** : Retorna uma váriavel booleana, indicando se o ponteiro interno do arquivo esta no final do mesmo.

3.2.8 A Família Directory

Esta família é responsável pela implementação do recursos de nomes a arquivos. Conforme pode ser visto na seção 2.3, um sistema de arquivos não precisa necessáriamente possuir um nome para representá-lo dentro do conjunto de arquivos que formam o sistema, ele pode simplesmente ser acessado através do seu índice, dentro deste conjunto. Contudo caso a específicação necessite de uma nomeação de seus arquivos, isto é realizado pelos membros desta família.

Note que os membros desta família não precisam ser utilizados apenas para implementações de sistemas de arquivos. Outros componentes que possuam a característica fundamental de um sistema de diretórios (tradução de um nome para algum outro tipo de informação) podem utilizar este componente para realizar tais funções. Exemplos de sistemas que utilizam o conceito de diretórios são sistemas de traduçao de nomes na Internet, através do protocolo *Domain Name Server (DNS)* e sistemas de bancos de dados através do protocolo *Lightweight Directory Access Protocol (LDAP)*.

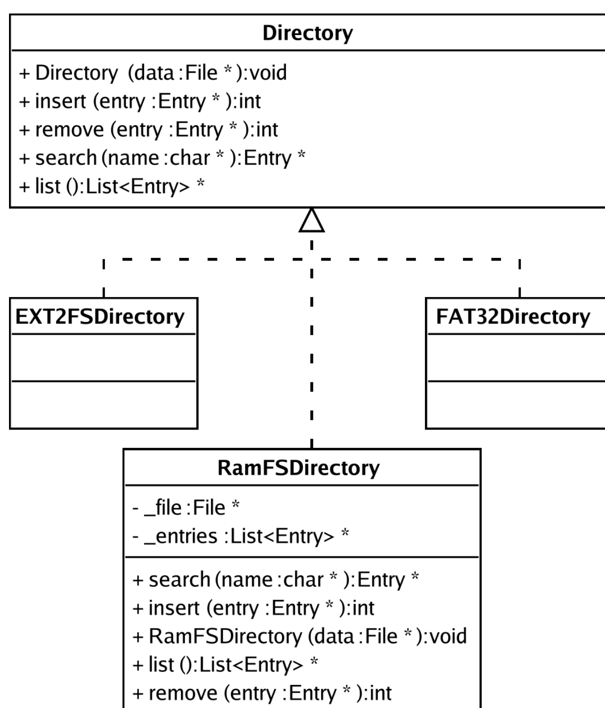

Figura 3.11: Interface inflada e membros da família Directory.

Basicamente, para cada específicação de sistema de arquivos, haverá um membro correspondente que implementa a especificação de diretórios do mesmo, conforme pode ser visto na figura 3.11. Embora não seja uma regra, uma grande maioria dos sistemas de arquivos implementam os diretórios como arquivos especiais presentes no disco. Com isso, cada sistema de arquivos irá definir uma formatação específica para o seu arquivo de diretório, e daí a necessidade de existir membros específicos a cada implementação. Abaixo apresentamos a interface inflada da família Directory.

Um sistema de diretórios é definido como um conjunto de entradas, cujo o conteúdo dessas pode variar de acordo com a aplicação do mesmo, e a estas são associados nomes que devem ser consideradas como chaves do sistema de diretórios. Em um mesmo diretório não é possível existir duas entradas com o mesmo nome.

Uma estrutura interna a cada membro da família Directory define o conteúdo de cada entrada do sistema de diretórios, sendo esta implementada internamente a cada membro da família na definição da classe *Entry*.

- **Directory(File * file)** : Abre o arquivo de diretório específicado pelo parâmetro *file*.
- **int insert(Entry * entry)** : Insere a entrada específicada pelo parâmetro *entry* no diretório, atualizado o arquivo que represeta este diretório.
- **int remove(Entry * entry)** : Remove a entrada específicada pelo parâmetro *entry* no diretório, atualizado o arquivo que represeta este diretório.
- **Entry * search(char * name)** : Efetua a busca por uma entrada específicada pelo parâmetro *name*, retornando um ponteiro para o mesmo caso ele seja localizado, ou caso contrário, um ponteiro nulo.
- **List<Entry> * list()** : Retorna uma lista de todas as entradas presentes neste diretório.

Capítulo 4

Protótipo do Sistema

O protótipo implementado para a validação da modelagem é um sistema de arquivos simples armazenado na própria memória do sistema. Suas estruturas de meta-dados são bem simples e fornecem apenas algumas operações básicas em sistemas de arquivos.

Nenhum tipo de controle de acesso, através de usuários e permissões foi especificado neste, assim como a utilização de dados de controle de tempo de criação, último acesso, etc. Basicamente os arquivos implementados neste sistema possuem apenas o atributo de tamanho e são nomeados através da abstração de diretórios.

Todo o controle de alocação de blocos do volume e índices das tabelas de descritores de arquivos são realizadas através de *Bitmaps* no volume, que possuem uma posicão fixa no mesmo e definida no superbloco do dispositivo.

O superbloco por sua vez, possue uma localização fixa no disco, estando sempre no primeiro bloco lógico do volume. A figura 4.1 mostra a disposição dos meta-dados e dados do sistema implementado.

O superbloco deste sistema possui as informações básicas para que os outros meta-dados possam ser localizados no disco. A primeira informação que o superbloco fornece é o tamanho do bloco lógico do sistema de arquivos. Este tamanho é definido pelo usuário do sistema no momento em que o mesmo está formatando o sistema de arquivos e que conforme visto na seção 2.1 é de fundamental importância

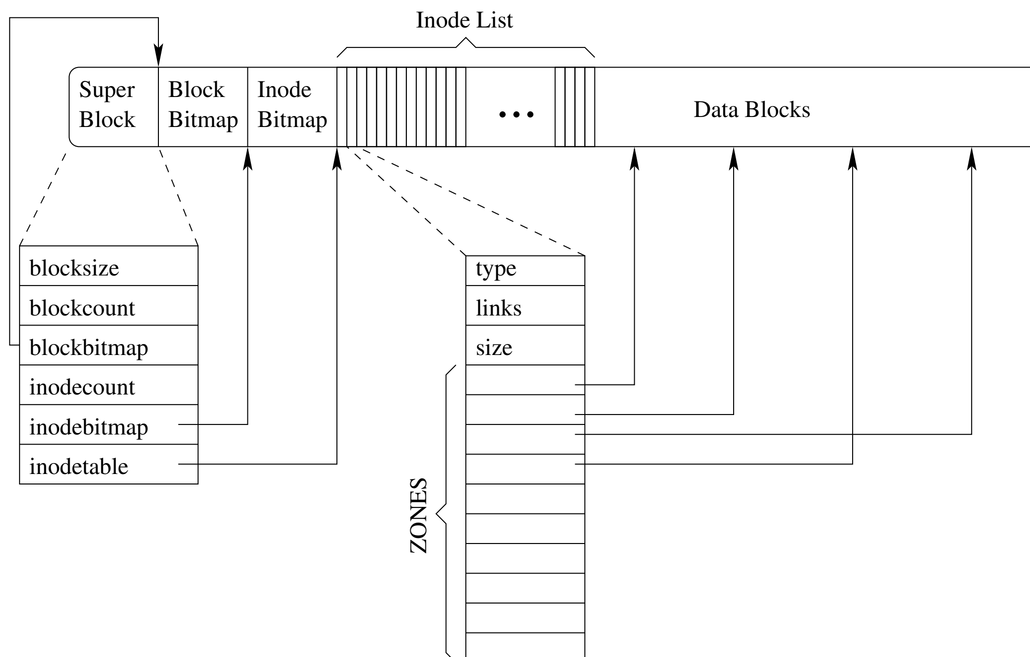

Figura 4.1: Estrutura geral da especificação implementada

a performance do sistema de arquivos.

Para ser capaz de localizar os outros meta-dados do sistema de arquivos, o superbloco possui uma referência ao bloco onde inicia os *bitmaps* de alocação de blocos, de descritores de arquivos e a tabela de descritores de arquivos. Adicionalmente o tamanho dessas estruturas (número de blocos e descritores presentes no sistema) são armazenadas no superbloco.

Os descritores de arquivos possuem um atributo do tipo de arquivo, que neste caso pode ser um arquivo de diretório ou um arquivo do usuário, além de um contador de referências a este arquivo, uma vez que o mesmo pode possuir mais de um mesmo nome em diversos diretórios distintos. Também possui um atributo que informa o tamanho do arquivo, em *bytes*.

O descritor deste sistema utiliza a alocação de blocos através de inodos, contudo não foi implementado neste protótipo a possibilidade de ocorrer referências duplas ou triplas, assim possuindo dez referências diretas a bloco de dados, este protótipo pode possuir arquivos até 10 vezes maiores que o tamanho de cada bloco lógico.

A figura 4.2 ilustra a composição de um sistema de arquivos, através da seleção dos respectivos membros de cada família no sistema de arquivos.

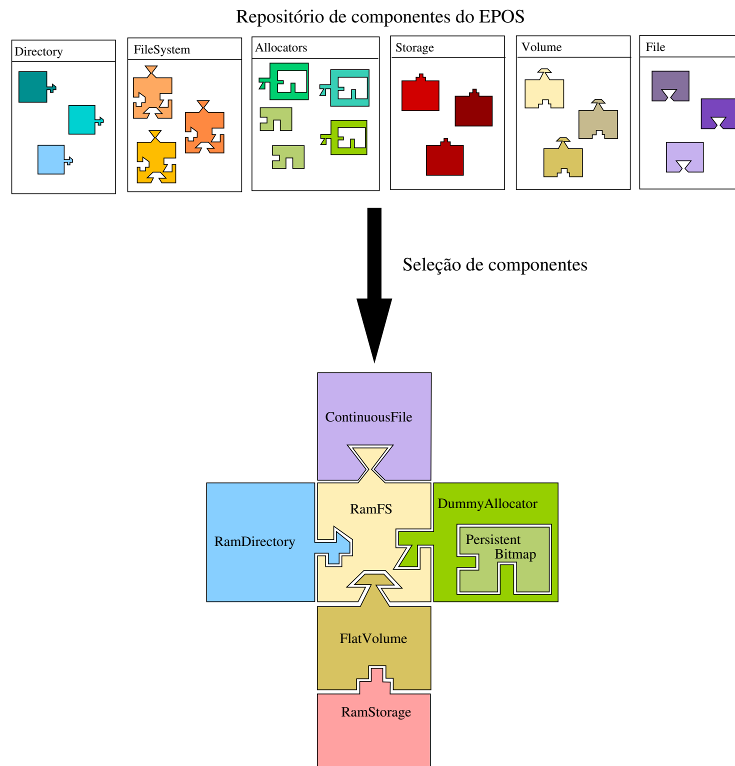

Figura 4.2: Composição de um sistema de arquivos

Capítulo 5

Conclusões

Um sistema de arquivos básico em memória RAM foi implementado dentro do sistema EPOS, oferecendo a possibilidade da aplicação armazenar e recuperar informações de uma mídia estruturada de forma organizada, mostrando-se viável a implementação de sistemas de arquivos baseados em componentes de software, através da metodologia de projeto de sistemas orientados a aplicação.

Destaco ainda como principal resultado deste trabalho, o grande favorecimento ao reuso de componentes de software decorrente da modelagem proposta, principalmente pela independência dos algoritmos presentes nos membros da família *File* e *FileSystem*, assim como a separação entre os dados utilizados pelos algoritmos de alocação de recursos e sua implementação, que é ainda mais favorecida pelo uso de meta-programação (*templates* da linguagem C++). Cabe ainda ressaltar que esses resultados puderam ser obtidos atráves da utilização das técnicas de desenvolvimento de sistemas orientados a aplicação.

Os resultados preliminares do protótipo desenvolvido geram sistemas com imagens do tamanho de cerca de 35 Kbytes, sendo que o código objeto dos componentes de sistemas de arquivos selecionados para o protótipo totalizam 19.348 bytes.

Do ponto de vista de conhecimento, a realização deste trabalho traz a comunidade científica uma proposta para a modelagem de sistemas de arquivos orientada

a componentes de software que favorece o reuso e a customização de sistemas de arquivos de acordo com as necessídades específicas da aplicação, servindo assim como ponto de partida para futuros trabalhos de pesquisa nesse tema.

Pessoalmente este trabalho certamente agregou conhecimento e experiência na área de desenvolvimento de componentes de software para sistemas computacionais básicos ao meu currículo acadêmico, abordando as áreas de metodologias para decomposição de domínios de conhecimento, modelagem orientada a objetos, linguagem C++ e programação generativa.

5.1 Trabalhos Futuros

Devido a complexidade enfrentada para a modelagem do domínio de sistemas de arquivos, que buscou principalmente atingir uma modelagem que favorecesse o reuso e composição dos componentes para o sistema de arquivos, não foi possível realizar a implementação dos membros da família *Directory*, sendo esta a primeira indicação de trabalho a ser realizado, até como uma forma natural de continuação deste.

Uma vez que o resultado deste trabalho gerou apenas um protótipo para a validação da modelagem proposta, é de fundamental importância ao projeto EPOS, que esforços sejam realizados para a implementação de mediadores para dispositivos de uso real no mercado computacional, como memórias FLASH, discos rígidos e dispositivos de leitura de mídias ópticas como CD e DVD, possibilitando assim testes de uso e maiores possibilidades para a realização de testes da modelagem.

Este trabalho também deve ser utilizado como ponto de partida para a implementação de componentes para que o sistema EPOS suporte o acesso a sistemas de arquivos existentes atualmente, como o ISO9660, UDF, EXT2 e FAT16/32, sendo de fundamental importância estratégica ao EPOS o suporte a esses sistemas.

Estudos comparativos também podem ser realizados, assim que os componentes necessários para a implementação de sistemas de arquivos existentes atualmente no mercado forem adicionados ao repositório de componentes do EPOS, podendo assim surgir novas propostas para o aperfeiçoamento da modelagem proposta.

Referências Bibliográficas

[Barr 1999]BARR, M. *Programming Embedded Systems in C and C++*. [S.l.]: O'Reilly, 1999.

[Booch 1994]BOOCH, G. *Object-Oriented Analysis and Design with Applications*. 2. ed. [S.l.]: Addison-Wesley, 1994.

[Eckel 2000]ECKEL, B. *Thinking in C++*. [S.l.]: Planet PDF, 2000.

[Fröhlich 1994]FRöHLICH, A. A. M. *Pyxis: Um sistema de arquivos distribuídos*. Florianópolis: CPGCC/UFSC, 1994. 98 p.

[Fröhlich 2001]FRöHLICH, A. A. M. *Application-Oriented Operating Systems*. 1. ed. [S.l.]: GMD - Forschungszentrum Informationstechnik, 2001.

[Fröhlich 2002]FRöHLICH, A. A. M. *EPOS: Embedded Parallel Operating System*. Florianópolis: Federal University of Santa Catarina, 2002.

[Fröhlich 2003]FRöHLICH, A. A. M. *Notas de aula - Dedicated Operational System*. 2003.

[Gamma et al. 1995]GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1995.

[Johnson e Foote 1988]JOHNSON, R. E.; FOOTE, B. Designing Reusable Classes. *Journal of Object-Oriented Programming*, v. 1, n. 2, p. 22–35, jun. 1988.

[Larman 2001]LARMAN, C. *Applying UML and Patterns*. second. [S.l.]: Unknown, 2001.

