FÁBIO RODRIGUES DE LA ROCHA

AMBIENTE DE CONCEPÇÃO PARA SISTEMAS DE ARQUIVOS DE TEMPO REAL EMBUTIDOS

FLORIANÓPOLIS
2003

UNIVERSIDADE FEDERAL DE SANTA CATARINA

CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

AMBIENTE DE CONCEPÇÃO PARA SISTEMAS DE ARQUIVOS DE TEMPO REAL EMBUTIDOS

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

FÁBIO RODRIGUES DE LA ROCHA

Florianópolis, Março de 2003.

AMBIENTE DE CONCEPÇÃO PARA SISTEMAS DE ARQUIVOS DE TEMPO REAL EMBUTIDOS

Fábio Rodrigues de la Rocha

'Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica, Área de Concentração em *Controle, Automação e Informática Industrial*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.'

Rômulo Silva de Oliveira, Dr.
Orientador

Edson Roberto De Pieri, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Rômulo Silva de Oliveira
Presidente

Carlos Barros Montez

Joni da Silva Fraga

Thadeu Botteri Corso

I hear and I forget, I see and I remember,
I do and I understand.

Provérbio chinês

AGRADECIMENTOS

Inicialmente, gostaria de agradecer ao meu orientador Rômulo Silva de Oliveira pelo tempo e atenção despendidos comigo nas dezenas de e-mails e reuniões. Também agradeço o seu empenho em melhorar o meu trabalho e me educar quanto a pesquisa científica.

Ao bolsista de IC, Guilherme Moreira, pelo auxilio na interface da ferramenta em Java.

Aos amigos e colegas de laboratório (Emerson, Ricardo, Fábio, Priscila, Cassia, Tercio, Neves, Tatiana, Obelheiro, Luciano, Karina, Patricia, Sam, Ana Paula, Carlos, Michelle e tantos outros....).

Gostaria de deixar registrado aqui que foi muito bom ter passado todo esse tempo com vocês. Sempre vou me lembrar das brincadeiras, dos cafés, das centenas de piadas e de tudo que nos fazia relaxar um pouco, fazendo do laboratório um lugar muito bom de se trabalhar e me impedindo de enlouquecer :-)

Aos amigos Baldo, Rui, Gesser e Leandro, este último que me aguentou dividindo apartamento durante um ano em Floripa.

A minha familia, principalmente a minha mãe.

Ao CNPq pelo apoio financeiro.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia Elétrica.

AMBIENTE DE CONCEPÇÃO PARA SISTEMAS DE ARQUIVOS DE TEMPO REAL EMBUTIDOS

Fábio Rodrigues de la Rocha

Março/2003

Orientador: Rômulo Silva de Oliveira
Área de Concentração: Controle, Automação e Informática Industrial
Palavras-chave: Tempo Real, Sistemas Embutidos, Sistemas de Arquivos
Número de Páginas: xiv + 116

No mundo atual, sistemas embutidos estão em toda a parte. Eles estão nos carros, indústrias, bens de consumo, equipamentos médicos e a cada dia conquistam mais espaço. Com o desenvolvimento da tecnologia e motivados por um mercado consumidor cada vez mais exigente, os dispositivos outrora simples tornaram-se cada vez mais complexos.

Para auxiliar no desenvolvimento de uma classe de dispositivos embutidos que necessita armazenar informações, propõe-se a criação de um sistema de arquivos voltado para esse tipo de aplicação.

Em sua implementação, ele faz uso de algoritmos e estruturas de dados que o tornam propício a ser utilizado em sistemas com requisitos de tempo real e em dispositivos com restrições de memória. Além disso, permite o uso de uma interface visual que age como um *front-end*, simplificando a tarefa de criação de sistemas de arquivos.

Como principais benefícios dessa ferramenta de *software*, temos a redução no tempo de desenvolvimento de um projeto, no seu custo final e no tempo que um produto leva até chegar ao mercado.

Abstract of Thesis presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

EMBEDDED REAL-TIME FILE SYSTEM CONCEPTION TOOL

Fábio Rodrigues de la Rocha

March /2003

Advisor: Rômulo Silva de Oliveira
Area of Concentration: Control, Automation and Industrial Computing
Key words: Real-Time, Embedded Systems, File Systems
Number of Pages: xiv + 116

Nowadays, embedded systems are everywhere. They are in cars, industries, consumer goods, medical equipment and every day they conquer more space. Technological improvements and an increasingly demanding market have led once simple devices to become complex ones.

To assist the development of embedded devices that need to store data, a new file system has been created to support these devices.

The algorithms and data structures used are suitable for environments with real-time requirements and memory constraints. Moreover, it allows the use of a visual interface that acts as a front-end, simplifying the file systems creation task.

As main advantages, we have reduction in both costs and time needed to develop a project, which leads to a smaller time to market.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Código

Capítulo 1

Introdução

Com o desenvolvimento da micro-eletrônica nas décadas de 70 e 80, ocorreu uma proliferação do uso de microprocessadores e microcontroladores. Esses dispositivos substituíram milhares de componentes eletrônicos discretos e permitiram que dispositivos programáveis de baixo custo fossem produzidos e utilizados para controlar equipamentos.

Inicialmente, o alvo desses dispositivos programáveis foi a automação de indústrias. Nela, os sistemas microprocessados/microcontrolados eram compostos, além de seu processador, por memórias, relógios, sensores e atuavam sobre válvulas, motores, temperatura de caldeiras, sistemas de alarme, balanças de precisão, etc. O *software* que existia dentro dos dispositivos era bastante simples e escrito em linguagem de baixo nível (*assembly*) (Barr, 1999).

O passo seguinte, foi a incorporação desta tecnologia em bens de consumo, tais como televisores, videocassetes, carros, máquinas de lavar roupas, fornos de micro-ondas, telefones celulares, videogames, etc (Barr, 1999; Berger, 2002).

Com o objetivo de suprir a demanda sempre crescente de integração entre dispositivos (principalmente após a popularização das redes de computadores) e suportar diversos recursos que o mercado cada vez mais exigente necessitava, grandes progressos foram feitos nesta área.

Os dispositivos outrora simples, tornaram-se cada vez mais complexos, com principal ênfase no *software* responsável pela sua funcionalidade. Um bom exemplo disto está no desenvolvimento dos telefones celulares, hoje bastante difundidos. Os primeiros modelos eram bastante simples e somente transmitiam voz. Com o desenvolvimento da tecnologia, recursos como agenda telefônica, senhas de acesso, jogos, envio de *e-mail*, acesso à Internet e câmera fotográfica foram adicionados.

Neste cenário, o *software* que inicialmente era composto por centenas de linhas de código em linguagem de baixo nível passou a ser composto por dezenas ou centenas de milhares de linhas de código em linguagem de alto nível. Não é mais possível desenvolver esses produtos da mesma forma que no passado, o grau de especialização requer o uso de novas ferramentas e abstrações de *software* (Berger, 2002).

Para o mercado, parece bastante improvável que alguém escolha adquirir um produto simples em detrimento de um muito mais avançado. Dessa forma, o fator econômico tem funcionado como um motor, impulsionando as mudanças tecnológicas. Para continuar na vanguarda dessa tecnologia, mudanças estão constantemente sendo feitas na forma de produzir *software* para esses dispositivos, com ganhos em qualidade e tempo de desenvolvimento.

1.1 Motivação

O mercado de dispositivos embutidos[1] é bastante competitivo, diversos fabricantes ao redor do mundo concorrem para produzir os dispositivos mais econômicos e com mais recursos (funções que estes possuem).

Além desses dois fatores, um outro bastante importante é o tempo que um produto leva até chegar ao mercado (Martin e Schirrmeister, 2002). De nada adiantaria projetar um dispositivo econômico e com muitos recursos inéditos, se este somente estará disponível num futuro distante (onde possivelmente o mercado já estará saturado ou exigirá outros recursos). Neste mercado, o sucesso de um produto depende fundamentalmente da ocorrência simultânea destes três fatores.

Muitas das aplicações que existem nos dispositivos embutidos necessitam armazenar dados durante a sua execução. Podemos imaginar, por exemplo, uma agenda eletrônica que deve armazenar compromissos, um GPS (*Global Positioning System*) que deve manter um histórico de coordenadas, um leitor de código de barras que armazena os dados obtidos e posteriormente produz um relatório de saída, etc.

Em todas essas aplicações, é necessário um sistema para manter essas informações. Desenvolver um *software* específico para tratar o armazenamento de informações é um tarefa que demanda muito tempo e fazer isto partindo do zero para cada um dos projetos é pouco produtivo.

Além disso, o projetista que desenvolve um produto, por exemplo um telefone celular, tem que se preocupar com a complexidade do problema que ele se propõe a resolver, ele não deveria perder tempo desenvolvendo *software* que não é o objetivo principal do projeto.