[Marwede 2003]MARWEDE, P. *Embedded System Design*. [S.l.]: Kluwer Academic Publishers, 2003. ISBN 1-4020-7690-9.

[Moore 1965]MOORE, G. E. Cramming more components onto integrated circuits. *Eletronics*, v. 38, n. 8, abr. 1965.

[Mullender e Tannembaum 1984]MULLENDER, S. J.; TANNEMBAUM, A. S. *Immendiate Files - Software: Practice and Experience*. [S.l.: s.n.], 1984. 365-597 p.

[OMG 1999]OMG. *OMG Unified Modeling Language Specification*. [S.l.], jun. 1999.

[Rosenblum e Ousterhout 1992]ROSENBLUM, M.; OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, v. 10, n. 1, p. 26–52, 1992. Disponível em: <citeseer.nj.nec.com/rosenblum91design.html>.

[Silberschatz, Galvin e Peterson 1998]SILBERSCHATZ, A.; GALVIN, P.; PETERSON, J. *Operating Systems Concepts*. 5. ed. [S.l.]: John Wiley and Sons, 1998.

[Smolik 1995]SMOLIK, T. An object-oriented file system: an example of using the class hierarchy framework concept. *ACM SIGOPS Operating Systems Review*, ACM Press, v. 29, n. 2, p. 33–53, 1995. ISSN 0163-5980.

[Stroustrup 1997]STROUSTRUP, B. *The C++ Programming Language*. 3. ed. [S.l.]: Addison-Wesley, 1997.

[Tanenbaum e Woodhull 1997]TANENBAUM, A. S.; WOODHULL, A. S. *Operational Systems - Design and Implementation*. 2. ed. [S.l.]: Prentice Hall, 1997. 939 p. ISBN 0-13-638677-6.