O ideal seria abstrair, do projetista da aplicação, o armazenamento de informações, escondendo-o sobre uma camada de *software* denominada **sistema de arquivos**. Infelizmente, o uso de sistemas de arquivos de propósito geral, tais como os existentes nos computadores pessoais não seria vantajoso, visto que estes são concebidos sem levar em consideração as necessidades e características das aplicações que fazem uso do sistema de arquivos, além de utilizarem muita memória, recurso geralmente escasso em dispositivos embutidos.

[1] Dispositivos programáveis concebidos para realizar uma função específica são também conhecidos como dispositivos embutidos (*embedded devices*) e serão vistos em detalhe no capítulo 3.

1.2 Justificativa

Neste trabalho, é proposta uma forma de auxiliar o desenvolvimento de *software* para dispositivos embutidos que necessitem de um sistema de arquivos. A proposta é exemplificada através de uma ferramenta de *software* para produzir sistemas de arquivos, isto é, uma ferramenta que gera código fonte em C para um sistema de arquivos personalizado.

A linguagem C foi escolhida pois ela é considerada um padrão para o desenvolvimento de *software* em aplicações embutidas, bem como *software* básico. Esta tem como principais vantagens a simplicidade, portabilidade, eficiência e facilidade com que dispositivos de *hardware* são manipulados.

Com esta ferramenta, um projetista de *software* pode, em questão de minutos criar um sistema de arquivos que será utilizado pela aplicação existente no dispositivo. O SA (Sistema de Arquivos) personalizado é concebido levando-se em consideração muitos detalhes da aplicação final, com o objetivo de perfeitamente adaptar-se a esta.

Além disso, o SA é feito para lidar com restrições de tempo real (prevendo o seu uso em sistemas operacionais de tempo real) e de memória, pois ambos estão freqüentemente presentes nos projetos atuais. Como vantagens principais desta ferramenta, temos uma redução acentuada no esforço de desenvolvimento de *software*, redução no tempo de depuração, otimização do uso de memória e redução do custo final do produto e do tempo necessário até ser lançado no mercado.

1.3 Objetivos

O principal objetivo deste trabalho é produzir um ambiente de desenvolvimento de sistemas de arquivos voltados para aplicações embutidas de tempo real. Esse ambiente (ferramenta de *software*) será portável e produzirá código fonte em C de um SAP (Sistema de Arquivos Personalizado). As principais características dos sistemas de arquivos gerados são:

- Codificado usando apenas *ANSI* C. Como os fontes do sistema de arquivos poderão ser compilados para diversos tipos de dispositivos, a escolha de um padrão amplamente aceito como o *ANSI* C, para o qual existem compiladores nos mais diversos processadores é um ponto positivo. Além disso, a linguagem C não possui construções que conhecidamente possuem um grande *overhead*, como o operador `new`, as *templates*, tratamento de exceções e identificação de tipos em tempo de execução, presentes em C++;
- Implementa uma *API* que se aproxima do padrão POSIX 1003.1. Como a *API* é a parte com a qual o programador tem contato, é importante que esta seja bem conhecida, para evitar o desperdício de tempo aprendendo uma nova *API*. Desta maneira, foram implementadas rotinas de acesso ao sistema de arquivos que se aproximam deste padrão, dada a sua grande aceitação;

- Suporta acesso de várias *threads*. As rotinas do sistema de arquivos são criadas para permitir várias linhas de execução num mesmo trecho de código, para melhor aproveitar os recursos do dispositivo. Ele utiliza semáforos binários para controlar o acesso a seções críticas de código;
- Minimiza as inversões de prioridade, pois os algoritmos são criados tendo em mente que as seções críticas existentes no código devem ser pequenas, minimizando o tempo de bloqueio nos semáforos e maximizando a concorrência das *threads*;
- Composto somente com os blocos de código necessários para a funcionalidade exigida, dessa forma é sempre gerado um código reduzido;
- Usa estruturas de dados e algoritmos escolhidos para adaptar-se a cada tipo de aplicação final;
- O código gerado utiliza algoritmos com complexidade de pior caso conveniente e explícita, desta forma, é propício a ser usado em sistemas de tempo real.

1.4 Organização do texto

Para facilitar o entendimento deste trabalho, o texto foi dividido nos seguintes capítulos que conduzem o leitor à sistemática do problema, a uma visão da situação atual e às vantagens da solução proposta. Sendo assim, temos:

capítulo 2: Sistemas de arquivos Neste capítulo será feita uma introdução ao tema, abordando um sistema de arquivos de uma forma genérica (sem levar em consideração características específicas de implementação para um determinado *hardware*), tanto na visão do usuário do sistema, como na visão do implementador;

capitulo 3: Aspectos de programação de sistemas embutidos de tempo real Neste capítulo serão caracterizados os sistemas embutidos e sua forma de construção. Também serão mostrados os principais problemas que existem na programação de sistemas embutidos com restrições de tempo real;

capítulo 4: Sistema de arquivos desenvolvido Neste capítulo será feita uma discussão das características do sistema de arquivos proposto neste trabalho, seus algoritmos, estruturas de dados e a motivação para seu uso;

capítulo 5: Ferramenta desenvolvida Neste capítulo será discutida a ferramenta de concepção de SA, uma ferramenta de alto nível com a qual pode-se criar um sistema de arquivos somente utilizando-se uma interface gráfica. Seu uso é demonstrado através de exemplos;

capítulo 6: Experiências e resultados Neste capítulo serão apresentadas algumas experiências realizadas com a ferramenta de composição, sobre o sistema de arquivos gerado e sobre uma aplicação exemplo;

capítulo 7: Conclusão Apresenta as conclusões deste trabalho, as vantagens e contribuições do trabalho para a área;

apêndice A: Interface de programação completa em C Apresenta a interface de programação do sistema de arquivos, com os nomes, parâmetros das funções e explicações detalhadas destas;

apêndice B: Diagramas de funções do SA Neste capítulo são apresentados diagramas que mostram as funções disponíveis na *API* do SA e suas chamadas às funções internas ao sistema de arquivos. Também são feitos comentários sobre as seções críticas presentes nas funções;

Glossário Definição de termos técnicos ou siglas utilizadas no trabalho.

Capítulo 2

Sistemas de arquivos

Historicamente, computadores foram concebidos para realizar cálculos complexos operando sobre um conjunto pequeno de dados de entrada. Com o desenvolvimento da micro-eletrônica, foi possível construir máquinas capazes de manter grandes conjuntos de dados de entrada e armazenar diversos resultados de saída para uso posterior. A partir deste momento, computadores ganharam uma nova função, o armazenamento de informações.

O aumento da capacidade dos dispositivos permitiu armazenar todo o tipo de informação e aliado à capacidade de processamento sempre crescente, propiciou uma solução para diversos problemas. Porém, em contrapartida, levou a um outro: "Como encontrar uma certa informação dentre tantas outras e como manipulá-la"

Para lidar com este problema foi criado o conceito de sistema de arquivo, uma camada de *software* responsável pelo armazenamento e recuperação de dados de uma forma transparente e cômoda para o usuário final. Com um sistema de arquivos não é preciso saber como a informação está armazenada, nem mesmo onde ela está localizada. O acesso à informação é simplesmente feito através de uma referência.

2.1 Interface do sistema de arquivos

A interface do sistema de arquivos refere-se à forma com que o usuário percebe e utiliza o Sistema de Arquivos. Nela, estão especificados como estes são acessados pelo usuário, que operações são possíveis e quais as funções destas.

2.1.1 Nomes de arquivos

Nos sistemas de arquivos, o usuário referencia seus dados através de uma entidade abstrata chamada arquivo e este é identificado por um nome. O nome é geralmente um cadeia de caracteres como

"exemplo.c " dada a facilidade que temos em lidar com palavras ao invés de números. Em alguns sistemas é feita distinção entre letras maiúsculas e minúsculas, em outros não.

Os sistemas de arquivos armazenam seus dados geralmente em dispositivos não voláteis, isto é, o conteúdo dos arquivos é permanentemente disponível mesmo após o término da operação na qual ele estava envolvido, sobrevivendo a quedas de energia[1] e reinicialização do sistema. Em alguns casos, nitidamente para aplicações específicas, o dispositivo de armazenamento pode ser volátil, mas seu conteúdo irá permanecer intacto enquanto ele for necessário para realizar o trabalho proposto e somente depois este será apagado.

Na maioria dos sistemas de arquivos, o arquivos são compostos de duas partes separadas por um ponto "." como em "exemplo.c ". A primeira parte é denominada nome dio arquivo e a segunda parte é chamada extensão. A extensão serve para indicar uma característica do arquivo. Em alguns sistemas a extensão, assim como o ponto "." fazem parte do nome do arquivo (Tanenbaum, 1992). A tabela 2.1 mostra algumas extensões com seus significados.