Apêndice A

Código Fonte

```
001 // EPOS Storage Common Package
002 //
003 // Author: Hugo Marcondes
004 // Documentation: $EPOS/doc/storage      Date: 17 Aug 2004
005
006 #ifndef __storage_common_h
007 #define __storage_common_h
008
009 #include <system/config_defs.h>
010
011 __BEGIN_SYS
012
013 class Storage_Common: public __INT(Storage_Common)
014 {
015 protected:
016     Storage_Common() {}
017
018 public:
019     // Storage common methods
020
021 private:
022     // Storage common attributes
023 };
024
025 __END_SYS
026
027 #endif
028
029
030 // EPOS RamStorage Declarations
031 //
032 // Author: Hugo Marcondes
033 // Documentation: $EPOS/doc/storage      Date: 28 Jul 2004
034
035 #ifndef __ramstorage_h
036 #define __ramstorage_h
037
038 #include <storage.h>
039 #include "common.h"
040
041 __BEGIN_SYS
042
043 class RamStorage: public __INT(Storage), protected Storage_Common
044 {
045 private:
```

```
046     typedef Traits<RamStorage> Traits;
047     static const Type_Id TYPE = Type<RamStorage>::TYPE;
048
049     // RamStorage private imports, types and constants
050
051 public:
052     RamStorage(unsigned int unit = 0);
053     ~RamStorage();
054     unsigned int size();
055     unsigned int blocksize();
056     int get_block(unsigned int block, unsigned char * buffer, unsigned int quantity =
1);
057     int put_block(unsigned int block, unsigned char * buffer, unsigned int quantity =
1);
058
059     static int init(System_Info *si);
060
061 private:
062     // RamStorage implementation methods
063
064 private:
065     // RamStorage attributes
066     unsigned int _unit;
067     unsigned int _size;
068     unsigned int _blocksize;
069     unsigned char _data[Traits::SIZE];
070
071 };
072
073 __END_SYS
074
075 #endif
076
077
078 // EPOS RamStorage Implementation
079 //
080 // Author: Hugo Marcondes
081 // Documentation: $EPOS/doc/storage      Date: 28 Jul 2004
082
083 #include <mediator/storage/ramstorage.h>
084
085 __BEGIN_SYS
086
087 // Class attributes
088 // type RamStorage::attribute;
089
090 // Constructors
091 RamStorage::RamStorage(unsigned int unit)
092 {
093     db<RamStorage>(TRC) << "RamStorage(unit= " << unit << ")\n";
094     db<RamStorage>(TRC) << "Creating with " << Traits::SIZE << " bytes (" <<
Traits::BLOCKSIZE << ")\n";
095     _unit = unit;
096     _size = Traits::SIZE;
097     _blocksize = Traits::BLOCKSIZE;
098 }
099
100 RamStorage::~RamStorage()
101 {
102     db<RamStorage>(TRC) << "~RamStorage()\n";
103 }
104
105 // Methods
106
107 // Class methods
108 unsigned int RamStorage::size()
109 {
```

```
110     db<RamStorage>(TRC) << "RamStorage::size()\n";
111     return _size;
112 }
113
114 unsigned int RamStorage::blocksize()
115 {
116     db<RamStorage>(TRC) << "RamStorage::blocksize()\n";
117     return _blocksize;
118 }
119
120 int RamStorage::get_block(unsigned int block, unsigned char * buffer, unsigned int
quantity)
121 {
122     db<RamStorage>(TRC) << "RamStorage::get_block(block= " << block << ", buffer= "
<< buffer << ", quantity= " << quantity << ")\n";
123     unsigned int target = block * _blocksize;
124     unsigned int aux = quantity * _blocksize;
125     if((target+aux) > _size) { return -1; }
126     while(aux--){
127       buffer[aux] = _data[target+aux];
128     }
129     return quantity * _blocksize;
130 }
131
132 int RamStorage::put_block(unsigned int block, unsigned char * buffer, unsigned int
quantity)
133 {
134     db<RamStorage>(TRC) << "RamStorage::put_block(block= " << block << ", buffer= "
<< buffer << ", quantity= " << quantity << ")\n";
135     unsigned int target = block * _blocksize;
136     unsigned int aux = quantity * _blocksize;
137     if((target+aux) > _size) { return -1; }
138     while(aux--){
139       _data[target+aux] = buffer[aux];
140     }
141     return quantity * _blocksize;
142 }
143
144 __END_SYS
145
146
147 // EPOS RamStorage Initialization
148 //
149 // Author: Hugo Marcondes
150 // Documentation: $EPOS/doc/storage      Date: 28 Jul 2004
151
152 #include <mediator/storage/ramstorage.h>
153
154 __BEGIN_SYS
155
156 // Class initialization
157 int RamStorage::init(System_Info * si)
158 {
159     db<RamStorage>(TRC) << "RamStorage::init()\n";
160
161     return 0;
162 }
163
164 __END_SYS
165
166
167 // EPOS RamStorage Test Program
168 //
169 // Author: Hugo Marcondes
170 // Documentation: $EPOS/doc/storage      Date: Sun May 30 18:56:47 BRT 2004
171
172 #include <utility/ostream.h>
```

```
173 #include <storage.h>
174 #include <framework.h>
175
176 __USING_SYS
177
178 int main()
179 {
180     OStream cout;
181
182     cout << "RamStorage test\n";
183
184     RamStorage storage(1);
185
186     unsigned int blocksize = storage.blocksize();
187     unsigned int size = storage.size();
188
189     cout << "Blocksize " << blocksize << " Size " << size << "\n";
190
191     cout << "Filling blocks with his own block number ...\n";
192
193     unsigned char * buffer = new unsigned char[blocksize];
194
195     unsigned char data = 0x00;
196
197     for(unsigned int z=0 ; z < (size / blocksize); z++){
198       for(unsigned int i=0; i < blocksize; i++){
199   buffer[i] = data;
200       }
201       int result = storage.put_block(z, buffer);
202       if(result == -1){
203   cout << "Error writing on device. -> block:" << (int)z << "\n";
204       }
205       data++;
206     }
207
208     cout << "Checking data written ...\n";
209
210     int total_errors = 0;
211
212     data = 0x00;
213     unsigned char * read_buffer = new unsigned char[blocksize];
214
215     for(unsigned int z=0 ; z < (size / blocksize); z++){
216       int result = storage.get_block(z, read_buffer);
217       if(result == -1){
218   cout << "Error reading block " << (int)z << "\n";
219       }
220       for(unsigned int i = 0; i < blocksize; i++){
221   if(read_buffer[i] != data){
222     //cout << (int)read_buffer[i] << ",";
223     total_errors++;
224   }
225       }
226       data++;
227     }
228
229     cout << "Done! " << total_errors << " errors occurred.\n";
230
231     return 0;
232
233 }
234
235
236 // EPOS Storage Common Package Implementation
237 //
238 // Author: Hugo Marcondes
239 // Documentation: $EPOS/doc/storage      Date: 17 Aug 2004
```

```
240
241 #include <storage.h>
242 #include <mediator/storage/common.h>
243
244 __BEGIN_SYS
245
246 // Class attribute
247 // type Storage_Common::attribute;
248
249 // Methods
250
251 // Class methods
252
253 __END_SYS
254
255
256 // EPOS Volume Common Package
257 //
258 // Author: Hugo Marcondes
259 // Documentation: $EPOS/doc/volume      Date: 18 Aug 2004
260
261 #ifndef __volume_common_h
262 #define __volume_common_h
263
264 #include <system/config_defs.h>
265
266 __BEGIN_SYS
267 __BEGIN_IMP
268
269 class Volume_Common: public __INT(Volume_Common)
270 {
271 protected:
272     Volume_Common() {}
273
274 public:
275     // Volume common methods
276
277 private:
278     // Volume common attributes
279 };
280
281 __END_IMP
282 __END_SYS
283
284 #endif
285
286
287 // EPOS FlatVolume Declarations
288 //
289 // Author: Hugo Marcondes
290 // Documentation: $EPOS/doc/volume      Date: 18 Aug 2004
291
292 #ifndef __flatvolume_h
293 #define __flatvolume_h
294
295 #include <volume.h>
296 #include <storage.h>
297 #include "common.h"
298
299
300 __BEGIN_SYS
301 __BEGIN_IMP
302
303 class FlatVolume: public __INT(FlatVolume), protected Volume_Common
304 {
305 private:
306     typedef Traits<FlatVolume> Traits;
```

```
307     static const Type_Id TYPE = Type<FlatVolume>::TYPE;
308
309     // FlatVolume private imports, types and constants
310
311 public:
312     class VolumeTable { };
313
314 public:
315     FlatVolume(Storage * storage);
316     ~FlatVolume();
317     int get_block(unsigned int block, unsigned char * buffer);
318     int put_block(unsigned int block, unsigned char * buffer);
319     unsigned int blocksize();
320     void blocksize(unsigned int value);
321     unsigned int size();
322
323     static int init(System_Info *si);
324
325 private:
326     // FlatVolume implementation methods
327
328 private:
329     // FlatVolume attributes
330     Storage * _storage;
331     unsigned int _start;
332     unsigned int _end;
333     unsigned int _blocksize;
334
335 };
336
337 __END_IMP
338 __END_SYS
339
340 #endif
341
342
343 // EPOS FlatVolume Implementation
344 //
345 // Author: Hugo Marcondes
346 // Documentation: $EPOS/doc/volume      Date: 18 Aug 2004
347
348 #include <abstraction/volume/flatvolume.h>
349
350 __BEGIN_SYS
351 __BEGIN_IMP
352
353 // Class attributes
354 // type FlatVolume::attribute;
355
356 // Constructors
357 FlatVolume::FlatVolume(Storage * storage)
358 {
359     db<FlatVolume>(TRC) << "FlatVolume(storage= " << storage << ")\n";
360     _storage = storage;
361     _start = 0; //First Block Index
362     _blocksize = _storage->blocksize();//Initially is equal Storage, FS will set
this on mount.
363     _end = (_storage->size() / _blocksize) - 1;//Last Block Index
364 }
365
366 FlatVolume::~FlatVolume()
367 {
368     db<FlatVolume>(TRC) << "~FlatVolume()\n";
369 }
370
371 // Methods
372 int FlatVolume::get_block(unsigned int block, unsigned char * buffer)
```

```
373 {
374     db<FlatVolume>(TRC) << "FlatVolume::get_block(block= " << block << ", buffer= "
<< buffer << ")\n";
375     unsigned int blockratio = _blocksize / _storage->blocksize();
376     unsigned int target = block * blockratio;
377     if(target > _end){ return -1; }
378     return _storage->get_block(target, buffer, blockratio);
379 }
380
381 int FlatVolume::put_block(unsigned int block, unsigned char * buffer)
382 {
383     db<FlatVolume>(TRC) << "FlatVolume::put_block(block= " << block << ", buffer= "
<< buffer << ")\n";
384     unsigned int blockratio = _blocksize / _storage->blocksize();
385     unsigned int target = block * blockratio;
386     if(target > _end){ return -1; }
387     return _storage->put_block(target, buffer, blockratio);
388 }
389
390 unsigned int FlatVolume::blocksize()
391 {
392     db<FlatVolume>(TRC) << "FlatVolume::blocksize()\n";
393     return _blocksize;
394 }
395
396 void FlatVolume::blocksize(unsigned int value)
397 {
398     db<FlatVolume>(TRC) << "FlatVolume::blocksize(value= " << value << ")\n";
399     _blocksize = value;
400 }
401
402 unsigned int FlatVolume::size()
403 {
404     db<FlatVolume>(TRC) << "FlatVolume::size()\n";
405     return ((_end-_start) + 1)/(_blocksize / _storage->blocksize());
406 }
407
408 // Class methods
409 __END_IMP
410 __END_SYS
411
412
413 // EPOS FlatVolume Initialization
414 //
415 // Author: Hugo Marcondes
416 // Documentation: $EPOS/doc/volume      Date: 18 Aug 2004
417
418 #include <abstraction/volume/flatvolume.h>
419
420 __BEGIN_SYS
421 __BEGIN_IMP
422
423 // Class initialization
424 int FlatVolume::init(System_Info * si)
425 {
426     db<FlatVolume>(TRC) << "FlatVolume::init()\n";
427
428     return 0;
429 }
430
431 __END_IMP
432 __END_SYS
433
434
435 // EPOS FlatVolume Test Program
436 //
437 // Author: Hugo Marcondes
```

```
438 // Documentation: $EPOS/doc/volume      Date: 18 Aug 2004
439
440 #include <utility/ostream.h>
441 #include <storage.h>
442 #include <volume.h>
443 #include <framework.h>
444
445 __USING_SYS
446
447 int main()
448 {
449     OStream cout;
450
451     cout << "FlatVolume test\n";
452
453     Storage ram(1);
454     FlatVolume volume(&ram);
455
456     unsigned int blocksize = volume.blocksize() * 2;
457     volume.blocksize(blocksize);
458     unsigned int size = volume.size();
459
460     cout << "Blocksize " << blocksize << " Size " << size << "\n";
461
462     cout << "Filling blocks with his own block number ...\n";
463
464     unsigned char * buffer = new unsigned char[blocksize];
465
466     unsigned char data = 0x00;
467
468     for(unsigned int z=0 ; z < size; z++){
469       for(unsigned int i=0; i < blocksize; i++){
470   buffer[i] = data;
471       }
472       int result = volume.put_block(z, buffer);
473       if(result == -1){
474   cout << "Error writing on volume. (block:" << (int)z << ")\n";
475       }
476       data++;
477     }
478
479     cout << "Checking data written ...\n";
480
481     int total_errors = 0;
482
483     data = 0x00;
484
485     for(unsigned int z=0 ; z < blocksize ; z++){
486   buffer[z] = 0xFF;
487     }
488
489     for(unsigned int z=0 ; z < size ; z++){
490       int result = volume.get_block(z, buffer);
491       if(result == -1){
492   cout << "Error reading block " << (int)z << "\n";
493       }
494       for(unsigned int i = 0; i < blocksize; i++){
495   if(buffer[i] != data){
496     total_errors++;
497   }
498       }
499       data++;
500     }
501
502     cout << "Done! " << total_errors << " errors occurred.\n";
503
504     return 0;
```

```
505 }
506
507
508 // EPOS Volume Common Package Implementation
509 //
510 // Author: Hugo Marcondes
511 // Documentation: $EPOS/doc/volume      Date: 18 Aug 2004
512
513 #include <volume.h>
514 #include <abstraction/volume/common.h>
515
516 __BEGIN_SYS
517 __BEGIN_IMP
518
519 // Class attribute
520 // type Volume_Common::attribute;
521
522 // Methods
523
524 // Class methods
525
526 __END_IMP
527 __END_SYS
528
529
530 // EPOS Generic Dummy Allocator Utility Declarations
531 //
532 // Author: Hugo Marcondes
533 // Documentation: $EPOS/doc/container      Date: 30 Sep 2004
534
535 #ifndef __dummy_allocator_h
536 #define __dummy_allocator_h
537
538 #include <system/config.h>
539
540 __BEGIN_SYS
541 __BEGIN_IMP
542
543 template<class AllocData>
544 class DummyAllocator {
545
546  private:
547   AllocData * _data;
548   void * _free; //Set on contructor with data->first(); Remove when impl.
549
550  public:
551
552   DummyAllocator(AllocData * arg) {
553     _data = arg;
554     _free = _data->first();
555     if(_data->state(_free)){
556       nextfree();
557     }
558   }
559
560   void * alloc(){ // Critical region ? Remove when impl.
561     if(!(_free == 0)){
562        void * ptr = _free;
563        _data->set(ptr, true);
564        nextfree();
565        return ptr;
566     }
567     return _free;
568   }
569
570   void free(void * ptr){ // Critical region ? Remove when impl.
571     _data->set(ptr, false);
```

```
572     if(_free == 0) { _free = ptr; }
573   }
574
575   void * alloc(int size){ // Critical region ? Remove when impl.
576     if(_free != 0){
577   int alloc = size - 1;
578   void * test_ptr = _data->next(_free);
579   void * ptr = _free;
580         bool tmp = false;
581         void * tmp_ptr = _free;
582   while(alloc > 0){ //Main search loop
583           if(tmp && (test_ptr == tmp_ptr)) {//If already searched before and after
free - return NULL NO SPACE FOUND!
584       return 0;
585     }
586           void * tmp_ptr = _data->next(_data->last());
587     if(test_ptr == tmp_ptr) { //THE END ... Go to the Begining
588       tmp = true;
589       test_ptr = _data->first();
590       ptr = test_ptr;
591       alloc = size - 1;
592     } else if(_data->state(test_ptr)){ //If test_ptr is used. Go next.
593       test_ptr = _data->next(test_ptr);
594       ptr = test_ptr; //make the next position our new candidate for alloc;
595       alloc = size - 1; //reset the pos_alloc counter;
596     } else { //Check Next, decrement alloc position.
597       test_ptr = _data->next(test_ptr);
598       alloc--;
599           }
600   }
601   //Needed ?? - if ((alloc != 0) && ((pos_test - alloc_candidate) != size)) { return
(void *)-1; } //sanity check
602   //Find the new first_free, starts from old first_free
603   //Preserves old first_free if the space is not allocated on first_free.
604   tmp_ptr = ptr;
605   while(size--){ // bitmap marks.
606     _data->set(tmp_ptr, true);
607     tmp_ptr = _data->next(tmp_ptr);
608   }
609         nextfree();
610   return ptr;
611     } else {
612       return 0;
613     }
614   }
615
616   void free(void * position, int size){ // Critical region ?
617     if(_free == 0) { _free = position; }
618     while(size--){
619       _data->set(position, false);
620       position = _data->next(position);
621     }
622   }
623
624   void alloc(void ** list, int size){ //TO DO
625   list[0] = (void *)-1;
626   }
627
628   void free(void ** list, int size){ // TO DO
629   }
630
631   ~DummyAllocator(){
632     _data->~AllocData();
633     kfree(_data);
634   }
635
636  private:
```

```
637
638   void nextfree(){
639     bool found = false;
640     void * initial = _free;
641     for(; _free <= _data->last(); _free = _data->next(_free)){
642       if(!(_data->state(_free))) { found = true; break; }
643     }
644     if(!found){
645       _free=_data->first();
646       for(; _free <= initial; _free = _data->next(_free)) {
647     if(!(_data->state(_free))) { found = true; break; }
648       }
649     }
650     if(!found){
651       _free = 0;
652     }
653   }
654
655 };
656
657 __END_IMP
658 __END_SYS
659
660 #endif
661
662
663 // EPOS Generic Dummy Allocator Utility Implementation
664 //
665 // Author: Hugo Marcondes
666 // Documentation: $EPOS/doc/allocator      Date: 30 Sep 2003
667
668 #include <utility/dummy_allocator.h>
669
670 __BEGIN_SYS
671 __END_SYS
672
673
674 // EPOS Bitmap Allocator Utility Test Program
675 //
676 // Author: Hugo Marcondes
677 // Documentation: $EPOS/doc/bitmap_allocator           Date: 18 Jun 2004
678
679 #include <utility/ostream.h>
680 #include <utility/dummy_allocator.h>
681 #include <utility/bitmap.h>
682
683 __USING_SYS;
684 __USING_IMP;
685
686 OStream cout;
687
688 class A {
689  public:
690
691 static void print_status(Bitmap * bitmap){
692   for(int i=1; i <= (int)bitmap->last(); i++){
693     if(bitmap->state((void *)i)){
694       cout << "1";
695     } else {
696       cout << "0";
697     }
698   }
699   cout << "\n";
700 }
701
702 static int aloca(DummyAllocator<Bitmap> * allocator){
703   cout << "alloc();\n";
```

```
704   int pos;
705   pos = (int)allocator->alloc();
706   if(pos < 1){
707     cout << "No space left!\n";
708   }
709   return pos;
710 }
711
712 static int aloca(DummyAllocator<Bitmap> * allocator, int size){
713   cout << "alloc(size=" << size << ");\n";
714   int pos;
715   pos = (int)allocator->alloc(size);
716   if(pos < 1){
717     cout << "No space left!\n";
718   }
719   return pos;
720 }
721
722 };
723
724 int main(){
725
726   cout << "Dummy Allocator Utility Test\n";
727
728   Bitmap bitmap(15);
729   DummyAllocator<Bitmap> alloc(&bitmap);
730
731   int size = bitmap.size();
732   cout << "\nBitmap Size -> " << size << "\n\n";
733
734   A::print_status(&bitmap);
735
736   for(int i=0; i < 7; i++){
737     A::aloca(&alloc);
738   }
739
740   A::print_status(&bitmap);
741
742   alloc.free((void *)2);
743   cout << "free(pos=2);\n";
744   A::print_status(&bitmap);
745
746   alloc.free((void *)4);
747   cout << "free(pos=4);\n";
748   A::print_status(&bitmap);
749
750   A::aloca(&alloc);
751   A::print_status(&bitmap);
752
753   alloc.free((void *)1);
754   cout << "free(pos=1);\n";
755   A::print_status(&bitmap);
756
757   A::aloca(&alloc);
758   A::print_status(&bitmap);
759
760   A::aloca(&alloc, 3);
761   A::print_status(&bitmap);
762
763   A::aloca(&alloc, 3);
764   A::print_status(&bitmap);
765
766   A::aloca(&alloc);
767   A::print_status(&bitmap);
768
769   A::aloca(&alloc);
770   A::print_status(&bitmap);
```

```
771
772   A::aloca(&alloc);
773   A::print_status(&bitmap);
774
775   A::aloca(&alloc);
776   A::print_status(&bitmap);
777
778   A::aloca(&alloc, 30);
779   A::aloca(&alloc, 5);
780
781   A::print_status(&bitmap);
782
783   alloc.free((void *)3, 5);
784   cout << "free(pos=3, size=5);\n";
785   A::print_status(&bitmap);
786
787   alloc.free((void *)10, 2);
788   cout << "free(pos=10, size=2);\n";
789   A::print_status(&bitmap);
790
791   A::aloca(&alloc, 2);
792   A::print_status(&bitmap);
793
794 }
795
796
797 // EPOS FileSystem Common Package
798 //
799 // Author: Hugo Marcondes
800 // Documentation: $EPOS/doc/filesystem      Date: 25 Aug 2004
801
802 #ifndef __filesystem_common_h
803 #define __filesystem_common_h
804
805 #include <system/config_defs.h>
806
807 __BEGIN_SYS
808 __BEGIN_IMP
809
810 class FileSystem_Common: public __INT(FileSystem_Common)
811 {
812 protected:
813     FileSystem_Common() {}
814
815 public:
816     // FileSystem common methods
817
818 private:
819     // FileSystem common attributes
820 };
821
822 __END_IMP
823 __END_SYS
824
825 #endif
826
827
828 // EPOS RamFS Declarations
829 //
830 // Author: Hugo Marcondes
831 // Documentation: $EPOS/doc/filesystem      Date: 25 Aug 2004
832
833 #ifndef __ramfs_h
834 #define __ramfs_h
835
836 #include <filesystem.h>
837 #include <volume.h>
```

```
838 #include <utility/dummy_allocator.h>
839 #include <utility/bitmap_allocator.h>
840 #include "common.h"
841
842 __BEGIN_SYS
843 __BEGIN_IMP
844
845 class RamFS: public __INT(RamFS), protected FileSystem_Common
846 {
847 private:
848     typedef Traits<RamFS> Traits;
849     static const Type_Id TYPE = Type<RamFS>::TYPE;
850
851     // RamFS private imports, types and constants
852     struct ramfs_sb {
853       unsigned int block_size;
854       unsigned int block_count;
855       unsigned int block_free;
856       unsigned int block_bitmap;
857       unsigned int first_data;
858       unsigned int inode_count;
859       unsigned int inode_free;
860       unsigned int inode_bitmap;
861       unsigned int inode_table;
862     };
863
864 public:
865
866    class FileDescriptor {
867
868      public:
869        enum Type {
870          EMPTY = 0x0000,
871          FILE = 0x0001,
872          DIRECTORY = 0x0002,
873        };
874
875      //private:
876        struct FileDescriptorData{
877    unsigned short type;
878    unsigned short links;
879    unsigned short size;
880    unsigned short zones[10];
881        };
882
883
884      private:
885        RamFS * _fs;
886        unsigned int _index;
887        unsigned int _block;
888        unsigned int _offset;
889        FileDescriptorData _data;
890
891      public:
892
893        static FileDescriptor * create(FileSystem * fs, unsigned int type, unsigned
int size, int index, bool continuous = false){
894    kout << "Creating one fdesc.\n";
895     if(index == 0){
896       index = fs->alloc_fdesc();
897       if(index == -1){
898         db<RamFS>(ERR) << "Couldn't allocate a index on fdesc table.\n";
899         return 0;
900       }
901           }
902           FileDescriptorData data;
903           data.type = (unsigned short)type;
```

```
904           data.links = 0;
905           //Based on size, allocdata for this new inode.
906     unsigned int fblocks = (size + (fs->volume()->blocksize() -
1))/fs->volume()->blocksize();
907           if(fblocks > 10){
908       db<RamFS>(ERR) << "Initial size too large.\n";
909             fs->free_fdesc(index);
910       return 0;
911     }
912           data.size = (unsigned short)size;
913           for(int i = 0; i < 10; i++){
914       data.zones[i] = 0;
915     }
916     int allocblock;
917           if(size){
918             if(continuous){
919         //make continuous allocation.
920               allocblock = fs->alloc_block(fblocks);
921               if(allocblock == -1){
922           db<RamFS>(ERR) << "Couldn't allocate continuous blocks for fdesc.\n";
923                 fs->free_fdesc(index);
924                 return 0;
925               }
926         data.zones[0] = allocblock;
927         for(unsigned int i = 1; i < fblocks; i++){
928           data.zones[i] = data.zones[0] + i;
929               }
930             } else {
931               //whatever allocation :)
932           for(unsigned int i=0; i < fblocks; i++){
933           allocblock = fs->alloc_block();
934           if(allocblock == -1){ //Rolling back transaction NO more space on fs!
935       while(i--){
936         fs->free_block(data.zones[i]);
937                   }
938                   db<RamFS>(ERR) << "Couldn't allocate blocks for one fdesc.\n";
939                   fs->free_fdesc(index);
940       return 0;
941                 }
942     data.zones[i] = allocblock;
943         }
944       }
945           }
946
947           //Saving fdesc on FS.
948           unsigned int block = fs->fdesc_table() +
((sizeof(FileDescriptorData)*index) / fs->volume()->blocksize());
949     unsigned int offset = (sizeof(FileDescriptorData) * index) %
fs->volume()->blocksize();
950     unsigned char * buffer = reinterpret_cast<unsigned
char*>(kmalloc(fs->volume()->blocksize()));
951     fs->volume()->get_block(block, buffer);
952     int aux = sizeof(FileDescriptorData);
953           //TO DO: Convert endian of data.
954     while(aux--){
955     buffer[offset+aux] = ((unsigned char *)&data)[aux];
956     }
957     fs->volume()->put_block(block, buffer);
958     kfree(buffer);
959
960           //Returning pointer to the new allocated fdesc.
961           return new(kmalloc(sizeof(FileDescriptor))) FileDescriptor(fs, index);
962        }
963
964        static void remove(FileSystem * fs, unsigned int index) {
965    unsigned int block = fs->fdesc_table() + ((sizeof(FileDescriptorData)*index) /
fs->volume()->blocksize());
```

```
966    unsigned int offset = (sizeof(FileDescriptorData) * index) %
fs->volume()->blocksize();
967
968    unsigned char * buffer = reinterpret_cast<unsigned
char*>(kmalloc(fs->volume()->blocksize()));
969
970    fs->volume()->get_block(block, buffer);
971
972          FileDescriptorData * data;
973    data = (FileDescriptorData *)&buffer[offset];
974
975    unsigned int blocks = (data->size +
(fs->volume()->blocksize()-1))/fs->volume()->blocksize();
976    data->type = 0;
977    data->links = 0;
978    data->size = 0;
979
980    while(blocks--){
981      fs->free_block(data->zones[blocks]);
982      data->zones[blocks] = 0;
983    }
984
985    fs->volume()->put_block(block, buffer);
986
987    fs->free_fdesc(index);
988
989          kfree(buffer);
990        }
991
992        FileDescriptor(FileSystem * fs, unsigned int idx){
993          _index = idx;
994    _fs = fs;
995    _block = fs->fdesc_table() + ((sizeof(FileDescriptorData)*_index) /
_fs->volume()->blocksize());
996    _offset = (sizeof(FileDescriptorData) * _index) % _fs->volume()->blocksize();
997    unsigned char * buffer = reinterpret_cast<unsigned
char*>(kmalloc(_fs->volume()->blocksize()));
998    _fs->volume()->get_block(_block, buffer);
999    int aux = sizeof(FileDescriptorData);
1000    while(aux--){
1001      ((unsigned char *)&_data)[aux] = buffer[_offset+aux];
1002    }
1003    kfree(buffer);
1004    //TO DO: Make conversions from network -> host on physicaldata.
1005        }
1006
1007        unsigned int index(){ return _index; }
1008        unsigned int type(){ return _data.type; }
1009        void type(unsigned int type){ _data.type = (unsigned short)type; flush(); }
1010        void inc_links(){ _data.links++; flush(); }
1011        void dec_links(){ _data.links--; flush(); }
1012        unsigned int size(){ return _data.size; }
1013
1014        void size(unsigned int size, bool continuous = false){//Think about resize()
method on file family.
1015          int newsizeblocks = (size + (_fs->volume()->blocksize() -
1))/_fs->volume()->blocksize();
1016    int oldsizeblocks = (_data.size + (_fs->volume()->blocksize() -
1))/_fs->volume()->blocksize();
1017          if(newsizeblocks > 10){
1018      db<RamFS>(ERR) << "New size too large.\n";
1019      return;
1020    }
1021          if(newsizeblocks < oldsizeblocks){
1022      for(int i = newsizeblocks; i < oldsizeblocks; i++){
1023        _fs->free_block(_data.zones[i]);
1024              _data.zones[i] = 0;
```

```
1025      }
1026    } else {
1027            if(continuous){
1028        int allocblock;
1029        allocblock = _fs->alloc_block((unsigned)newsizeblocks);
1030              if(allocblock == -1){
1031          db<RamFS>(ERR) << "No space on fs for new size.\n";
1032                return;
1033        }
1034              //Copy data for new destination.
1035              unsigned char * buffer = reinterpret_cast<unsigned char
*>(kmalloc(_fs->volume()->blocksize()));
1036        for(int i = 0; i < oldsizeblocks; i++){
1037          getdata(i, buffer);
1038          int  res = _fs->volume()->put_block((allocblock + i), buffer);//Check
errors.
1039                if(res == -1){ //Could copy file... Aborting resize.
1040                  db<RamFS>(ERR) << "Error copying data for new destination on
descriptor resize.\n";
1041      _fs->free_block((unsigned)allocblock, (unsigned)newsizeblocks);
1042      return;
1043                }
1044        }
1045              kfree(buffer);
1046              //Free old blocks.
1047        for(int i = 0; i < oldsizeblocks; i++){
1048          _fs->free_block((unsigned int)_data.zones[i]);
1049        }
1050              //Save new blocks.
1051              for(int i = 0; i < newsizeblocks; i++){
1052          _data.zones[i] = allocblock + i;
1053        }
1054      } else {
1055        int allocblock;
1056        for(int i = oldsizeblocks; i < newsizeblocks; i++){
1057          allocblock = _fs->alloc_block();
1058          if(allocblock == -1){
1059      while(i-- > oldsizeblocks){
1060        _fs->free_block(_data.zones[i]);
1061      }
1062      db<RamFS>(ERR) << "No space left on fs.\n";
1063      return;
1064          }
1065          _data.zones[i] = allocblock;
1066        }
1067      }
1068    }
1069    _data.size = size;
1070    flush();
1071        }
1072
1073        int getdata(unsigned int index, unsigned char * data){
1074          if(index > 9) { return 0; }
1075    if(_data.zones[index] == 0) { return 0; }
1076    return _fs->volume()->get_block(_data.zones[index], data);
1077        }
1078
1079        int putdata(unsigned int index, unsigned char * data){
1080          if(index > 9) { return 0; }
1081    if(_data.zones[index] == 0) { return 0; }
1082    return _fs->volume()->put_block(_data.zones[index], data);
1083        }
1084
1085      private:
1086        void flush(){
1087    //TO DO: Make conversions from host -> network on physicaldata.
1088    unsigned char * buffer = reinterpret_cast<unsigned
```

```
char*>(kmalloc(_fs->volume()->blocksize()));
1089    _fs->volume()->get_block(_block, buffer);
1090    int aux = sizeof(FileDescriptorData);
1091    while(aux--){
1092     buffer[_offset+aux] = ((unsigned char *)&_data)[aux];
1093    }
1094    _fs->volume()->put_block(_block, buffer);
1095    kfree(buffer);
1096        }
1097    };
1098
1099    class AllocableData {
1100
1101      private:
1102        Volume * _volume;
1103        unsigned int _location;
1104        unsigned int _size;
1105        unsigned char * _data;
1106        unsigned int _data_size;
1107
1108      public:
1109        AllocableData(Volume * volume, unsigned int location, unsigned int size){
1110    _volume = volume;
1111    _location = location;
1112    _size = size;
1113    _data_size = (size+7)/8;
1114    _data = reinterpret_cast<unsigned char *>(kmalloc(_data_size));
1115    unsigned int blocksize = _volume->blocksize();
1116          unsigned int number_of_blocks = (_data_size + blocksize - 1 )/blocksize;
1117    unsigned char * buffer = reinterpret_cast<unsigned char
*>(kmalloc(number_of_blocks * blocksize));
1118    while(number_of_blocks--){
1119      unsigned int target = number_of_blocks * blocksize;
1120      _volume->get_block(location+number_of_blocks, &buffer[target]);
1121    }
1122    unsigned int tmp = _data_size;
1123    while(tmp--){
1124      _data[tmp] = buffer[tmp];
1125    }
1126        }
1127
1128        unsigned int size() { return _size; }
1129
1130        void * set(void * entry, bool state) {
1131    unsigned int pos = reinterpret_cast<unsigned int>(entry);
1132    pos--;
1133    if(state){
1134      _data[pos/8] |= (0x80 >> pos%8);
1135    } else {
1136      _data[pos/8] &= ~(0x80 >> pos%8);
1137    }
1138    flush(pos);
1139    return entry;
1140        }
1141
1142        bool state(void * entry) {
1143    unsigned int pos = reinterpret_cast<unsigned int>(entry);
1144    pos--;
1145    if(pos >= _size) { return true; }
1146    if ((_data[pos/8] & (0x80 >> pos%8)) == 0x00){;
1147      return false;
1148    } else {
1149      return true;
1150    }
1151        }
1152
1153        void * next(void * entry) {
```

```
1154          return (void *)((char *)entry + 1);
1155        }
1156
1157        void * first(){
1158    return reinterpret_cast<void *>(1);
1159        }
1160
1161        void * last(){
1162    return reinterpret_cast<void *>(_size);
1163        }
1164
1165        ~AllocableData(){
1166          kfree(_data);
1167        }
1168
1169      private:
1170
1171        void flush(unsigned int pos){
1172    unsigned int target_block, blocksize;
1173    blocksize = _volume->blocksize();
1174    target_block = pos / (8 * blocksize);
1175    unsigned char * buffer = reinterpret_cast<unsigned char *>(kmalloc(blocksize));
1176    while(blocksize--){
1177      buffer[blocksize] = _data[target_block+blocksize];
1178    }
1179    _volume->put_block(_location+target_block, buffer);
1180        }
1181
1182    };
1183
1184 public:
1185     RamFS(Volume * volume);
1186     ~RamFS();
1187     static void format(Volume * volume, unsigned int block_size, unsigned int
blocks_per_file);
1188
1189     unsigned int fdesc_table();
1190
1191     //Data access methods.
1192     Volume * volume();
1193
1194     //Allocation methods.
1195     int alloc_fdesc();
1196     void free_fdesc(unsigned int fdesc);
1197     int alloc_block();
1198     int alloc_block(unsigned int size);
1199     void free_block(unsigned int block);
1200     void free_block(unsigned int block, unsigned int size);
1201
1202     //FileDescriptor methods.
1203     FileDescriptor * open_filedescriptor(unsigned int index);
1204     void close_filedescriptor(FileDescriptor * fdesc);
1205     void delete_filedescriptor(unsigned int index);
1206
1207     //Directory Methods.
1208
1209     static int init(System_Info *si);
1210
1211 private:
1212     // RamFS implementation methods
1213
1214 private:
1215     // RamFS attributes
1216     bool _lock;
1217     ramfs_sb _physicaldata;
1218     Volume * _volume;
1219     DummyAllocator<AllocableData> * blocks_alloc;
```

```
1220     DummyAllocator<AllocableData> * fdesc_alloc;
1221
1222
1223 };
1224
1225 __END_IMP
1226 __END_SYS
1227
1228 #endif
1229
1230
1231 // EPOS FileSystem Common Package Implementation
1232 //
1233 // Author: Hugo Marcondes
1234 // Documentation: $EPOS/doc/filesystem      Date: 25 Aug 2004
1235
1236 #include <filesystem.h>
1237 #include <abstraction/filesystem/common.h>
1238
1239 __BEGIN_SYS
1240 __BEGIN_IMP
1241
1242 // Class attribute
1243 // type FileSystem_Common::attribute;
1244
1245 // Methods
1246
1247 // Class methods
1248
1249 __END_IMP
1250 __END_SYS
1251
1252
1253 // EPOS RamFS Implementation
1254 //
1255 // Author: Hugo Marcondes
1256 // Documentation: $EPOS/doc/filesystem      Date: 25 Aug 2004
1257
1258 #include <abstraction/filesystem/ramfs.h>
1259 #include <utility/dummy_allocator.h>
1260 #include <utility/bitmap.h>
1261
1262 __BEGIN_SYS
1263 __BEGIN_IMP
1264
1265 // Class attributes
1266 // type RamFS::attribute;
1267
1268 // Constructors
1269 RamFS::RamFS(Volume * volume)
1270 {
1271     db<RamFS>(TRC) << "RamFS(volume= " << volume << ")\n";
1272     _volume = volume;
1273     //set volume_block_size == SUPERBLOCK to read SB and discover TRUE blocksize.
1274     //Initial Block Size
1275     int ibs = sizeof(ramfs_sb) + _volume->blocksize();
1276     _volume->blocksize(ibs);
1277     unsigned char * data = reinterpret_cast<unsigned char *>(kmalloc(ibs));
1278     _volume->get_block(0, data);
1279
1280     ramfs_sb * sb = reinterpret_cast<ramfs_sb *>(data);
1281
1282     //Make endian conversion ... TODO
1283     _physicaldata.block_size = sb->block_size;
1284     _physicaldata.block_count = sb->block_count;
1285     _physicaldata.block_free = sb->block_free;
1286     _physicaldata.block_bitmap = sb->block_bitmap;
```

```
1287     _physicaldata.first_data = sb->first_data;
1288     _physicaldata.inode_count = sb->inode_count;
1289     _physicaldata.inode_free = sb->inode_free;
1290     _physicaldata.inode_bitmap = sb->inode_bitmap;
1291     _physicaldata.inode_table = sb->inode_table;
1292
1293     db<RamFS>(INF) << "SUPERBLOCK MOUNTED(" <<
1294       _physicaldata.block_size << "," <<
1295       _physicaldata.block_count << "," <<
1296       _physicaldata.block_free << "," <<
1297       _physicaldata.block_bitmap << "," <<
1298       _physicaldata.first_data << "," <<
1299       _physicaldata.inode_count << "," <<
1300       _physicaldata.inode_free << "," <<
1301       _physicaldata.inode_bitmap << "," <<
1302       _physicaldata.inode_table << ")\n";
1303
1304     //Setting TRUE filesystem block size.
1305     _volume->blocksize(_physicaldata.block_size);
1306
1307     //Instantiating the block allocator.
1308     AllocableData * bmap = new(kmalloc(sizeof(AllocableData)))
AllocableData(_volume, _physicaldata.block_bitmap, _physicaldata.block_count);
1309     blocks_alloc = new(kmalloc(sizeof(DummyAllocator<AllocableData>)))
DummyAllocator<AllocableData>(bmap);
1310
1311     //Instantiating the filedescriptor allocator.
1312     AllocableData * imap = new(kmalloc(sizeof(AllocableData)))
AllocableData(_volume, _physicaldata.inode_bitmap, _physicaldata.inode_count);
1313     fdesc_alloc = new(kmalloc(sizeof(DummyAllocator<AllocableData>)))
DummyAllocator<AllocableData>(imap);
1314
1315     //TODO -> Mount Root Directory.
1316 }
1317
1318 RamFS::~RamFS()
1319 {
1320   db<RamFS>(TRC) << "~RamFS()\n";
1321   blocks_alloc->~DummyAllocator<AllocableData>();
1322   kfree(blocks_alloc);
1323   fdesc_alloc->~DummyAllocator<AllocableData>();
1324   kfree(fdesc_alloc);
1325
1326 }
1327
1328 // Methods
1329 unsigned int RamFS::fdesc_table(){
1330   return _physicaldata.inode_table;
1331 }
1332
1333 Volume * RamFS::volume(){ return _volume; }
1334
1335 int RamFS::alloc_block(){
1336   unsigned int value = reinterpret_cast<unsigned int>(blocks_alloc->alloc());
1337   if(value == 0) {
1338     db<RamFS>(ERR) << "RamFS::alloc_block() -> Couldn't allocate blocks.\n";
1339   }
1340   value--;
1341   return value;
1342 }
1343
1344 int RamFS::alloc_fdesc(){
1345   unsigned int value = reinterpret_cast<unsigned int>(fdesc_alloc->alloc());
1346   if(value == 0) {
1347     db<RamFS>(ERR) << "RamFS::alloc_fdesc() -> Couldn't allocate fdesc.\n";
1348   }
1349   value--;
```

```
1350   return value;
1351 }
1352
1353 int RamFS::alloc_block(unsigned int size){
1354   unsigned int value = reinterpret_cast<unsigned int>(blocks_alloc->alloc(size));
1355   if(value == 0) {
1356     db<RamFS>(ERR) << "RamFS::alloc_fdesc(size = " << size << ") -> Couldn't
allocate blocks.\n";
1357   }
1358   value--;
1359   return value;
1360 }
1361
1362 void RamFS::free_block(unsigned int block){
1363   block++;
1364   blocks_alloc->free((void *)block);
1365 }
1366
1367 void RamFS::free_fdesc(unsigned int fdesc){
1368   fdesc++;
1369   fdesc_alloc->free((void *)fdesc);
1370 }
1371
1372 void RamFS::free_block(unsigned int block, unsigned int size){
1373   block++;
1374   blocks_alloc->free((void *)block, size);
1375 }
1376
1377 RamFS::FileDescriptor * RamFS::open_filedescriptor(unsigned int index){
1378   RamFS::FileDescriptor * tmp = new(kmalloc(sizeof(RamFS::FileDescriptor)))
RamFS::FileDescriptor(this, index);
1379   if(tmp->type() == RamFS::FileDescriptor::EMPTY) {
1380     db<RamFS>(ERR) << "Opening a EMPTY descriptor(" << index << ")\n";
1381     return 0;
1382   }
1383   return tmp;
1384 }
1385
1386 void RamFS::close_filedescriptor(FileDescriptor * fdesc){
1387   fdesc->~FileDescriptor();
1388   kfree(fdesc);
1389 }
1390
1391 void RamFS::delete_filedescriptor(unsigned int index) {
1392   RamFS::FileDescriptor::remove(this, index);
1393 }
1394
1395 // Class methods
1396 // TODO -> Need a return status !
1397 void RamFS::format(Volume * volume, unsigned int block_size, unsigned int
blocks_per_file)
1398 {
1399