Extensão	**Significado**
.bak	*Backup* de um arquivo
.bas	Programa-fonte escrito em linguagem Basic
.c	Programa-fonte escrito em linguagem C
.doc	Arquivo de documentação
.hlp	Arquivo de ajuda
.lib	Arquivo de biblioteca, usado pelo ligador (linker)
.obj	Arquivo objeto, resultado de uma compilação
.tex	Arquivo fonte formatado em TEX
.txt	Arquivo de texto genérico
.html	Arquivo em *HyperTextMarkupLanguage*

Tabela 2.1: Extensões típicas e seus significados

2.1.2 Atributos de arquivos

Além de possuir um nome, os sistemas de arquivos fornecem atributos para os arquivos. Atributos são características especiais que geralmente estão associadas a determinadas funções. Os atributos variam de sistema para sistema, mas geralmente possuem um conjunto básico, tal como representado na tabela 2.2

2.1.3 Métodos de acesso a arquivos

O acesso a arquivos refere-se ao modo como os dados de um arquivos podem ser acessados. Existem duas formas básicas de acesso, seqüencial e direto (ou relativo) (Tanenbaum, 1992; Silberschatz e Galvin, 1997).

[1] Desconsiderando os casos em que as informações tenham sido corrompidas.

Atributo	Significado
Tipo	Informação necessária nos sistemas que suportam diferentes tipos de arquivos, baseado neste tipo o sistema operacional pode tratar diferentemente os arquivos. Os tipos possíveis são: leitura, leitura/escrita, oculto, sistema, *Backup*, *ASCII*, binário, etc.
Proteção	Utilizado para especificar quem pode ter acesso ao arquivo.
Criador	Identificação de quem criou o arquivo.
Proprietário	Identificação do dono atual do arquivo.
Backup	Informa de foi feito *Backup* do arquivo.
Tamanho atual	Número de bytes do arquivo.
Tamanho máximo	Número de bytes que o arquivo pode possuir no máximo.
Data criação	Data em que o arquivo foi criado.
Último acesso	Data em que o arquivo foi acessado pela última vez.
Última mudança	Data em que o arquivo foi alterado pela última vez.

Tabela 2.2: Atributos mais utilizados

Acesso seqüencial Neste método, a informação é acessada seqüencialmente registro após registro ou byte após byte. Uma operação de *read* lê um dado e automaticamente incrementa um indicador de posição corrente para o próximo dado. Assim, caso fosse necessário acessar um dado localizado no final do arquivo seria preciso ler o arquivo inteiro, conforme mostra a figura 2.1. De forma semelhante, a rotina *write* escreve dados no final do arquivo e posiciona a próxima escrita uma posição adiante.

O acesso seqüencial é baseado no funcionamento dos dispositivos de fita magnética que possuem um comportamento puramente seqüencial. Como historicamente estes dispositivos foram desde cedo utilizados nos computadores, os sistemas de arquivos forneciam este método de acesso apenas. Mesmo hoje, algumas aplicações continuam utilizando apenas acesso puramente seqüencial, tais como editores de texto e compiladores. Além das rotinas de *read* e *write*, os sistemas de arquivos geralmente forneciam uma rotina *rewind* que tinha a função de deslocar o indicador de posição corrente do arquivo para o início, assim as leituras poderiam ser feitas no início do arquivo novamente.

Acesso direto ou relativo Neste método, o arquivo é constituído de registros de tamanho fixo e estes podem ser lidos ou escritos em qualquer ordem. O método de acesso direto, também conhecido como acesso aleatório ou acesso relativo é baseado no modelo do disco magnético, pois os discos magnéticos permitem acesso direto a qualquer bloco do disco. O funcionamento deste método é mais complexo que o seqüencial. O arquivo é visto como uma seqüência numerada de registros ou blocos. Uma operação de leitura pode ler o bloco 10 e logo depois ler o bloco 1 e depois disso, escrever o bloco 2.

Como neste método pode-se acessar qualquer posição do arquivo, existe a necessidade de um método para indicar essa posição. Uma forma seria incluir na rotina *read* e *write* um campo que indicasse a posição. Mas a forma geralmente utilizada pelos sistemas de arquivos consiste em fornecer uma outra rotina, chamada *seek* que tem a função de mudar o indicador de posição relativa do arquivo para o valor informado. Assim, antes de efetuar uma operação, é necessário ajustar a posição que será lida/escrita com a rotina *seek*.

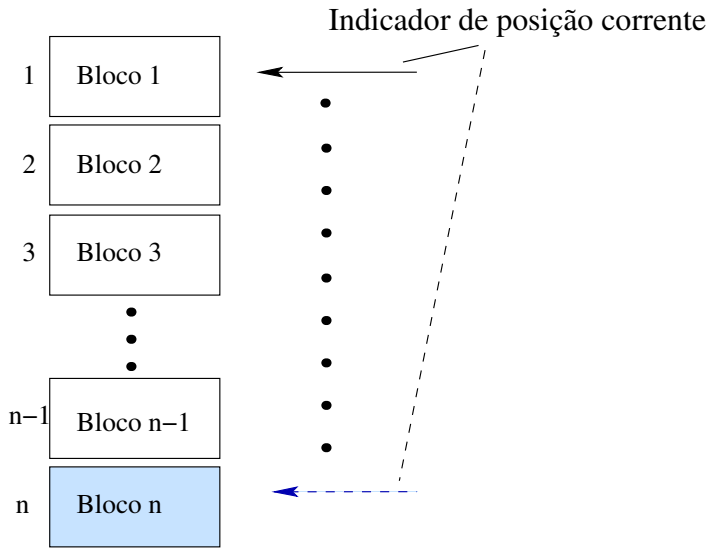


Figura 2.1: Acesso seqüencial ao último registro

O método de acesso direto é mais geral que o método puramente seqüencial e desta forma o acesso seqüencial pode ser facilmente simulado com o acesso direto, mas o contrário não é verdadeiro. É extremamente ineficiente simular um acesso direto com um acesso seqüencial.

O método de acesso direto também pode ser utilizado para fornecer outros métodos de acesso. Utilizando as mesmas rotinas de *read*, *write* e *seek* pode-se implementar métodos indexados. Neste método de acesso, além de possuir os dados armazenados no arquivo, existe um outro arquivo, chamado arquivo de índices que possui ponteiros para os registros do arquivo de dados. Pesquisando no arquivo de índices, encontra-se a posição da informação desejada no arquivo de dados e assim basta posicionar o arquivo de dados e efetuar a leitura. Os arquivos de índices são geralmente mantidos ordenados e uma pesquisa binária é utilizada para acelerar a procura pelo registro. Diversos algoritmos são utilizados para otimizar a inserção e remoção de registros e acelerar a pesquisa.

2.1.4 Operações sobre arquivos

Os sistemas de arquivos fornecem diversas operações para armazenamento e recuperação de dados em arquivos, as operações mais comuns segundo (Tanenbaum, 1992) são:

create - permite a criação do arquivo sem dados e especifica alguns de seus atributos.

delete - elimina um arquivo do sistema de arquivos, assim o espaço que este anteriormente ocupava é liberado para uso.

open - a rotina *open* é utilizada para abrir o arquivo e deixá-lo disponível para as outras operações, tais como *read* e *write*. Na abertura de um arquivo, o sistema de arquivos lê os atributos do arquivo e busca a lista de endereços de blocos para que o acesso a estes seja efetuado mais rapidamente.

close - Fecha um arquivo.

read - como já mencionado, *read* lê os bytes da posição corrente do arquivo, o processo que faz a leitura das informações deve informar qual a quantidade de bytes serão lidos e providenciar um região para comportar estas informações.

write - escreve dados na posição corrente do arquivo, se a posição corrente for no final do arquivo, os dados escritos serão concatenados e assim o tamanho do arquivo será aumentado. Se a posição corrente estiver apontando uma região anterior ao fim do arquivo, os dados que anteriormente existiam nesta posição serão sobrescritos e perdidos.

seek - permite ajustar o indicador de posição relativa do arquivo. A partir desta posição os dados serão lidos ou escritos.

append - tem a função de adicionar dados somente no final do arquivo. Esta rotina pode ser implementada utilizando-se a rotina *seek* para posicionar o indicador de posição corrente para o final do arquivo e depois utilizar *write* para escrever o dado.

get_attributes - utilizado para obter os atributos de um arquivo.

set_attributes - utilizado para ajustar os atributos de um arquivo, mas nem todos os atributos podem ser modificados.

rename - utilizada para modificar o nome de um arquivo.

2.1.5 Diretórios

Diretórios são estruturas que contém um conjunto de arquivos. A forma mais simples é prover o SA com somente um único diretório contendo todos os arquivos do sistema, como mostrado na figura 2.2. Diretórios dessa forma somente permitem nomes de arquivos únicos em todo o sistema, isto é, não seria possível utilizar o mesmo nome de arquivo designando arquivos diferentes.

Mesmo com limitações, essa forma é bastante simples de se implementar e representa uma boa escolha para aplicações que não utilizam muitos arquivos, tais como aplicações para dispositivos embutidos.

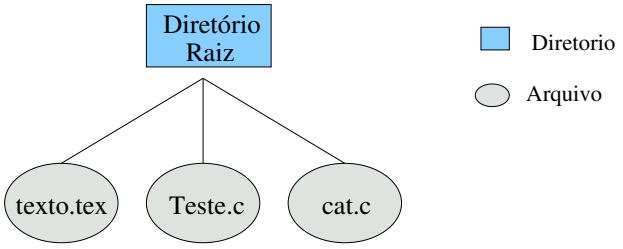


Figura 2.2: SA com somente um diretório

Uma generalização dessa idéia é estabelecer uma hierarquia, através de uma árvore de diretórios. Assim, pode-se ter tantos diretórios e subdiretórios quanto desejado. Essa forma de representar diretórios está ilustrada na figura 2.3.

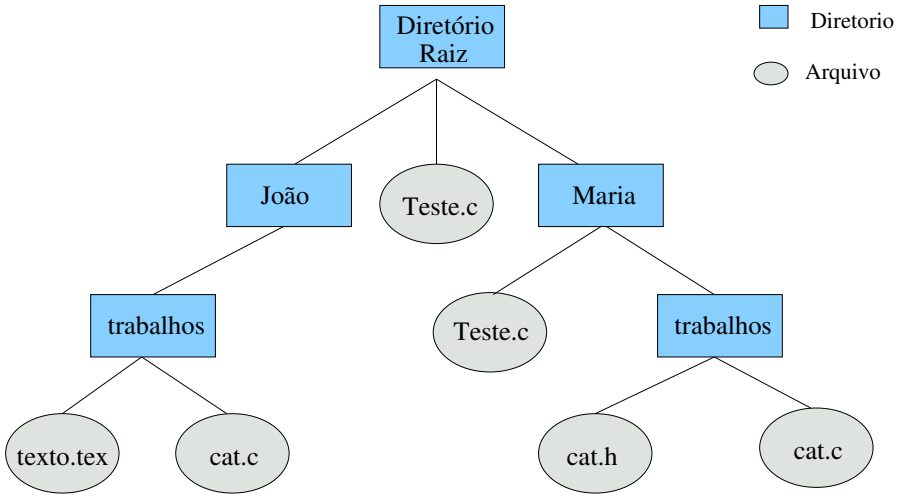


Figura 2.3: Sistema de diretórios hierárquico

2.1.6 Exemplo de uso

Nesta seção são apresentados exemplos em linguagem C (Ritchie e Kernighan, 1988) do uso das rotinas de um sistema de arquivos. O primeiro exemplo mostrado no Código 1 constitui-se apenas de um programa que abre um arquivo informado pela linha de comando e mostra o seu conteúdo na tela (ex: `cat teste.c`).

Como demonstrado neste exemplo, após a chamada *open* ser executada com sucesso, o programa executa um laço infinito. A função *open* possui dois parâmetros, o primeiro é o nome do arquivo a ser aberto e o segundo parâmetro refere-se ao modo de abertura (somente leitura). Esta função retorna um valor inteiro que representa um descritor de arquivo, isso é uma referência para o arquivo em disco. Assim, todo o acesso ao arquivo é feito utilizando-se esse descritor.

Dentro do laço *while* é feita a leitura de dados do arquivo com a função *read* e posteriormente estes são impressos por *printf*. A função *read* possui 3 parâmetros, o primeiro refere-se ao descritor de arquivo, o segundo é a região de memória na qual serão armazenados os dados lidos e o terceiro parâmetro é a quantidade de bytes que devem ser lidos. Esta função retorna o número de bytes que efetivamente foram lidos, em caso de fim de arquivo retorna zero e em caso de erro retorna um número negativo.

O retorno da função *read* é utilizado para determinar o ponto em que o programa deve terminar, pois neste caso a função *read* retorna zero e o comando *break* é executado causando a saída do laço e posteriormente a execução da rotina *close* que fecha o arquivo.

```
#define  BUFFER_SIZE 100

int main (int argc, char *argv[])
{
   int src, in;
```

```
    char  buff [BUFFER_SIZE];


    if (argc != 2) exit(1);

    src  = open ( argv[1], O_RDONLY);
                                /* Abre o arquivo somente para    */
                                /* leitura .                      */
    if (src < 0 ) exit ( 2 );   /* Em caso de erro na abertura .  */

    while  (1)
    {
         in = read (src, buff, BUFFER_SIZE);
                                /* Le os elementos do arquivo     */
                                /* de entrada                     */
                                /* a funcao read retorna o numero */
                                /* de elementos lidos.            */
         if (in <= 0) break ;   /* Sair .                         */

         printf ( "%d", in );
    }
    close (src);                /* Fecha o arquivo                */

    return  0;
}
```

Código 1: Exemplo do programa cat.c

O segundo exemplo mostrado no Código 2 constituí-se de um copiador de arquivos. São passados dois parâmetros para este programa, o primeiro especifica o arquivo de origem e segundo o arquivo de destino (ex: cp teste.c teste.new). O arquivo de origem é aberto com *open* da mesma forma que no Código 1.

O arquivo de saída é criado com a função *creat* que neste caso tem dois parâmetros, o primeiro é o nome do arquivo que será criado e o segundo é o modo de criação, isto é, as permissões que ele possui (segundo as permissões do sistema de arquivos do UNIX). Essa função retorna um inteiro positivo em caso de sucesso e um inteiro negativo em caso de erro. O restante do programa é semelhante ao programa anterior, usando *read* para ler os dados do arquivo, mas ele utiliza *write* para escrever os dados no arquivo destino. A função *write* possui 3 parâmetros, o primeiro é o descritor de arquivo, o segundo é a região de memória onde estão as informações e o terceiro é o número de bytes que serão escritos.

Esta função retorna o número de elementos que foram escritos corretamente ou um número negativo em caso de erro. Tanto as condições de erro como o fim de arquivo encontrado (quando da leitura) fazem que o comando *break* seja executado causando uma saída do *while* e execução das

rotinas *close* que fecham os arquivos.

```
#define  BUFFER_SIZE 4096
#define  MODE 0666

int main (int argc, char *argv[])
{
   int src, dest, in, out;
   char buff [BUFFER_SIZE];

   if (argc != 3) exit(1);

   src  = open ( argv[1], O_RDONLY);
                                /* Abre o arquivo somente para   */
                                /* leitura .                     */
   if (src < 0 ) exit ( 2 );    /* Em caso de erro na abertura . */
   dest = creat (argv[2], MODE); /* Tenta criar o arquivo com as */
                                /* as permissoes MODE.           */
   if (dest < 0 ) exit(3);      /* Arquivo nao pode ser criado.  */

   while (1)
   {
        in = read (src, buff, BUFFER_SIZE);
                                /* Le os elementos do arquivo    */
                                /* de entrada                    */
                                /* a funcao read retorna o numero */
                                /* de elementos lidos.           */

        if (in <= 0) break ;    /* Sair .                        */
        out = write(dest, buffer, in);
                                /* Escreve o numero de elementos
                                /* lidos .                       */
        if (out <= 0) break ;   /* Se nao conseguiu escrever .   */
   }
   close (src);                 /* Fecha os arquivos             */
   close (dest);
   return  0;
}
```

Código 2: Exemplo do programa `cp.c`

2.2 Implementação do sistema de arquivos

Nesta seção serão examinados os métodos para alocar memória para os arquivos e os métodos para controlar o espaço que está livre. Mais informações sobre estes métodos podem ser encontradas em (Tanenbaum, 1992; Silberschatz e Galvin, 1997; Oliveira et al., 2000).

2.2.1 Métodos de alocação

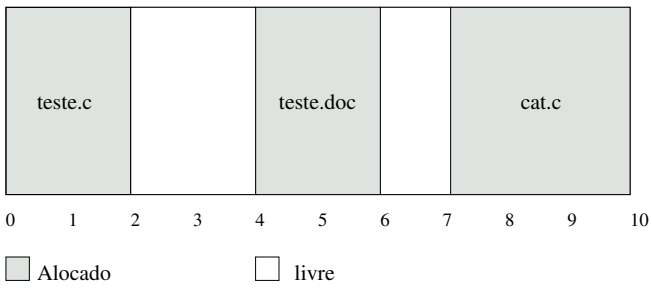
Os métodos de alocação referem-se às formas utilizadas para reservar espaço para arquivos no sistema. Os métodos empregados para esta tarefa têm, cada um, seus pontos positivos e negativos.

Em geral, os sistemas de arquivos escolhem apenas um dentre esses métodos para implementar, mas isto não elimina a possibilidade de sistemas de arquivos especializados implementarem vários métodos para prover mais recursos para uma aplicação específica.

Alocação contígua ou fixa

Nesta forma de alocação, um arquivo consiste de um conjunto de segmentos contíguos no dispositivo de armazenamento, isto é, um arquivo pode possuir os blocos seqüenciais tais como 7, 8 e 9. Assim, um arquivo está associado a um endereço inicial (por exemplo o bloco número 7) e a um número de blocos que este arquivo ocupa (tal como 3). Essa forma de alocar arquivos está ilustrada na figura 2.4.