1400     db<RamFS>(TRC) << "RamFS::format(volume= " << volume << ", block_size=" <<
block_size << ", blocks_per_file=" << blocks_per_file << ")\n";
1401
1402     if(block_size < 17){ db<RamFS>(ERR) << "Unable to format. TMP Error!\n"; return;
}//Garantir que rootDir file sera de apenas 1 bloco.
1403
1404     volume->blocksize(block_size);
1405     unsigned char * buffer = reinterpret_cast<unsigned char *>(kmalloc(block_size));
1406     ramfs_sb * superblock = reinterpret_cast<ramfs_sb *>(buffer);
1407
1408     BitmapAllocator<0> bb_map(volume->size());
1409
1410     db<RamFS>(INF) << "Creating superblock data\n";
1411
```

```
1412     unsigned int block_bitmap_size = (((volume->size() + 7)/8) + block_size - 1)/
block_size;
1413     unsigned int numbers_of_inode = (volume->size() - 5) / blocks_per_file;
1414     unsigned int inode_bitmap_size = (((numbers_of_inode + 7)/8) + block_size -
1)/block_size;
1415     unsigned int ilist_size = ((numbers_of_inode *
sizeof(RamFS::FileDescriptor::FileDescriptorData)) + block_size -1)/block_size;
1416
1417     int tmp, tmp2, block_bitmap, inode_bitmap;
1418
1419     superblock->first_data = reinterpret_cast<unsigned int>(bb_map.alloc(1));
1420     superblock->block_size = block_size;
1421
1422     block_bitmap = reinterpret_cast<int>(bb_map.alloc(block_bitmap_size));
1423     if(block_bitmap > 0){
1424       superblock->block_bitmap = block_bitmap;
1425     } else {
1426       db<RamFS>(ERR) << "Unable to format. Couldn't alloc block_bitmap.\n";
1427       return;
1428     }
1429
1430     inode_bitmap = reinterpret_cast<int>(bb_map.alloc(inode_bitmap_size));
1431     if(inode_bitmap > 0){
1432       superblock->inode_bitmap = inode_bitmap;
1433     } else {
1434       db<RamFS>(ERR) << "Unable to format. Couldn't alloc inode_bitmap.\n";
1435       return;
1436     }
1437
1438     tmp = reinterpret_cast<int>(bb_map.alloc(ilist_size));
1439     if(tmp > 0){
1440       superblock->inode_table = tmp;
1441       tmp2 = tmp;
1442     } else {
1443       db<RamFS>(ERR) << "Unable to format. Couldn't alloc inode_list.\n";
1444       return;
1445     }
1446
1447     superblock->block_count = volume->size();
1448     superblock->block_free = volume->size() -
(1+block_bitmap_size+inode_bitmap_size+ilist_size-1);
1449     superblock->inode_count = numbers_of_inode;
1450     superblock->inode_free = numbers_of_inode - 1;
1451
1452     volume->put_block(0, buffer);
1453
1454     db<RamFS>(INF) << "SUPERBLOCK FORMATED(" <<
1455       superblock->block_size << "," <<
1456       superblock->block_count << "," <<
1457       superblock->block_free << "," <<
1458       superblock->block_bitmap << "," <<
1459       superblock->first_data << "," <<
1460       superblock->inode_count << "," <<
1461       superblock->inode_free << "," <<
1462       superblock->inode_bitmap << "," <<
1463       superblock->inode_table << ")\n";
1464
1465     //Create RootDir File;
1466     struct direntry {
1467      unsigned int inode;
1468      unsigned short lenght;
1469      unsigned char name_lenght;
1470      char name[257];
1471     };
1472     struct direntry entry;
1473     int buffer_size = 0;
1474     //Definition of "." entry
```

```
1475     entry.inode = 1;
1476     entry.lenght = 8;
1477     entry.name_lenght = 1;
1478     entry.name[0] = '.';
1479     int i = 0;
1480     while(i++ < entry.lenght){
1481       buffer[buffer_size] = ((unsigned char *)&entry)[i];
1482       buffer_size++;
1483     }
1484     //Definition of ".." entry
1485     entry.inode = 1;
1486     entry.lenght = 9;
1487     entry.name_lenght = 2;
1488     entry.name[0] = '.';
1489     entry.name[1] = '.';
1490     i = 0;
1491     while(i++ < entry.lenght){
1492       buffer[buffer_size] = ((unsigned char *)&entry)[i];
1493       buffer_size++;
1494     }
1495
1496     tmp = reinterpret_cast<unsigned int>(bb_map.alloc(1));
1497     if(tmp < 0){
1498       db<RamFS>(ERR) << "Unable to format. Couldn't alloc data for root dir.\n";
1499       return;
1500     }
1501
1502     //Definition of rootDir inode.
1503     FileDescriptor::FileDescriptorData rootInode;
1504     rootInode.type = RamFS::FileDescriptor::DIRECTORY;
1505     rootInode.links = 2;
1506     rootInode.size = buffer_size;
1507     rootInode.zones[0] = tmp;
1508
1509     //Save DirFile
1510     volume->put_block(rootInode.zones[0], buffer);
1511
1512     //Save InodeFile
1513     BitmapAllocator<0> inode_map(numbers_of_inode);
1514     tmp = reinterpret_cast<int>(inode_map.alloc(1));
1515     int inodeblock = tmp2 + ((sizeof(RamFS::FileDescriptor::FileDescriptorData) *
tmp ) / block_size);
1516     int inodeoffset = (sizeof(RamFS::FileDescriptor::FileDescriptorData) * tmp) %
block_size;
1517     volume->get_block(inodeblock, buffer);
1518     tmp = sizeof(RamFS::FileDescriptor::FileDescriptorData);
1519     while(tmp--){
1520   buffer[inodeoffset+tmp] = ((unsigned char *)&rootInode)[tmp];
1521     }
1522     volume->put_block(inodeblock, buffer);
1523
1524     //Save BlockBitmap
1525     buffer = reinterpret_cast<unsigned char *>(bb_map.get_map());
1526     while(block_bitmap_size--){
1527       int block = block_bitmap+block_bitmap_size;
1528       volume->put_block(block, (buffer+(block_bitmap_size * block_size)));
1529     }
1530
1531     //Save InodeBitmap
1532     buffer = reinterpret_cast<unsigned char *>(inode_map.get_map());
1533     while(inode_bitmap_size--){
1534       int block = inode_bitmap+inode_bitmap_size;
1535       volume->put_block(block, (buffer+(inode_bitmap_size * block_size)));
1536     }
1537
1538 }
1539 __END_IMP
```

```
1540 __END_SYS
1541
1542
1543 // EPOS RamFS Initialization
1544 //
1545 // Author: Hugo Marcondes
1546 // Documentation: $EPOS/doc/filesystem      Date: 25 Aug 2004
1547
1548 #include <abstraction/filesystem/ramfs.h>
1549
1550 __BEGIN_SYS
1551 __BEGIN_IMP
1552
1553 // Class initialization
1554 int RamFS::init(System_Info * si)
1555 {
1556     db<RamFS>(TRC) << "RamFS::init()\n";
1557
1558     return 0;
1559 }
1560
1561 __END_IMP
1562 __END_SYS
1563
1564
1565 // EPOS RamFS Test Program
1566 //
1567 // Author: Hugo Marcondes
1568 // Documentation: $EPOS/doc/filesystem      Date: 25 Aug 2004
1569
1570 #include <utility/ostream.h>
1571 #include <filesystem.h>
1572 #include <framework.h>
1573 #include <storage.h>
1574 #include <volume.h>
1575
1576 __USING_SYS
1577
1578 int main()
1579 {
1580     OStream cout;
1581     Storage myDisk(1);
1582     Imp::Volume myVolume(&myDisk);
1583     //Volume myVolume(&myDisk);
1584
1585     cout << "RamFS test\n";
1586
1587     FileSystem::format(&myVolume, 64, 4);
1588
1589     RamFS myFileSystem(&myVolume);
1590
1591     return 0;
1592 }
1593
1594
1595 // EPOS File Common Package
1596 //
1597 // Author: Hugo Marcondes
1598 // Documentation: $EPOS/doc/file      Date: Mon Oct 25 17:13:11 BRST 2004
1599
1600 #ifndef __file_common_h
1601 #define __file_common_h
1602
1603 #include <system/config_defs.h>
1604
1605 __BEGIN_SYS
1606 __BEGIN_IMP
```

```
1607
1608 class File_Common: public __INT(File_Common)
1609 {
1610 protected:
1611     File_Common() {}
1612
1613 public:
1614     // File common methods
1615
1616 private:
1617     // File common attributes
1618 };
1619
1620 __END_IMP
1621 __END_SYS
1622
1623 #endif
1624
1625
1626 // EPOS ContinuousFile Declarations
1627 //
1628 // Author: Hugo Marcondes
1629 // Documentation: $EPOS/doc/file      Date: Mon Oct 25 17:13:11 BRST 2004
1630
1631 #ifndef __continuousfile_h
1632 #define __continuousfile_h
1633
1634 #include <file.h>
1635 #include <filesystem.h>
1636 #include "common.h"
1637
1638 __BEGIN_SYS
1639 __BEGIN_IMP
1640
1641 class ContinuousFile: public __INT(ContinuousFile), protected File_Common
1642 {
1643 private:
1644     typedef Traits<ContinuousFile> Traits;
1645     static const Type_Id TYPE = Type<ContinuousFile>::TYPE;
1646
1647     // ContinuousFile private imports, types and constants
1648
1649 public:
1650     ContinuousFile(FileSystem * fs, unsigned int index, unsigned int size);
1651     ContinuousFile(FileSystem * fs, char * name, unsigned int size);
1652     ContinuousFile(FileSystem * fs, unsigned int index);
1653     ContinuousFile(FileSystem * fs, char * name);
1654     ~ContinuousFile();
1655     int read(unsigned char * buffer, unsigned int size);
1656     int write(unsigned char * buffer, unsigned int size);
1657     unsigned int size();
1658     void seek(unsigned int pos);
1659     void resize(unsigned int size);
1660     bool eof();
1661
1662     static int init(System_Info *si);
1663
1664 private:
1665     // ContinuousFile implementation methods
1666
1667 private:
1668     // ContinuousFile attributes
1669     FileSystem * _fs;
1670     FileSystem::FileDescriptor * _descriptor;
1671     unsigned int _pos;
1672     unsigned int _fs_blocksize;
1673     unsigned char * _buffer;
```

```
1674 };
1675
1676 __END_IMP
1677 __END_SYS
1678
1679 #endif
1680
1681
1682 // EPOS ContinuousFile Implementation
1683 //
1684 // Author: Hugo Marcondes
1685 // Documentation: $EPOS/doc/file      Date: Mon Oct 25 17:13:11 BRST 2004
1686
1687 #include <abstraction/file/continuousfile.h>
1688
1689 __BEGIN_SYS
1690 __BEGIN_IMP
1691
1692 // Class attributes
1693 // type ContinuousFile::attribute;
1694
1695 // Constructors
1696 ContinuousFile::ContinuousFile(FileSystem * fs, unsigned int index, unsigned int
size)
1697 {
1698     db<ContinuousFile>(TRC) << "ContinuousFile(fs= " << fs << ", index= " << index
<< ", size= " << size << ")\n";
1699     _fs = fs;
1700     _fs_blocksize = _fs->volume()->blocksize();
1701     _pos = 0;
1702     _descriptor = _fs->open_filedescriptor(index);
1703     if(_descriptor == 0){
1704       //Must create a new file;
1705       _descriptor = FileSystem::FileDescriptor::create(_fs,
FileSystem::FileDescriptor::FILE, size, index, true);
1706     } else {
1707       //Truncate File to size.//Must be call resize maybe ?.
1708       _descriptor->size(size);
1709     }
1710     _buffer = reinterpret_cast<unsigned char *>(kmalloc(_fs_blocksize));
1711 }
1712
1713 ContinuousFile::ContinuousFile(FileSystem * fs, char * name, unsigned int size)
1714 {
1715     db<ContinuousFile>(TRC) << "ContinuousFile(fs= " << fs << ", name= " << name <<
", size= " << size << ")\n";
1716     //Future Work ?
1717 }
1718
1719 ContinuousFile::ContinuousFile(FileSystem * fs, unsigned int index)
1720 {
1721     db<ContinuousFile>(TRC) << "ContinuousFile(fs= " << fs << ", index= " << index
<< ")\n";
1722     _fs = fs;
1723     _fs_blocksize = _fs->volume()->blocksize();
1724     _pos = 0;
1725     _descriptor = _fs->open_filedescriptor(index);
1726     if(_descriptor == 0){
1727       //Must create a new file;
1728       _descriptor = FileSystem::FileDescriptor::create(_fs,
FileSystem::FileDescriptor::FILE, Traits::DEFAULTSIZE, index, true);
1729     }
1730     _buffer = reinterpret_cast<unsigned char *>(kmalloc(_fs_blocksize));
1731 }
1732
1733 ContinuousFile::ContinuousFile(FileSystem * fs, char * name)
1734 {
```

```
1735     db<ContinuousFile>(TRC) << "ContinuousFile(fs= " << fs << ", name= " << name <<
")\n";
1736     //Future Work?
1737 }
1738
1739 ContinuousFile::~ContinuousFile()
1740 {
1741     db<ContinuousFile>(TRC) << "~ContinuousFile()\n";
1742     _fs->close_filedescriptor(_descriptor);
1743     kfree(_buffer);
1744 }
1745
1746 // Methods
1747 int ContinuousFile::read(unsigned char * buffer, unsigned int size)
1748 {
1749     db<ContinuousFile>(TRC) << "ContinuousFile::read(buffer= " << buffer << ",
size= " << size << ")\n";
1750     int bytes_read = 0;
1751     unsigned int actual_zone = _pos / _fs_blocksize;
1752     unsigned int offset = _pos % _fs_blocksize;
1753     _descriptor->getdata(actual_zone, _buffer);
1754
1755     while(size){
1756       //checks if i'm on zone boundary.
1757       if(offset >= _fs_blocksize){
1758         //kout << "Zone boundary arrived.\n";
1759         offset = 0; //Go to the start of new zone.
1760         _descriptor->getdata(++actual_zone, _buffer);
1761       }
1762       if(_pos >= _descriptor->size()){//Am I in the end of life, oops file ?
1763         //kout << "End of file\n";
1764         return bytes_read;
1765       }
1766       //Okay I'm at the right place!
1767       buffer[bytes_read++] = _buffer[offset++];
1768       size--;
1769       _pos++;
1770     }
1771
1772     db<ContinuousFile>(INF) << "New position = " << _pos << "\n";
1773
1774     return bytes_read;
1775 }
1776
1777 int ContinuousFile::write(unsigned char * buffer, unsigned int size)
1778 {
1779     db<ContinuousFile>(TRC) << "ContinuousFile::write(buffer= " << buffer << ",
size= " << size << ")\n";
1780
1781     int bytes_wrote = 0;
1782     unsigned int actual_zone = _pos / _fs_blocksize;
1783     unsigned int offset = _pos % _fs_blocksize;
1784
1785     _descriptor->getdata(actual_zone, _buffer); //Recover actual zone.
1786
1787     while(size != 0){
1788       //checks if i'm on zone boundary.
1789       if(offset >= _fs_blocksize){
1790         _descriptor->putdata(actual_zone, _buffer);//Save actual buffer.
1791         _descriptor->getdata(++actual_zone, _buffer);//Restore next zone.
1792         offset = 0;
1793       }
1794       if(_pos >= _descriptor->size()){//Am I in the end of life, oops file ?
1795         _descriptor->putdata(actual_zone, _buffer);
1796         return bytes_wrote;
1797       }
1798       _buffer[offset++] = buffer[bytes_wrote++];
```

```
1799       size--;
1800       _pos++;
1801     }
1802
1803     db<ContinuousFile>(INF) << "New position = " << _pos << "\n";
1804
1805     _descriptor->putdata(actual_zone, _buffer);
1806
1807     return bytes_wrote;
1808 }
1809
1810 unsigned int ContinuousFile::size()
1811 {
1812     db<ContinuousFile>(TRC) << "ContinuousFile::size()\n";
1813     return _descriptor->size();
1814 }
1815
1816 void ContinuousFile::seek(unsigned int pos)
1817 {
1818     db<ContinuousFile>(TRC) << "ContinuousFile::seek(pos= " << pos << ")\n";
1819     if(pos > _descriptor->size()) {
1820       _pos = _descriptor->size();
1821     } else {
1822       _pos = pos;
1823     }
1824 }
1825
1826 void ContinuousFile::resize(unsigned int size)
1827 {
1828     db<ContinuousFile>(TRC) << "ContinuousFile::resize(size= " << size << ")\n";
1829     _descriptor->size(size, true);
1830 }
1831
1832 bool ContinuousFile::eof()
1833 {
1834     db<ContinuousFile>(TRC) << "ContinuousFile::eof()\n";
1835     if(_pos >= _descriptor->size()){
1836       return true;
1837     }else{
1838       return false;
1839     }
1840 }
1841
1842 // Class methods
1843 __END_IMP
1844 __END_SYS
1845
1846
1847 // EPOS ContinuousFile Initialization
1848 //
1849 // Author: Hugo Marcondes
1850 // Documentation: $EPOS/doc/file      Date: Mon Oct 25 17:13:11 BRST 2004
1851
1852 #include <abstraction/file/continuousfile.h>
1853
1854 __BEGIN_SYS
1855 __BEGIN_IMP
1856
1857 // Class initialization
1858 int ContinuousFile::init(System_Info * si)
1859 {
1860     db<ContinuousFile>(TRC) << "ContinuousFile::init()\n";
1861
1862     return 0;
1863 }
1864
1865 __END_IMP
```

```
1866 __END_SYS
1867
1868
1869 // EPOS ContinuousFile Test Program
1870 //
1871 // Author: Hugo Marcondes
1872 // Documentation: $EPOS/doc/file      Date: Mon Oct 25 17:13:11 BRST 2004
1873
1874 #include <utility/ostream.h>
1875 #include <file.h>
1876 #include <storage.h>
1877 #include <volume.h>
1878 #include <filesystem.h>
1879 #include <framework.h>
1880
1881 __USING_SYS
1882
1883 int main()
1884 {
1885     OStream cout;
1886
1887     cout << "ContinuousFile test\n";
1888
1889     Storage myDisk(1);
1890     Imp::Volume myVolume(&myDisk);
1891     //Volume myVolume(&myDisk);
1892
1893     FileSystem::format(&myVolume, 512, 4);
1894
1895     Imp::FileSystem myFileSystem(&myVolume);
1896
1897     ContinuousFile * file = new ContinuousFile(&myFileSystem, (unsigned int)1,
(unsigned int)2048);
1898     cout << "The size of opened file is " << file->size() << "\n";
1899
1900     unsigned int tsize = 1024 ;
1901     char * text = new char[tsize];
1902     for(int i = 0; i < tsize; i++){
1903       text[i] = '1';
1904     }
1905     cout << file->write((unsigned char*)text, tsize) << " bytes wroted on file.\n";
1906     delete file;
1907
1908     file = new ContinuousFile(&myFileSystem, (unsigned int)2, (unsigned int)2048);
1909     cout << "The size of opened file is " << file->size() << "\n";
1910     for(int i = 0; i < tsize; i++){
1911       text[i] = '2';
1912     }
1913     cout << file->write((unsigned char*)text, tsize) << " bytes wroted on file.\n";
1914     delete file;
1915
1916     file = new ContinuousFile(&myFileSystem, (unsigned int)3);
1917     cout << "The size of opened file is " << file->size() << "\n";
1918     for(int i = 0; i < tsize; i++){
1919       text[i] = '3';
1920     }
1921     cout << file->write((unsigned char*)text, tsize) << " bytes wroted on file.\n";
1922     delete file;
1923
1924     char * check_buffer = new char[tsize];
1925     int bytesread, errors;
1926     file = new ContinuousFile(&myFileSystem, (unsigned)1);
1927     cout << "Opening file 1 which have size " << file->size() << " bytes.\n";
1928     bytesread = file->read((unsigned char *)check_buffer, tsize);
1929     cout << "Readed " << bytesread << " bytes from file.\n";
1930     errors = 0;
1931     for(int i = 0; i < bytesread; i++){
```

```
1932   if(check_buffer[i] != '1'){
1933      errors++;
1934   }
1935     }
1936     cout << "Found " << errors << " on file 1.\n";
1937     delete file;
1938
1939     file = new ContinuousFile(&myFileSystem, (unsigned)2);
1940     cout << "Opening file 2 which have size " << file->size() << " bytes.\n";
1941     bytesread = file->read((unsigned char *)check_buffer, tsize);
1942     cout << "Readed " << bytesread << " bytes from file.\n";
1943     errors = 0;
1944     for(int i = 0; i < bytesread; i++){
1945         if(check_buffer[i] != '2'){
1946            errors++;
1947         }
1948     }
1949     cout << "Found " << errors << " on file 2.\n";
1950     delete file;
1951
1952     file = new ContinuousFile(&myFileSystem, (unsigned)3);
1953     cout << "Opening file 3 which have size " << file->size() << " bytes.\n";
1954     bytesread = file->read((unsigned char *)check_buffer, tsize);
1955     cout << "Readed " << bytesread << " bytes from file.\n";
1956     errors = 0;
1957     for(int i = 0; i < bytesread; i++){
1958         if(check_buffer[i] != '3'){
1959            errors++;
1960         }
1961     }
1962     cout << "Found " << errors << " on file 3.\n";
1963
1964     file->resize((unsigned) 2048);
1965     cout << file->size() << "\n";
1966     cout << file->write((unsigned char*)text, tsize) << " bytes wroted on file.\n";
1967     delete file;
1968
1969     file = new ContinuousFile(&myFileSystem, (unsigned)3);
1970     cout << "Opening file 3 which have size " << file->size() << " bytes.\n";
1971     bytesread = file->read((unsigned char *)check_buffer, tsize);
1972     cout << "Readed " << bytesread << " bytes from file.\n";
1973     errors = 0;
1974     for(int i = 0; i < bytesread; i++){
1975         if(check_buffer[i] != '3'){
1976            errors++;
1977         }
1978     }
1979     bytesread = file->read((unsigned char *)check_buffer, tsize);
1980     cout << "Readed " << bytesread << " bytes from file.\n";
1981     for(int i = 0; i < bytesread; i++){
1982         if(check_buffer[i] != '3'){
1983            errors++;
1984         }
1985     }
1986     cout << "Found " << errors << " on file 3.\n";
1987
1988
1989     return 0;
1990 }
1991
1992
1993 // EPOS File Common Package Implementation
1994 //
1995 // Author: Hugo Marcondes
1996 // Documentation: $EPOS/doc/file      Date: Mon Oct 25 17:13:11 BRST 2004
1997
1998 #include <file.h>
```

```
1999 #include <abstraction/file/common.h>
2000
2001 __BEGIN_SYS
2002 __BEGIN_IMP
2003
2004 // Class attribute
2005 // type File_Common::attribute;
2006
2007 // Methods
2008
2009 // Class methods
2010
2011 __END_IMP
2012 __END_SYS
2013
2014
2015

2016
```