O acesso ao arquivo é simples, para tanto basta pesquisar na tabela da figura 2.4 pelo nome do arquivo e utilizar o endereço de início para ler o bloco inicial deste arquivo e utilizar o campo de quantidade de blocos para saber quantos serão acessados.



Nome	Bloco inicial	Quantidade
teste.c	0	2
teste.doc	4	2
cat.c	7	3

Figura 2.4: Associação de arquivos contíguos

Os principais problemas deste método são:

Encontrar espaço livre No diagrama de memória da figura 2.4 podemos ver que temos duas regiões de memória livre. Para encontrar essas regiões precisa-se utilizar algoritmos de gerenciamento de partições variáveis, como o *first-fit, next-fit, best-fit* entre outros (Tanenbaum, 1992; Johnstone e Wilson, 1999) (mais conhecidos por seu uso em gerência de memória principal). No

caso em que se utilize uma lista encadeada, esta deve ser percorrida até que se encontre um espaço livre e que o número de seja maior ou igual ao desejado. Caso o número de blocos seja maior, parte deste será alocado para o arquivo e o restante será inserido na lista de blocos livres.

A implementação desse algoritmo é bastante custosa, visto que no pior caso será necessário uma pesquisa da lista inteira, além do fato que esta lista deve ser mantida ordenada por endereço.

Tamanho fixo Quando um arquivo é criado, seu espaço é reservado com um tamanho máximo, isto é, deve-se escolher o tamanho máximo do arquivo e este não poderá ser aumentado facilmente, dadas as características do método de alocação.

Muitas vezes é difícil determinar o tamanho máximo que um arquivo deve possuir. Se o arquivo for criado com um tamanho muito pequeno, em pouco tempo teremos o problema de falta de espaço, já se o tamanho for superestimado, haverá um potencial desperdício de espaço.

Como vantagens, temos a simplicidade de implementação e ausência de operações de cálculo de endereço que contribui para um maior desempenho das operações de entrada e saída. Essa estrutura se enquadra bem em sistemas nos quais a criação e destruição de arquivos não é freqüente.

Em sistemas embutidos *(embedded systems)* arquivos deste tipo poderiam ser utilizados para armazenar programas executáveis, assim todo o programa estaria seqüencialmente armazenado no sistema de arquivos e seria bastante simples copiá-lo para a memória para ser executado. Se o dispositivo físico utilizado para armazenar fosse a memória *RAM*, o sistema operacional teria o programa num endereço linear, pronto para ser posto em execução.

Alocação Encadeada

Para lidar com alguns dos problemas vistos anteriormente, foi desenvolvida uma outra estratégia de alocação de memória. Nesta, os arquivos são compostos por blocos de dados que não precisam estar dispostos em seqüência na unidade de armazenamento. Assim, um arquivo poderia muito bem ser composto pelos blocos 10, 1, 3 ,6 , etc.

Além de armazenar informações, os blocos dos arquivos também armazenam um ponteiro para o próximo bloco, formando assim uma lista encadeada como ilustrado na figura 2.5.

Inicialmente o arquivo tem tamanho nulo, quando é feita uma operação de escrita sobre este arquivo, verifica-se que não existe espaço suficiente para armazená-la. Então um bloco qualquer da lista de blocos livres é alocado e utilizado. No caso de mais um bloco ser necessário (em virtude de mais dados terem sido escritos) outro bloco é alocado e o ponteiro do bloco anterior é feito apontar para o último bloco alocado.

Essa estratégia possui alguns problemas, tais como gasto de memória em decorrência do armazenamento de ponteiros para blocos e a própria forma de lista encadeada (Silberschatz e Galvin, 1997). Esse espaço ocupado pelos ponteiros mesmo sendo pequeno em comparação com o montante de informação armazenada, não pode ser desconsiderado. O outro ponto problemático refere-se à forma de

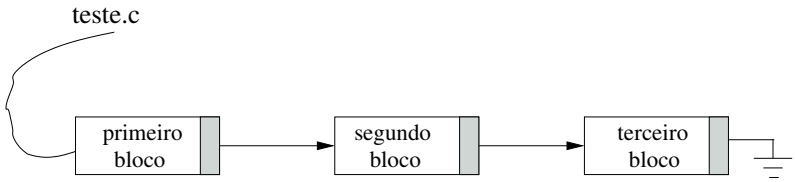


Figura 2.5: Alocação encadeada

lista encadeada que os arquivos tomam, nesta estrutura o acesso seqüencial funciona muito bem, pois nele o arquivo é lido do início até o final e à medida que vai sendo lido, os endereços dos próximos blocos são encontrados.

O problema aparece quando a intenção é fazer acesso relativo. Neste caso, a estrutura em forma de lista encadeada produz um baixo desempenho, pois para acessar uma informação contida num bloco i, será necessário ler todos os $i-1$ anteriores para que se consiga obter os endereços dos blocos até o bloco i.

Alocação Indexada

Uma variação desta estratégia, é reunir todos os ponteiros em um único bloco chamado de bloco de índices *(index block)* mostrado na figura 2.6. Este bloco de índices representa um vetor com "n" entradas, cada qual é um ponteiro para um bloco de dados.

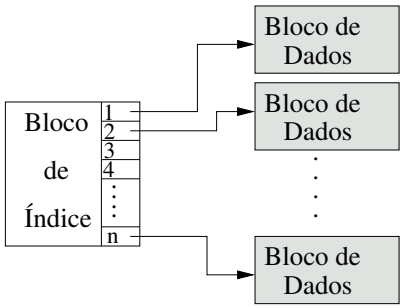


Figura 2.6: Bloco de índice

Inicialmente todos os ponteiros são nulos, mas quando os blocos de dados são alocados, eles são referenciados numa entrada adequada no bloco de índices. Nessa variação da estratégia de alocação encadeada, não é mais preciso percorrer o arquivo para encontrar o bloco desejado. O endereço do bloco pode ser facilmente obtido de uma entrada no bloco de índices.

Uma questão importante na implementação é o tamanho do bloco de índices. Se este for muito pequeno, ele não será capaz de gerenciar arquivos grandes e se for muito grande representará um desperdício para os arquivos pequenos, que mesmo não utilizando muita memória, ocupam um bloco de índices que gasta muito espaço.

O ideal seria que os arquivos tivessem um bloco de índices de um tamanho consistente com a sua necessidade. Uma técnica com essa idéia básica foi utilizada nos sistemas UNIX *(ffs)* e é chamada de inodo (*i-node*) (Tanenbaum, 1992; Silberschatz e Galvin, 1997; Oliveira et al., 2000; Vahalia, 1996).

Um inodo possui os campos mostrados na figura 2.7. Neste sistema, o inodo mantém alguns ponteiros (tipicamente 15) que estão divididos em quatro categorias (ponteiros para blocos diretos, para blocos indiretos simples, duplos e triplos). Geralmente 12 ponteiros são reservados para blocos diretos, isto significa que os ponteiros apontam diretamente para blocos de dados. Considerando que os blocos de dados possuem 4KB, podemos representar arquivos de até 48KB [2] usando os blocos diretos.

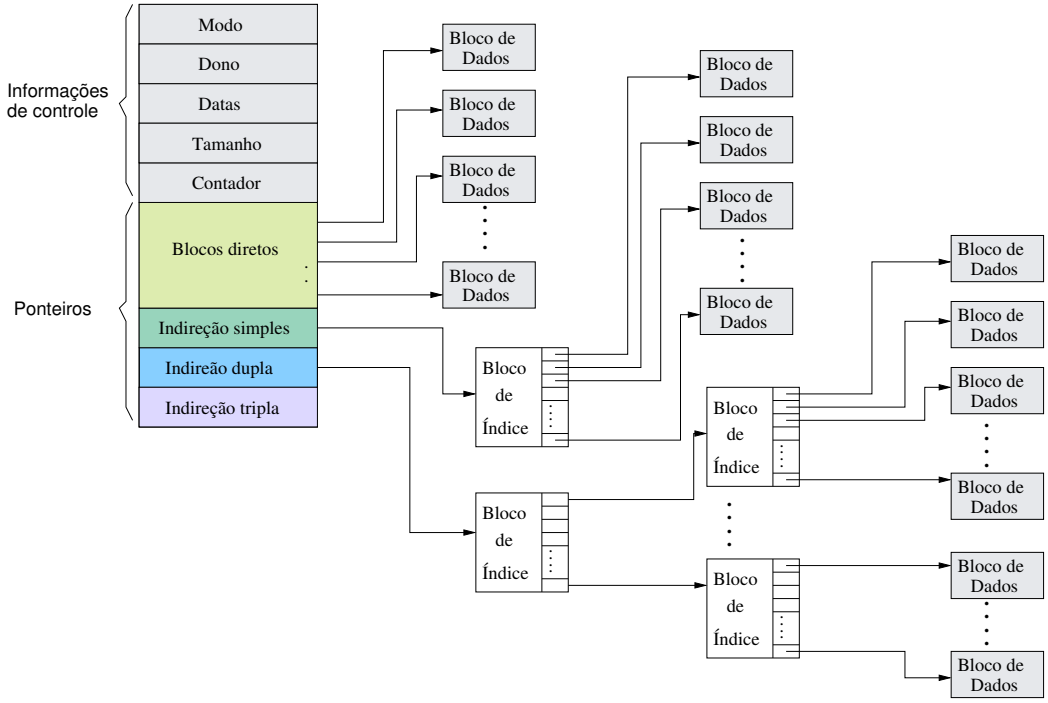


Figura 2.7: Estrutura do Inodo

Esse tamanho máximo não é suficiente para representar arquivos grandes, por esta razão os três outros ponteiros são usados. O ponteiro indireto simples aponta para um bloco de índices e as entradas nesse bloco de índice irão apontar para os blocos de dados. Com blocos de índices de 4KB e usando endereços de 32 bits (4 bytes) pode-se representar arquivos de até 4MB somente usando os ponteiros indiretos simples.

Depois temos a indireção dupla e tripla, onde temos respectivamente dois e três níveis de blocos de índices até chegar aos blocos de dados. Toda essa estrutura permite que o tamanho de um arquivo exceda o limite máximo de representação usando ponteiros de 32 bits (4 Gigabytes).

2.2.2 Gerenciamento de espaço livre

Para controlar os blocos livres, o sistema de arquivos utiliza uma lista de blocos livres. Se um arquivo precisa alocar um bloco, este é retirado da lista de blocos livres e no caso de o arquivo ser destruído, os blocos de dados que o compunham serão inseridos na lista para poderem ser re-utilizados posteriormente (Tanenbaum, 1992; Silberschatz e Galvin, 1997).

[2] Um estudo estatístico em sistemas UNIX mostrou que o tamanho médio de arquivos é de 1KB (Tanenbaum, 1992).

Implementação por mapa de bits

O mapa de bits, é uma forma de implementar uma lista de blocos livres. Os blocos livres são representados por bits 1 e os blocos ocupados são representados por bits em 0. Encontrar um bloco livre resume-se em pesquisar o mapa de bits por um bit em 1. O sistema de arquivos mantém uma área do seu dispositivo de armazenamento para armazenar os bits que descrevem o estado de cada bloco existente no SA. No seguinte exemplo 0000100100000101 temos 4 blocos livres, o primeiro deles está na posição 5, o segundo está na posição 8 e os outros nas posições 13 e 15 respectivamente.

Implementação por lista encadeada

Uma outra técnica é agrupar os blocos livres numa lista encadeada. Cada um dos blocos possui um ponteiro indicando o próximo bloco da lista, o último bloco aponta para nulo indicando que não existem mais blocos. Um outro ponteiro é utilizada para indicar o primeiro elemento da lista.

Quando um bloco precisa ser alocado, basta obter o primeiro bloco da lista e atualizar os ponteiros. Na eliminação de um arquivo, os blocos que pertenciam a este arquivo serão inseridos nesta lista. Tanto a inserção como a retirada de um bloco da lista é uma operação simples e nunca é necessário percorrer a lista encadeada.

Dependendo do dispositivo de armazenamento utilizado, esse método pode se tornar muito custoso. Nos casos de discos magnéticos é muito ineficiente fazer um acesso ao disco para obter cada um dos blocos livres. Uma otimização dessa técnica para discos magnéticos agrupa vários endereços de blocos livres nos blocos da lista.

2.3 Conclusão

Neste capítulo foram apresentados alguns aspectos de um sistema de arquivos na visão tanto de um usuário como na do implementador. Em geral, a forma de se implementar um SA não mudou significativamente nos últimos anos como conseqüência do bom enquadramento destes com os requisitos das aplicações, do sistema operacional e dos usuários, em sistemas de propósito geral.

Essa situação foi mudada com o surgimento de novas classes de aplicações com diferentes requisitos (tais como requisitos de memória para sistemas embutidos). Neste caso, para um maior aproveitamento do sistema de arquivos, este deve ser concebido levando-se em consideração os problemas dessa classe de aplicação ou mesmo de uma aplicação alvo.

Por exemplo, para uma aplicação que faz uso de recursos multimídia, tais como recuperação e armazenamento de vídeo e áudio, obtêm-se um ganho em utilizar um sistema de arquivos voltados para multimídia, pois ele é concebido levando-se em consideração as necessidades e problemas desta área.

Capítulo 3

Aspectos de programação em sistemas embutidos de tempo real

Este capítulo descreve os dispositivos típicos para o qual este trabalho está focado, além de discutir os problemas e soluções conhecidos para a construção de programas de tempo real em sistemas embutidos. Ele foi montado com base em entrevistas, pesquisas, em estudos de sistemas reais e será assumido como premissa para o restante deste trabalho.

3.1 Sistemas embutidos de tempo real

Um sistema embutido *(embedded system)* é uma combinação de *hardware* e *software* projetados para realizar uma tarefa específica (Barr, 1999).

Geralmente, um sistema embutido é um componente contido num sistema maior, por exemplo: Num computador de uso geral, pode-se utilizar *modems*, *scanners* e impressoras. Dentro desses dispositivos, temos sistemas embutidos compostos por alguns microprocessadores e memórias, que controlam o dispositivo para o qual foram projetados e ainda trocam informações com o computador.

O termo embutido *(embedded)* pode gerar alguma confusão, mas considera-se que um sistema embutido é qualquer dispositivo que inclua um computador programável, mas que o propósito deste dispositivo não é ser um computador de propósito geral e sim realizar tarefas específicas. Desta forma, um computador pessoal (*PC*) não é por si só um sistema embutido (pois é capaz de realizar uma grande variedade de tarefas), mesmo que em muitos casos o *hardware* utilizado em *PCs* seja freqüentemente utilizado para construir sistemas embutidos (Wolf, 2001).

Um sistema embutido, pode ser visto como um diagrama em camadas, como na figura 3.1.

Uma grande parcela dos sistemas embutidos também podem ser classificados como sistemas de tempo real. Um sistema de tempo real é um sistema no qual se espera que, além de fornecer uma resposta correta para uma dada tarefa, também dê garantias quanto ao tempo de conclusão desta.

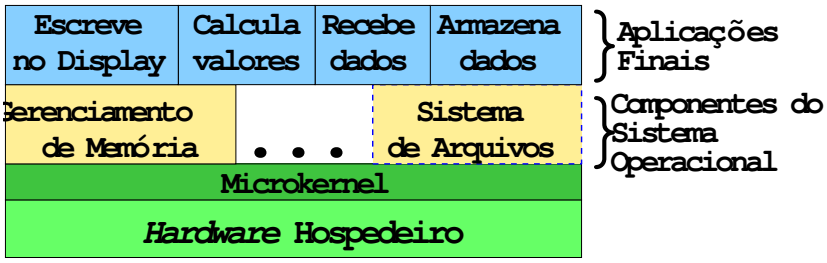


Figura 3.1: Sistema em camadas

Nestes sistemas as operações possuem *deadlines* para sua conclusão, isto é, se espera que estas tarefas sejam concluídas dentro de um tempo limite. Na maioria dos casos, um *deadline* perdido é tão ruim quanto uma resposta incorreta (Farines et al., 2000).

As aplicações com requisitos de tempo real variam muito em relação à complexidade e às necessidades de garantia no atendimento das restrições temporais. De um lado do espectro, temos aplicações simples que estão embutidas em televisões, videocassetes, fornos de micro-ondas, rádio-relógios, cafeteiras, etc. Na outra extremidade do espectro temos sistemas de controle de tráfego aéreo, sistemas de controle de plantas industriais, sistemas militares, etc.

Dentre as aplicações de tempo real, podemos diferenciar aplicações que não possuem restrições críticas[1], tais como videogames e sistemas de áudio, nas quais o não cumprimento de suas restrições temporais não provocarão grandes problemas além da redução da qualidade do resultado final. Por outro lado, existem aplicações com restrições críticas tais como sistemas de monitoramento de pacientes em hospitais, sistema de controle de veículos, controle de robôs, etc. Nestas, o não cumprimento de suas restrições temporais poderão causar problemas graves, perda de vidas ou prejuízo financeiro.

Entre as várias formas de combinar *hardware* e *software* para construir a solução específica de controle, a divisão em duas categorias ou contextos é a que mais facilmente fornece uma visão do atual estado da arte desses sistemas.

3.1.1 Projeto de *hardware* personalizado

Composto basicamente por um microprocessador, uma memória não volátil para armazenamento dos programas *(ROM / FlashROM / EEPROM)*, uma memória volátil *(RAM)* e um relógio *(clock)*.

Neste contexto um projetista de *hardware* utiliza estes elementos básicos e adiciona outros tais como conversores *AD / DA*, multiplexadores, sensores e atuadores para criar uma solução de controle para um projeto específico. Por exemplo, um telefone celular de nova geração, um equipamento médico, um controle de servo-motores, um *WEB computer*, um controle de injetora plástica, controle de freio de carro, uma máquina de fax, um modem, um alarme, uma central telefônica, etc.