Apêndice B

Artigo

Um Sistema de Arquivos para o EPOS

Hugo Marcondes[1] , **Antônio Augusto Medeiros Fröhlich**[1]

[1]Laboratório de Integração Software Hardware – Univerisidade Federal de Santa Catarina
Campus Universitário - LISHA/CTC/UFSC - CP 476 – 88040-900 Florianópolis - SC

hugom@lisha.ufsc.br, guto@lisha.ufsc.br

Abstract. *With the growth of the market of embedded systems the support to a file system is each more present in the requirements of the applications. This work presents a modeling of file systems, following the methodology of Application Oriented System Design, of the necessary abstractions so that operational system* EPOS *offers support to file systems in such environment.*

Keywords: *operating systems, embedded systems, file systems, software components.*

Resumo. *Com o crescimento do mercado de sistemas embutidos é cada vez mais presente nos requisitos das aplicações o suporte a um sistema de arquivos. Este trabalho apresenta um modelagem de sistemas de arquivos, seguindo a metodologia de desenvolvimento de sistemas orientados a aplicação, das abstrações necessárias para que o sistema operacional* EPOS *ofereça suporte a sistemas de arquivos em tal ambiente.*

Palavras-chave: *sistemas operacionais, sistemas embutidos, sistema de arquivos, componentes de software.*

1. Introdução

Embora grande parte das aplicações embarcadas se destinam a efetuar apenas controle de sistemas maiores, armazenando seus dados na memória RAM do sistema, cada vez se torna mais presente em nossas vidas, aplicações embarcadas de maior complexidade e de maior valor agregado, que necessitam armazenar seus dados em um sistema de arquivos. Exemplos de tais aplicações são os tocadores de MP3, câmeras digitais e computadores de bolso[1].

Por outro lado, sistemas de suporte ao tempo de execução de sistemas embarcados devem fornecer a aplicação somente os requisitos estritamente necessários a sua execução, evitando assim *overheads* desnecessários em sua natureza dedicada. Pensando nisso, [Fröhlich, 2001] apresenta uma metodologia para o desenvolvimento de sistemas denominado *Application Oriented System Design*(AOSD). A partir deste trabalho o sistema operacional EPOS (*Embedded Parallel Operanting System*) foi concebido com o intuito de demonstrar as idéias propostas em [Fröhlich, 2001].

Este trabalho se baseia na modelagem do domínio de sistemas de arquivos seguindo os conceitos abordados no trabalho de [Fröhlich, 2001], identificando as famílias de componentes de software que irão integrar o sistema de arquivos do EPOS.

[1]PDA e PALMTOPS

2. Sistemas de Arquivos

”Toda aplicação necessita armazenar e recuperar informações” [Tanenbaum and Woodhull, 1997], para isso o sistema operacional fornece a aplicação um sistema de arquivos, que define uma forma organizacional com que os dados são armazenados em um dispositivo de armazenamento persistente do ponto de vista físico, permitindo assim que diversos arquivos coexistam e se organizem de forma lógica para o ponto de vista do usuário do sistema operacional.

Os sistemas de arquivos visam solucionar três fatores que limitam a execução de aplicações em um sistema computacional. São elas:

- Quantidade limitada para armazenamento de informações no espaço de endereçamento de um processo
- Perda das informações contidas no espaço de endereçamento após a execução de um processo
- Compartilhamento de informações entre processos distintos em um sistema

Segundo [Tanenbaum and Woodhull, 1997], a solução mais comum para esses fatores é o armazenamento de informações em discos ou outras mídias externas em unidades chamadas de arquivos. Assim os processos podem ler e escrever em arquivos as informações que necessitam ser compartilhadas, mantidas após a sua execução e as informações que são grandes o suficiente para não caberem na memória principal do sistema, dentro do espaço de endereçamento do processo.

Assim os arquivos são gerenciados pelo sistema operacional e a maneira como eles são estruturados, nomeados, acessados, utilizados, protegidos e implementados são tópicos importantes no desenvolvimento de um sistema operacional [Tanenbaum and Woodhull, 1997]. Dá-se o nome de sistemas de arquivos a parte do sistema operacional que manipula os arquivos.

Afim de garantir a persistência dos dados, os sistemas de arquivos são armazenados em dispositivos de dados permanentes. Em sua grande maioria esses dispositivos são discos rígidos magnéticos ou ópticos[2], embora atualmente existam tecnologias de armazenamento em memórias não voláteis, como a memória FLASH, que se assemelham as memórias SDRAM e garantem a persistência dos dados mas infelizmente possuem um custo muito elevado, aumentando conseqüentemente o custo de produção em sistemas embutidos que requerem armazenar grandes quantidades de dados.

Os dispositivos de armazenamento de dados é divido em blocos físicos, e devido ao crescente aumento da capacidade de armazenamento do mesmo, tornou-se necessário criar um nível de endereçamento lógico, favorecendo a gestão de blocos por parte do sistema de arquivos.

O tamanho do bloco lógico definido pelo sistema de arquivos é de suma importância para a performance do sistema de arquivos. Blocos muito pequenos implicam em arquivos com muitos blocos, ocasionando em perda de performance devido ao aumento do tempo de busca desses blocos no disco. Blocos muito grandes aumentam a fragmentação interna, ocasionando assim em um desperdício da capacidade de armazenamento. Um estudo de [Mullender and Tannembaum, 1984] mostram que o tamanho médio de um arquivo no sistema UNIX é por volta de 1Kb.

Um dispositivo de armazenamento também pode ser particionado em diversos volumes, de capacidade inferior a capacidade total do mesmo. Os sistema de arquivos estão contidos nos volumes e só conhecem o volume a qual pertencem, sendo assim, cada volume é uma unidade independente para o sistema. Somente uma parte do sistema operacional diferencia um dispositivo físico e um volume, sendo que o resto do sistema conhece apenas os volumes [Fröhlich, 1994].

[2] CD-ROM, CD-RW, DVD

Arquivos são um mecanismo de abstração [Tanenbaum and Woodhull, 1997]. Eles provem uma maneira para o armazenamento de informações (dados) de forma persistente, para um acesso futuro. Embora [Tanenbaum and Woodhull, 1997] cite como característica mais importante a um mecanismo de abstração a forma como estas são nomeadas, um arquivo não precisa necessariamente possuir um nome contextualizado, ou seja, um nome definido pelo seu criador e que geralmente possui alguma relação com seu conteúdo. Pensando em um Sistema de Arquivos como um conjunto de elementos chamados de arquivo, podemos fazer referência a eles, e conseqüentemente acessá-los, através da sua posição dentro deste conjunto. Um exemplo seria acessar o arquivo através do índice dentro da lista de descritores de arquivos. Em um sistema interativo como o Windows, talvez seja inaceitável que o sistema de arquivos não possua um sistema de nomeação, contudo no mundo de sistemas embutidos é bastante razoável os arquivos não possuírem nomes.

Sua estrutura pode ser definida de diversas formais, com um conjunto de registros, cujo tamanho seja pré-determinado, ou simplesmente como um grande *array* linear de bytes. Devido a praticidade, atualmente, grande parte dos sistemas de arquivos implementam um arquivo como um *array* de bytes. Adicionalmente o sistema operacional pode atribuir um tipo para o arquivo em sua implementação, com isso alguns arquivos podem possuir uma semântica distinta e serem tratados de forma diferente. Os dois tipos de arquivos mais comuns são os arquivos própriamente ditos, ou os diretórios, que geralmente são implementados sobre o próprio sistema de arquivos como uma arquivo com uma estrutura interna especial.

Adicionalmente o sistema operacional pode definir e implementar atributos adicionais ao sistema de arquivos, extendendo assim a suas funcionalidades, os mais comuns são os atributos para controle de acesso e temporal aos dados e são implementados como atributos adicionais, contudo uma infinidade de atributos podem ser utilizados na especificação do sistema de arquivos. Alguns exemplos são atributos para o controle de *backup* de arquivos, senha para acesso, atributos para ocultar arquivos e atributos para definir arquivos temporários. O próprio tamanho do arquivo é um atributo do mesmo, sendo este presente em todas as específicações.