O microprocessador executa um microkernel de tempo real que provê certas funcionalidades mínimas, tais como gerenciamento de processos, escalonamento e mecanismos de sincronização. Outras

[1] A criticalidade é apenas uma das classificações existentes.

funcionalidades podem estar também presentes no microkernel, mas sempre tendo em mente que o dispositivo possui recursos limitados como memória, que impedem a construção de soluções genéricas. Dessa forma, o programador se vê forçado a inserir no dispositivo, somente o código necessário a execução da aplicação, isto é, o *footprint*[2] da aplicação deve ser minimizado.

Nestes sistemas, o armazenamento de informações é feito em memórias de estado sólido (*RAM* alimentada a bateria ou um *FlashROM*) e dificilmente em discos magnéticos. Os principais motivos para isso são os altos tempos envolvidos no acesso a disco, que as vezes se tornam proibitivos, uma maior complexidade no *software*/*hardware*, além da necessidade de certas condições para seu correto funcionamento, tais como ausência de campos magnéticos e baixa trepidação, que são difíceis de serem alcançados num ambiente industrial (onde está presente grande parte da eletrônica embutida).

Esse contexto alvo apresenta diversos pontos positivos, tais como baixo custo final da solução de controle, alto desempenho e adequação completa ao problema. Recentemente esse fato tem levado a um grande aumento na utilização de microprocessadores embutidos, principalmente de processadores para fins específicos tais como *DSPs - Digital Signal Processors*. Esses processadores de fins específicos são projetados para processar sinais digitais numa freqüência muito alta e realizar filtragens de sinais. Comparativamente, microprocessadores de propósito geral não são capazes de realizar essas tarefas, pelo menos não na mesma freqüência (Sargent III e Shoemaker, 1995).

Para facilitar o desenvolvimento desses sistemas embutidos, foram desenvolvidas ferramentas para programação, depuração, simulação e ferramentas de síntese de controle. Segundo (Stankovic, 2001), em 1998, 98% dos microprocessadores produzidos foram utilizado em *hardware* embutido, apenas 2% foram utilizados em computadores de propósito geral.

3.1.2 Microcomputadores de propósito geral

Outro contexto alvo é o da utilização de microcomputadores de propósito geral, tais como os encontrados atualmente no mercado (como *PCs*). Os que seguem este contexto alvo argumentam que dada a variedade e o custo dos componentes de computadores fica difícil justificar o custo e o tempo despendido no projeto a partir do zero de uma solução de controle similar para ser utilizada por exemplo no controle de uma máquina, ou de um parque de máquinas (Lages e Hemerly, 1999). Também afirmam que dada a alta disponibilidade de *hardware* padrão poderia-se facilmente substituir um componente defeituoso por outro, sem o alto custo e o tempo de seu desenvolvimento. Na figura 3.2 temos um exemplo de uso, um computador é utilizado para controlar a posição de uma antena de satélite, capturar e processar o sinal recebido. Os dados tais como registros de histórico, dados recebidos e registros de órbitas ficam armazenados no computador.

[2] *Footprint* - quantidade de memória utilizada por uma aplicação.

Figura 3.2: Computador controlando dispositivo

Fica claro que o sistema operacional que está inserido neste caso deve ser um sistema operacional de tempo real, com as mesmas funcionalidades mínimas de um microkernel embutido, sobre o qual será executada a aplicação. A existência de outros recursos, tais como facilidades de comunicação em rede, grande quantidade de memória principal, ferramentas de programação, dispositivo de entrada de comandos (teclado), dispositivo de saída de informações/status (monitor de vídeo) e bibliotecas de rotinas padronizadas, representam outras facilidades.

Neste contexto, o armazenamento de informações é geralmente feito em dispositivos magnéticos, pois nesse caso, toda a interface *hardware*/*software* já existe e esses dispositivos provêem uma grande quantidade de dados a um custo muito baixo. Essa forma de armazenamento só não se torna atraente quando realmente não se pode arcar com os tempos de acesso, caso em que um sistema de arquivos em memória principal deve ser utilizado. Sendo assim, por ser menos restrito em termos de recursos, podem-se utilizar as mesmas soluções de *software* usadas no contexto de *hardware* personalizado.

Como exemplos clássicos de sistemas operacionais de tempo real existentes para microcomputadores de uso geral, temos o RT-Linux [3] e RTAI [4]. Embora nenhum desses utilize sistemas de arquivos concebidos para tempo real, pode-se construir um sistema de arquivos em memória principal que trate dos aspectos de tempo real, da mesma forma que seria feito num contexto de *hardware* personalizado.

3.1.3 Técnicas para lidar com o problema

Independente do contexto alvo utilizado, os dispositivos microprocessados já fazem parte de nossas vidas; eles estão na manufatura, automação, comunicação, medicina, em nossos carros, escritórios e a cada dia conquistam um novo mercado.

No passado os dispositivos eram mais simples, compostos por somente alguns circuitos integrados (CIs) com um pequeno poder computacional e com poucas centenas de linhas de código em linguagem de baixo nível *(assembly)*. Atualmente, esse quadro não mais se aplica, hoje os dispositivos possuem muitos microprocessadores e executam dezenas de milhares de linhas de código em linguagem de alto nível. Fica claro que para lidar com essa complexidade, precisamos utilizar técnicas que facilitem a criação desses dispositivos.

[3] RT-Linux é produzido pela FSMLabs http://www.fsmlabs.com .
[4] RTAI é mantido pelo DIAPM http://www.aero.polimi.it/projects/rtai/.

Dentre as técnicas que auxiliam no projeto de *hardware* estão: Uso de placas programáveis conectadas ao computador para construção do protótipo e uso de simuladores/emuladores. Já em relação às técnicas para construção de *software*, temos: Utilização de camadas de abstração de *software* (utilização de um SO), prover o sistema operacional ou o microkernel com a abstração de um sistema de arquivos de tempo real e produzir/utilizar outras ferramentas automatizadas que auxiliem na elaboração do *software* para aplicações embutidas com requisitos de tempo real.

Um bom exemplo disso é a *Run-time support library (RTS)*[5](Huber, 2002) uma biblioteca C/C++ mantida pela Texas Instruments que provê funções de entrada/saída (tais como `fopen()`, `fread()`, etc.) para um sistema de arquivos que pode ser implementado em *RAM* ou mesmo em um dispositivo qualquer. O usuário pode escrever seu *device driver* para controlar um dispositivo e instalá-lo usando chamadas dessa biblioteca.

3.2 Requisitos das aplicações embutidas de tempo real

Todos os sistemas operacionais desenvolvidos ou adaptados para tempo real mostram grande preocupação com o escalonamento de tarefas visando atender os requisitos temporais da aplicação. Mas um bom algoritmo de escalonamento não será suficiente se as demais partes do sistema não forem cuidadosamente planejadas e codificadas. É necessário um levantamento de pior caso dos algoritmos envolvidos e também tratar problemas decorrentes da concorrência interna, tais como bloqueios e inversões de prioridade.

3.2.1 Previsibilidade

O aspecto mais importante que um *software* (por exemplo, um sistema de arquivos) voltado para aplicações de tempo real deve ter é um tempo de resposta máximo conhecido para todas as suas funções. Esse tempo máximo poderia ganhar a forma da equação 3.1 ou simplesmente ser um valor máximo estimado.

$$t_{read} = max(t_{bloqueio}) + max(t_{comp}) + \ldots \tag{3.1}$$

onde:

t_{read}	tempo máximo de uma chamada a *read*;
$max(t_{bloqueio})$	tempo máximo em que uma chamada pode ser bloqueada por outras rotinas;
$max(t_{comp})$	tempo máximo que o código da rotina *read* leva para executar.

Um fator que tem grande influência sobre a previsibilidade é o tempo de computação de um dado algoritmo. Não seria interessante utilizar um algoritmo eficiente na maioria dos casos mas que é muito

[5]Mais informações podem ser obtidas em: `http://www.ti.com`.

ruim em casos críticos. Assim, é necessário um estudo minucioso da complexidade dos algoritmos utilizados para estabelecer uma comparação e posterior escolha de um algoritmo com um bom tempo de computação no pior caso. Uma das notações mais utilizadas para complexidade de algoritmos usam os símbolos Ω para complexidade média, O para complexidade de pior caso e Θ para o melhor caso. Como em Sistemas de Tempo Real a grande preocupação é com o pior caso, nosso interesse será pela notação O.

Complexidades de pior caso geralmente encontradas são $O(1)$, onde o tempo de execução do algoritmo é sempre constante. $O(n)$ onde o tempo de execução do algoritmo é proporcional a "n" onde este "n" é o número de elementos utilizados como entrada para o algoritmo (assim se o algoritmo tem a função de ordenar 100 números o valor de "n" é 100). Na complexidade $O(log(n))$ o tempo de execução do algoritmo é proporcional ao logaritmo de base 2 de "n". As complexidades $O(n^2)$, $O(n^3)$, etc. são ditas polinomiais, pois são descritas por um polinômio de base "n" e a complexidade $O(2^n)$, $O(3^n)$, etc são ditas exponenciais pois são descritas por uma equação exponencial (Toscani e Veloso, 2001).