Pensando-se um uma classificação dos dados presentes em um sistema de arquivos, os mesmos podem ser divididos em dois grupos, os dados do próprio usuário do sistema de arquivos, que compõe os arquivos em si, e em dados denominados meta-dados, que são utilizados pelo sistema operacional para suportar as funcionalidades entregues ao usuário e descrever os dados dos arquivos. Os meta-dados são divididos basicamente em três grupos. Os meta-dados que descrevem a organização do sistema de arquivos como um todo, os meta-dados responsáveis por descrever a organização dos arquivos em si e os metadados responsáveis por descrever a utilização dos recursos presentes internamente no sistema de arquivos, embora em alguns sistemas de arquivos esses meta-dados podem estar presentes de forma unificados em uma mesma estrutura, como por exemplo na FAT[3] que além de descrever o blocos pertencentes ao mesmo arquivo, descreve os blocos que estão livres para alocação.

Um sistema de diretórios também pode ser implementado por um sistema de arquivos, possibilitando que um determinado arquivo possua um nome, que pode ser mapeado para o seu respectivo descritor de arquivos. Adicionalmente o sistema de diretórios permite que os arquivos sejam organizados em diretórios (também conhecido como pastas), facilitando o acesso aos arquivos e permitindo que arquivos distintos possuam o mesmo nome, desde que estejam em um diretório distinto. O sistema de diretórios também pode oferecer funcionalidades adicionais, como a possibilidade de um mesmo arquivos possuir mais de um nome no sistema.

[3]File Allocation Table

3. Decomposição do domínio

Segundo a metodologia apresentada por [Fröhlich, 2001], o trabalho de modelagem de um sistema orientado a aplicação (AOSD) se inicia na decomposição do domínio em questão em famílias de abstrações independentes de cenário que, através de re-usabilidade, podem ser instâncias do mesmo sistema, conforme mostra a figura 1. Através de *configurable features* e aspectos de cenário, a configurabilidade do sistema pode ser alcançada. Tais famílias de abstrações constituem componentes de software que unidos através de um *framework* formam o sistema adaptado a aplicação. Os componentes de tal *framework* são acessados através de Interfaces Infladas que garantem a portabilidade do sistema e são o principal objeto de análise dos requisitos da aplicação.

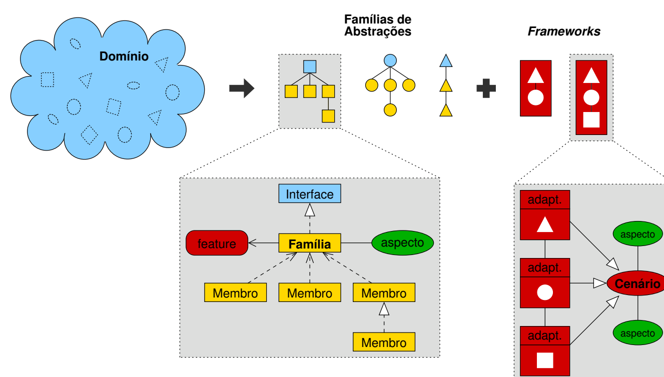

Figure 1: Decomposição de um domínio através da AOSD

O sistema de arquivos foi divido inicialmente em duas visões distintas, a visão do sistema operacional, que enxerga um sistema de arquivos como uma coleção de estruturas que contém meta-dados armazenados em um dispositivo e a visão do usuário que prove a realização das abstrações de arquivos e diretórios, fornecendo acesso ao usuário do sistema operacional ao seus dados. A figura 2 mostra as famílias identificadas no domínio, sendo as mesmas descritas abaixo:

- **Storage :** Responsável por implementar as funcionalidades de determinado hardware de armazenamento de dados. Esta é uma família de mediadores
- **Volume :** Responsável pela interface entre o sistema de arquivos e o hardware de armazenamento de dados, implementando o conceito de volumes através de uma tabela de volumes.
- **Allocator :** Responsável pelo gerenciamento dos blocos livres no sistema e utilização da tabela de descritores de arquivos, implementando algoritmos de alocamento de espaço sobre as estruturas que fazem esse controle.
- **FileSystem :** Responsável por gerenciar os meta-dados específicos de um sistema de arquivos e algumas das operações que são realizadas no sistema de forma global, como por exemplo a exclusão de arquivos.
- **Shared :** Esta é uma família de aspectos e sua função é implementar os algoritmos necessários para que o sistema de arquivos suporte o compartilhamento de arquivos.
- **Protection :** Esta é uma família de aspectos e sua função é implementar os algoritmos necessários para que o sistema de arquivos suporte mecanismos de proteção de acesso aos arquivos.
- **Logging :** Esta é uma família de aspectos e sua função é implementar os algoritmos necessários para que o sistema de arquivos suporte a utilização de arquivos baseados em Log[Rosenblum and Ousterhout, 1992].
- **File :** Implementa as chamadas de sistemas necessários para criar, acessar e modificar arquivos do sistema.

- **Directory :** Implementa as chamadas de sistema responsáveis por acessar diretórios no sistema de arquivo além de incluir, modificar e excluir entradas no diretório.

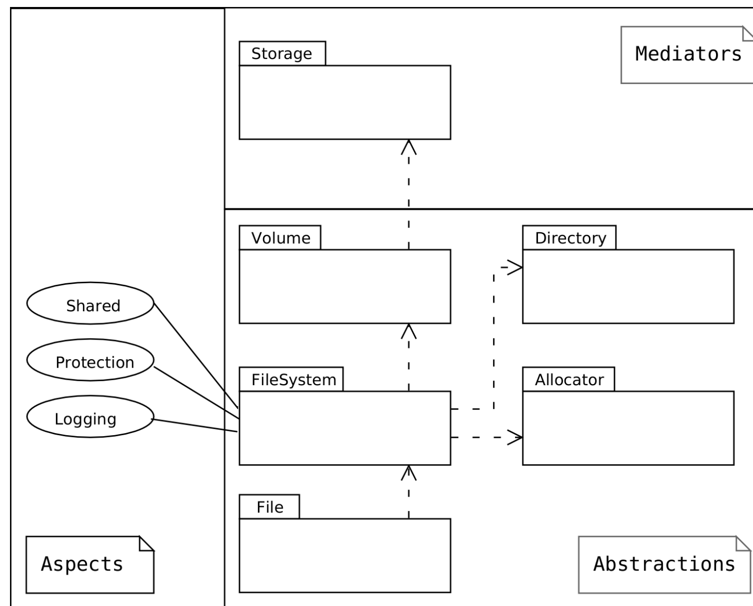

Figure 2: Famílias do Sistemas de Arquivos

3.1. A Família Storage

A família storage visa implementar os mediadores para acesso a um dispositivo de armazenamento de dados persistentes, entre eles, dispositivos ATA, SCSI, memórias Flash, e todos os dispositivos baseados em blocos. A figura 3 mostra alguns membros desta família.

Neste trabalho, o storage foi implementado "emulando" um dispositivo orientado a blocos, sobre a memória RAM do sistema.

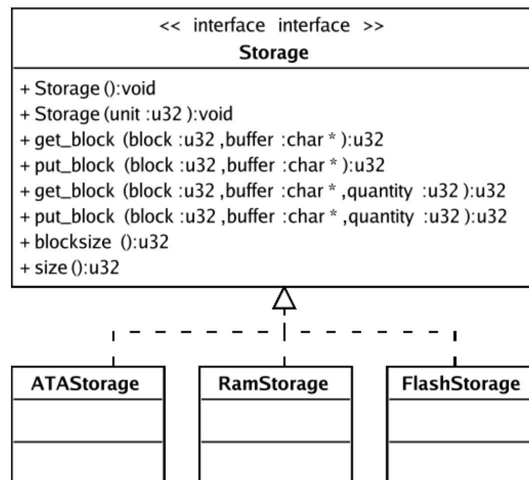

Figure 3: Interface Inflada e membros da família Storage

Esta família possui um interface inflada dissociativa segundo o trabalho de Fröhlich [Fröhlich, 2001], ou seja, seus membros possuem tanto métodos em comum, como métodos exclusivos aos mesmos. É o caso de membro FlashStorage, que necessita implementar métodos para apagar setores de uma memória flash, e que não fazem sentido em dispositivos ATA.

3.2. A Família Volume

Esta família implementa uma unidade lógica denominada VOLUME, também conhecida como partição em alguns sistemas. Cada membro desta família encapsula o conhecimento de uma determinada especificação de tabela de volumes. Esta família possui as seguintes funções no sistema como um todo:

- Garantir, caso necessário, a tradução de endereços de blocos lógicos para blocos físicos.
- Garantir o acesso de um determinado sistema de arquivos exclusivamente a região que lhe foi destinada.
- Encapsular o conceito de tabelas de volumes.

Na figura 4, podemos verificar os membros, inicialmente atribuídos a essa família. O membro FlatVolume implementa um volume que ocupa todo o espaço de armazenamento do componente Storage, ficando a ele atribuída apenas as funções de tradução de endereços lógicos para físicos. O membro EposVolume implementa uma especificação de tabela de volumes própria do EPOS, visando suprir algumas necessidades de se conhecer o tamanho do bloco lógico. Em grande parte dos sistemas, o tamanho do bloco lógico esta armazenado dentro do próprio sistema de arquivos, com isso é necessário inicialmente, efetuar uma busca no disco para localizar o descritor do sistema, geralmente chamado de superbloco, para se conhecer o real tamanho do bloco lógico, enquanto essa informação já poderia estar armazenada na própria tabela de volumes, simplificando assim o algoritmo de inicialização do sistema de arquivos. O membro DosVolume implementa a especificação de uma tabela de volumes do Sistema Operacional DOS.

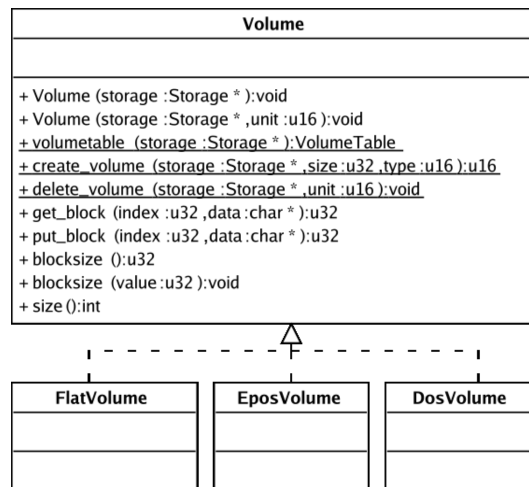

Figure 4: Interface Inflada e membros da família Volume

3.3. A Família Allocator

Esta família é implementada como uma classe utilitária do EPOS, ou seja, seus membros são genéricos para que outras partes do sistema operacional, assim como a própria aplicação, possam utiliza-lo para efetuarem alocação de recursos.

Nesta modelagem, buscou-se encontrar a separação entre o algoritmo de alocação e a estrutura de dados utilizada pelo mesmo para efetuar o gerenciamento dos blocos que estão livre. Através do paradigma de programação genérica, este resultado foi alcançado, utilizando-se o recurso de templates da linguagem C++.

Através da definição de interface comum, conforme a figura 5, foi possível modelar um conjunto de classes que são passadas a classe que implementa o algoritmo de alocação através de

um template, gerando assim um alocador específico em tempo de compilação, conforme pode ser visualizado na figura 6.

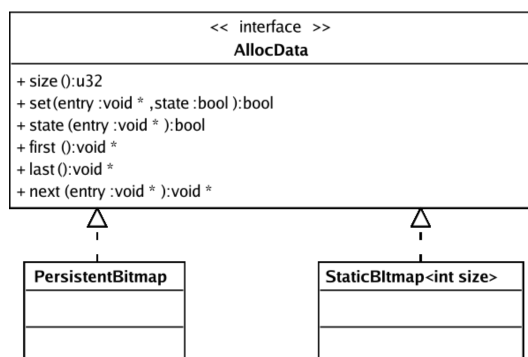

Figure 5: Interface dos dados de alocação.

Para a implementação do protótipo deste trabalho, foi criada uma classe de dados, chamada PersistentBitmap que é responsável por efetuar a leitura do Bitmap[4] do sistema de arquivos no disco e disponibilizá-lo ao algoritmo através de sua interface, assim como manter os dados atualizados no disco, conforme ocorrem modificações neste Bitmap.

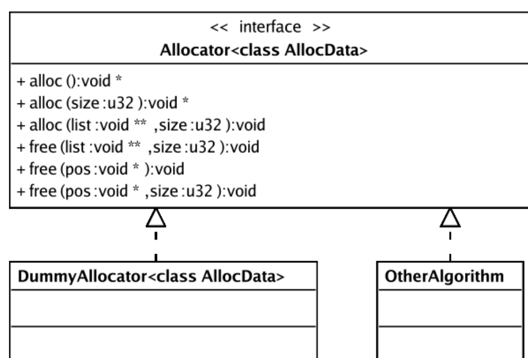

Figure 6: Interface dos alocadores do sistema.

3.4. A Família FileSystem

Sendo esta a família central de um de sistemas de arquivos, cabe a esta efetuar grande parte da implementação de uma determinada especificação de um sistema de arquivos.

Todos os algoritmos necessários para acessar os meta-dados que compõem um sistema de arquivos são realizados nos membros desta. Dessa maneira a implementação dos membros da família File se torna genérica, podendo assim ser utilizada independente do sistemas de arquivos em questão.

Algumas das operações definidas para arquivos e diretórios são implementadas nesta família, uma vez que o contexto destas operações necessitam modificar diversas estruturas internas ao sistema de arquivos. As operações de exclusão de arquivos e diretórios é um caso típico. Ao excluir-se um arquivos do sistema de arquivos é necessária a atualização de diversos meta-dados no descritor do sistema de arquivos, assim como a desalocação dos blocos que este utiliza entre outros dados. Caso, esta operação fosse definida na interface da família arquivo, seria necessário garantir acesso a diversas informações (descritor do sistema de arquivos, alocadores de descriptores de

[4] Mapa de bits - um array de bits, onde cada bit representa o estado de cada posição do alocador

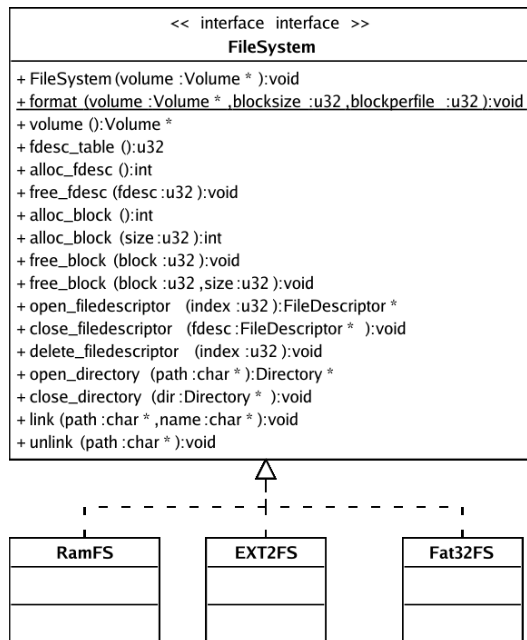

Figure 7: Interface inflada e membros da família FileSystem.

arquivos e blocos) que não são pertinentes ao arquivo em si, quebrando-se o encapsulamento de dados da programação orientada a objetos.

Internamente aos membros do sistema de arquivos é definido uma classe denominada FileDescriptor, responsável por fornecer acesso aos meta-dados necessários a implementação dos algoritmos presentes na família File. A interface desta classe pode ser visualizada na figura 8, assim como alguns membros da mesma.

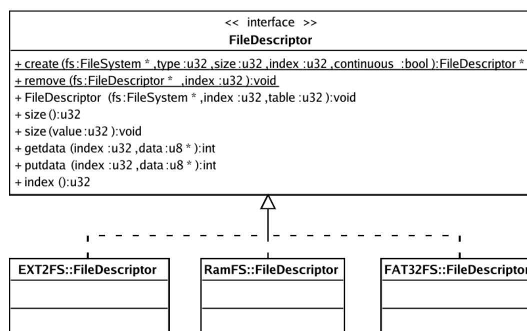

Figure 8: Interface dos descritores de arquivos.

Sendo assim, a família FileSystem além de implementar diversos métodos para algumas das operações sobre arquivos e diretórios, ainda implementa uma ”fábrica” de FileDescriptor que são utilizados pela família File, afim desta poder fornecer o acesso de leitura e escrita aos dados do arquivo. Esta funcionalidade é implementada através do método da interface inflada ***FileDescriptor * open_filedescriptor(index: u32)***.

Adicionalmente, três aspectos foram identificados nesta família, sendo eles o aspecto *Shared*, *Protection* e *Logging*, conforme descrito por [Fröhlich, 2001], aspectos compõem um cenário de execução do sistema, implementando diversas caracteríticas de configurabilidade do mesmo. Assim o aspecto *Shared*, diz respeito a todo o compartilhamento de arquivos entre processos implementando as necessidades em termos de algoritmo e estruturas para compartilhar

um arquivo. O aspecto *Protection* implementa os algoritmos de proteção de acesso ao arquivo, verificando por exemplo nas operações de abertura de arquivos, se determinado processo possui permissão de acesso ao mesmo, e em que modo (leitura/escrita). O aspecto *Logging* executa os algoritmos de realocação de meta-dados no atual segmento de log do sistema de arquivos, além de atualizar as estruturas em memória como o mapa de descritores de arquivos com a nova localização destes elementos.

3.5. A Família File

Esta família é o principal componente desta modelagem, sendo esta responsável pelo acesso efetivo aos arquivos do sistema. Sua modelagem foi efetuada visando ser a mais genérica possível, sem que peculiaridades de uma determinada especifícação de sistema de arquivos influencie sua interface. Desta maneira foi possível criar componentes realmente genéricos e que podem ser aproveitados independentemente do sistema de arquivo que está sendo utilizado.

Foi modelado três membros distintos nesta família, são eles: *ContinuousFile*, *RandomFile* e *CircularFile*, conforme mostra a figura 9. A principal diferença entre estes membros, se dá na forma como o arquivo é acessado pelo usuário.

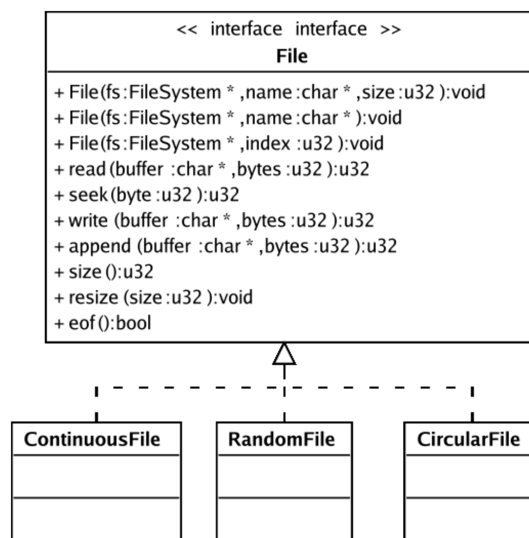

Figure 9: Interface inflada e membros da família File.

O membro *ContinuousFile* implementa um arquivo onde os dados são alocados de forma contínua no disco, seu acesso pode ser efetuado de forma randômica, contudo um tamanho inicial para o arquivo deve ser específicado no momento de sua criação e toda vez que o mesmo necessitar crescer em tamanho, uma tarefa custosa de realocar e copiar todo o arquivo para uma nova área contínua do disco pode ser efetuada.

O membro *RandomFile* implementa a idéia mais comum de arquivos que temos atualmente, um arquivo que pode ser alocado por blocos não consecutivos no disco, possuindo acesso randômico a estes.

O membro *CircularFile* implementa um tipo de arquivo bem específico e geralmente utilizados em arquivos de log. A ídeia básica para este arquivo é que o mesmo possui um tamanho fixo, e da mesma maneira que no membro ContinuousFile, pode ser aumentado ou reduzido através de uma operação custosa ao sistema, contudo este arquivo não possui um final. A partir do momento que é efetuada uma gravação no último bloco alocado a este, novos dados são armazenados no começo do arquivo, sendo que os dados que anteriormente estavam no início do arquivo são

perdidos. Essa é uma característica bastante interessante para ser adotada em arquivos de log de sistemas. Muitas vezes só existe a necessidade de manter armazenado as últimas informações de log de um sistema, não existindo a necessidade de armazenar dados antigos. Nestes casos um arquivo deste tipo pode ser utilizado com o objetivo de se manter apenas uma certa quantidade de logs recentes em um arquivo, garantindo também que o mesmo não irá crescer e ocupar um espaço maior que o estipulado pelo usuário ao arquivo, durante sua criação, ou através do redimensionamento do mesmo explícitamente pela chamada do método ***u32 resize(u32 size)***.

3.6. A Família Directory

Esta família é responsável pela implementação do recursos de nomes a arquivos. Conforme pode ser visto no capítulo 2, um sistema de arquivos não precisa necessáriamente possuir um nome para representá-lo dentro do conjunto de arquivos que formam o sistema, ele pode simplesmente ser acessado através do seu índice, dentro deste conjunto. Contudo caso a específicação necessite de uma nomeação de seus arquivos, isto é realizado pelos membros desta família.

Note que estes membros não precisam ser utilizados apenas para implementações de sistemas de arquivos. Outros componentes que possuam a característica fundamental de um sistema de diretórios (tradução de um nome para algum outro tipo de informação) podem utilizar este componente para realizar tais funções. Exemplos de sistemas que utilizam o conceito de diretórios são sistemas de traduçao de nomes na Internet, através do protocolo *Domain Name Server (DNS)* e sistemas de bancos de dados através do protocolo *Lightweight Directory Access Protocol (LDAP)*.

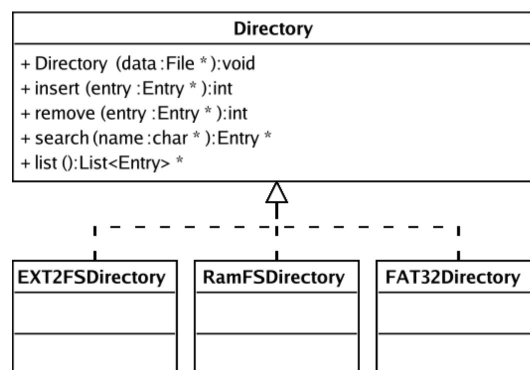

Figure 10: Interface inflada e membros da família Directory.

Basicamente, para cada específicação de sistema de arquivos, haverá um membro correspondente que implementa a especificação de diretórios do mesmo, conforme pode ser visto na figura 10. Embora não seja uma regra, a grande maioria de sistemas de arquivos implementam o sistema de diretórios como arquivos especiais presentes no disco. Com isso, cada sistema de arquivos irá definir uma formatação específica para o seu arquivo de diretório, e daí a necessidade de existir membros específicos a cada implementação.

Um sistema de diretórios é definido como um conjunto de entradas, cujo o conteúdo dessas pode variar de acordo com a aplicação do mesmo, e a estas são associados nomes que devem ser consideradas como chaves do sistema de diretórios. Em um mesmo diretório não é possível existir duas entradas com o mesmo nome.

Uma estrutura interna a cada membro da família Directory define o conteúdo de cada entrada do sistema de diretórios, sendo esta implementada internamente a cada membro da família na definição da classe *Entry*.

4. Protótipo implementado

O protótipo implementado para a validação da modelagem é um sistema de arquivos simples armazenado na própria memória do sistema. Suas estruturas de meta-dados são bem simples e fornecem apenas algumas operações básicas em sistemas de arquivos.

Nenhum tipo de controle de acesso, através de usuários e permissões foi especificado neste, assim como a utilização de dados de controle de tempo de criação, último acesso, etc. Basicamente os arquivos implementados neste sistema possuem apenas o atributo de tamanho e são nomeados através da abstração de diretórios.

Todo o controle de alocação de blocos do volume e índices das tabelas de descritores de arquivos são realizadas através de *Bitmaps* no volume, que possuem uma posicão fixa no mesmo e definida no superbloco do dispositivo.

O superbloco por sua vez, possue uma localização fixa no disco, estando sempre no primeiro bloco lógico do volume. A figura 11 mostra a disposição dos meta-dados e dados do sistema implementado.

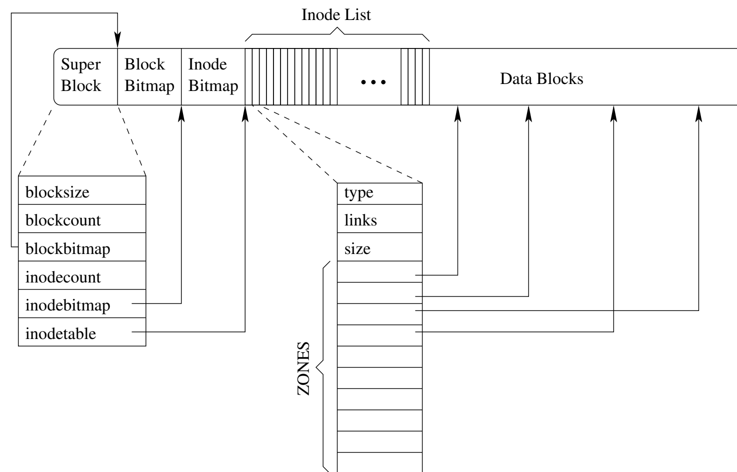

Figure 11: Estrutura geral da especificação implementada

O superbloco deste sistema possui as informações básicas para que os outros meta-dados possam ser localizados no disco. A primeira informação que o superbloco fornece é o tamanho do bloco lógico do sistema de arquivos. Este tamanho é definido pelo usuário do sistema no momento em que o mesmo está formatando o sistema de arquivos e é de fundamental importância a performance do sistema de arquivos.

Para ser capaz de localizar os outros meta-dados do sistema de arquivos, o superbloco possui uma referência ao bloco onde inicia os *bitmaps* de alocação de blocos, de descritores de arquivos e a tabela de descritores de arquivos. Adicionalmente o tamanho dessas estruturas (número de blocos e descritores presentes no sistema) são armazenadas no superbloco.

Os descritores de arquivos possuem um atributo do tipo de arquivo, que neste caso pode ser um arquivo de diretório ou um arquivo do usuário, além de um contador de referências a este arquivo, uma vez que o mesmo pode possuir mais de um nome em diversos diretórios distintos. Também possui um atributo que informa o tamanho do arquivo, em *bytes*.

O descritor deste sistema utiliza a alocação de blocos através de inodos, contudo não foi implementado neste protótipo a possibilidade de ocorrer referências duplas ou triplas, assim possuindo dez referências diretas a bloco de dados, este protótipo pode possuir arquivos no máximo

10 vezes maiores que o tamanho de cada bloco lógico.

A figura 12 ilustra a composição de um sistema de arquivos, através da seleção dos respectivos membros de cada família no sistema de arquivos.

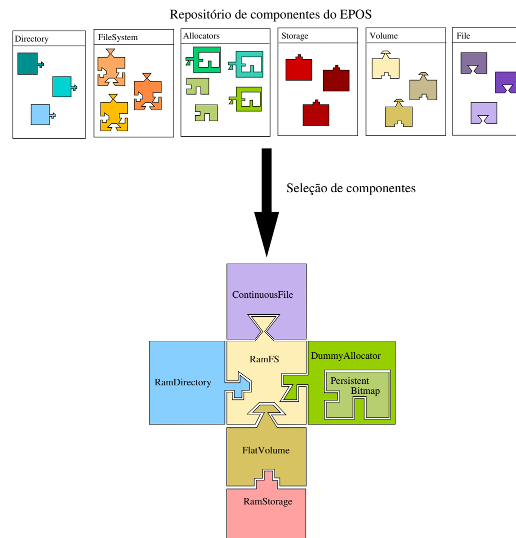

Figure 12: Composição de um sistema de arquivos

5. Conclusões

Um sistema de arquivos básico em memória RAM foi implementado dentro do sistema EPOS, oferecendo a possibilidade da aplicação armazenar e recuperar informações de uma mídia estruturada de forma organizada, mostrando-se viável a implementação de sistemas de arquivos baseados em componentes de software, através da metodologia de projeto de sistemas orientados a aplicação.

Destaco ainda como principal resultado deste trabalho, o grande favorecimento ao reuso de componentes de software decorrente da modelagem proposta, principalmente pela independência dos algoritmos presentes nos membros da família *File* e *FileSystem*, assim como a separação entre os dados utilizados pelos algoritmos de alocação de recursos e sua implementação, que é ainda mais favorecida pelo uso de meta-programação (*templates* da linguagem C++). Cabe ainda ressaltar que esses resultados puderam ser obtidos atráves da utilização das técnicas de desenvolvimento de sistemas orientados a aplicação.

Os resultados preliminares do protótipo desenvolvido geram sistemas com imagens do tamanho de cerca de 35 Kbytes, sendo que o código objeto dos componentes de sistemas de arquivos selecionados para o protótipo totalizam 19.348 bytes.

Do ponto de vista de conhecimento, a realização deste trabalho traz a comunidade científica uma proposta para a modelagem de sistemas de arquivos orientada a componentes de software que favorece o reuso e a customização de sistemas de arquivos de acordo com as necessídades específicas da aplicação, servindo assim como ponto de partida para futuros trabalhos de pesquisa nesse tema.

References

Barr, M. (1999). *Programming Embedded Systems in C and C++*. O'Reilly.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition.

Eckel, B. (2000). *Thinking in C++*. Planet PDF.

Fröhlich, A. A. M. (1994). Pyxis: Um sistema de arquivos distribuídos.

Fröhlich, A. A. M. (2001). *Application-Oriented Operating Systems*. GMD - Forschungszentrum Informationstechnik, 1 edition.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Johnson, R. E. and Foote, B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35.

Larman, C. (2001). *Applying UML and Patterns*. Unknown, second edition.

Marwede, P. (2003). *Embedded System Design*. Kluwer Academic Publishers.

Mullender, S. J. and Tannembaum, A. S. (1984). *Immendiate Files - Software: Practice and Experience*, volume 14.

OMG (1999). *OMG Unified Modeling Language Specification*. OMG.

Rosenblum, M. and Ousterhout, J. K. (1992). The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52.

Smolik, T. (1995). An object-oriented file system: an example of using the class hierarchy framework concept. *ACM SIGOPS Operating Systems Review*, 29(2):33–53.

Stroustrup, B. (1997). *The C++ Programming Language*. Addison-Wesley, 3 edition.

Tanenbaum, A. S. and Woodhull, A. S. (1997). *Operational Systems - Design and Implementation*. Prentice Hall, 2 edition.