A figura 3.3 mostra o crescimento do tempo de computação em relação a entrada dos algoritmos ("n") para algumas complexidades de algoritmos. O grande mérito desta figura é dar a idéia do crescimento das funções linear, logarítmica e exponencial. Deste ponto em diante, podemos comparar complexidades de algoritmos e saber quais terão um desempenho mais eficiente.

Deve-se compreender que o comportamento de um algoritmo somente se evidencia para valores grandes de entrada e independe da plataforma sobre o qual será implementado.

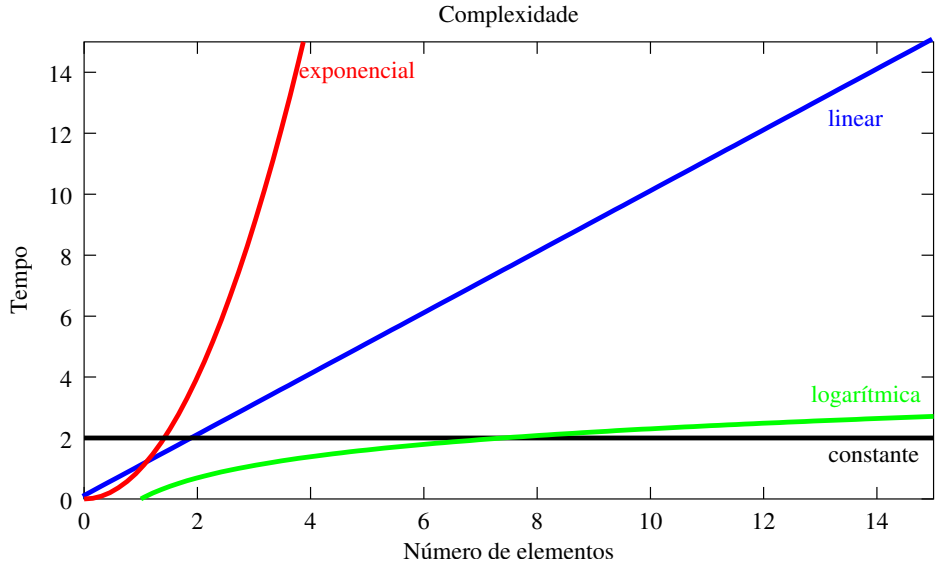


Figura 3.3: Complexidade de alguns algoritmos

Uma boa maneira de compreender a importância do estudo da complexidade dos algoritmos utilizados é imaginando o problema de pesquisar, inserir e eliminar valores. Na literatura de ciência da computação podemos encontrar diversos algoritmos (mostrados na tabela 3.1) que solucionam este problema, cada qual com características distintas na forma de armazenar, pesquisar, inserir e eliminar os elementos. No caso de tempo real, o mais interessante seria um algoritmo que implementasse

essas operações num tempo hábil e que não apresentasse muita variação entre a pesquisa, inserção e eliminação.

Assim, após uma comparação da complexidade dos algoritmos e características de implementação verifica-se que as árvores *Red-Black* (Cormen, 1989) e as árvores *AVL* (Kruse, 1984), se destacam por possuírem uma mesma complexidade para os três tipos de operação e por seus algoritmos revelarem-se muito melhores do que os demais.

3.2.2 Concorrência

Os sistemas multitarefa tornaram-se populares nos anos 70 e são usados até hoje pela sua capacidade de aproveitar melhor os recursos de *hardware* dando a ilusão de execução simultânea de várias tarefas (*threads*) num mesmo processador. No centro disso está a figura do escalonador *(scheduler)* que faz o chaveamento entre tarefas e assim escolhe qual tarefa deve ganhar o direito ao uso do processador. Assim como em qualquer programa, em um sistema de arquivos que suporte multitarefa, poderão existir várias linhas de execução *(threads)* executando "simultaneamente" um mesmo trecho de código.

Neste caso, alguns problemas (muitas vezes difíceis de identificar e reproduzir) ocorrerão. Suponhamos que a função mostrada no código 3 seja executada por duas *threads* e faça uso da estrutura compartilhada mostrada na figura 3.4 (nesta figura não estão representados alguns campos da estrutura) que armazena algumas informações de arquivos, tais como nome, tamanho, posição atual, data e *status*. Essas informações estão dispostas numa tabela de tamanho MAX. Além dessas, existe uma variável compartilhada que diz quantos arquivos estão sendo utilizados.

```
int cria_arquivo (char *nome)
{
     for (x = 0; x < MAX; x++)
     {
          if (Arquivos[x].livre) {
               strcpy(Arquivos[x].nome,nome);
               Arquivos[x].tamanho = 0;
               Arquivos[x].posicao = 0;
               Arquivos[x].status  = DEFAULT;
               Arquivos[x].data    = getTime();
               Arquivos[x].livre   = OCUPADO;
               numero_ocupados++;
               return (x);
          }
     }
     return (-1);
}
```

Código 3: Função que cria arquivo

Se a *thread* A executar integralmente, ela irá percorrer a tabela até a posição 3, preencher os campos com os dados no novo arquivo, incrementar a variável "número_ocupados" para 7 e retornar

Método	*Inserção*	*Pesquisa*	*Eliminação*
Seqüencial	Rápida, complexidade $O(1)$	Lenta $O(n)$	Lenta $O(n)$
Binária	Lenta (até $n-1$ deslocamentos)	Rápida $O(log_2(n))$	Lenta (até $n-1$ deslocamentos
Árvore Binária	Lenta $O(n)$ (decaiu para uma lista seqüencial)	Lenta $O(n)$	Lenta $O(n)$
Árvores *AVL*	Rápida $O(log_2(n))$ mas algumas trocas	Rápida $O(log_2(n))$	Rápida $O(log_2(n))$ (custo na retirada)
Interpolada	Lenta	Lenta $O(n)$ no pior caso mas $O(log_2(log_2 n))$ no caso médio	Lenta
Árvores B	Rápida, mas com custo na inserção e desperdício de memória no armazenamento	Muito rápida $O(log_a n)$	Média (com muitas trocas)
Tabela *Hash*	Lenta $O(n)$ mas eficiente no caso médio e necessidade de tratar colisões	Lenta $O(n)$	Lenta $O(n)$
Árvores *Red-Black*	Rápida $O(log_2(n))$ com poucas trocas	Rápida $O(log_2(n))$	Rápida $O(log_2(n))$ com poucas trocas

Tabela 3.1: Comparação de alguns algoritmos (Horowitz e Sahni, 1984; Kruse, 1984; Santos e Azevedo, 2001; Cormen, 1989)

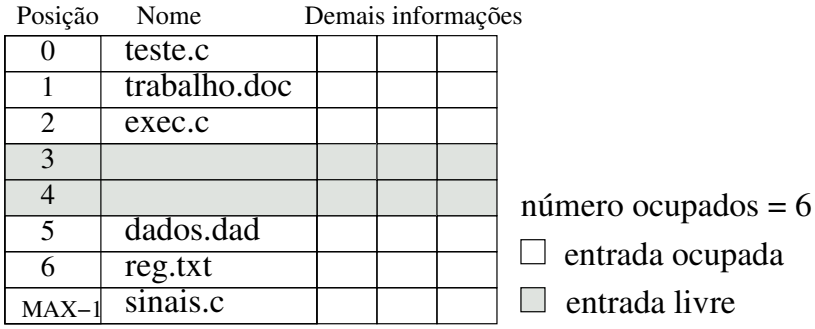


Figura 3.4: Lista de arquivos

o valor da posição para o seu chamador. Logo após, a *thread* B é posta para rodar pelo escalonador e ela encontra uma posição livre, preenche com os valores do novo arquivo, incrementa o número de arquivos no sistema para 8, e retorna o valor para o seu chamador.

No caso contrário (em que B inicia primeiro) tudo também funciona corretamente. O problema ocorre quando a primeira *thread* (digamos que seja A) está executando a função, encontrou uma posição disponível e antes marcar esta como ocupada, o escalonador decide que está na hora de a outra *thread* rodar (digamos B). Neste caso, a *thread* B encontrará a mesma posição livre que a da *thread* A e assim criará uma situação de erro, pois as duas *threads* irão retornar o mesmo valor para o seu chamador. Esta situação de erro que existe em decorrência da ordem da execução das *threads* é chamada de *condições de disputa* e deve ser eliminada a todo o custo usando o conceito de *seções críticas* (Tanenbaum, 1992).

Seções críticas são áreas do código que deveriam ficar protegidas de acesso concorrente[6], assim, somente uma das *threads* pode estar ativa dentro de uma seção crítica, as demais ficarão bloqueadas, incapazes de entrar na seção crítica. Uma das formas de evitar condições de disputa em seções críticas é através de um mecanismo de sincronização conhecido como semáforo (Oliveira et al., 2000).

O semáforo é um tipo abstrato de dado, composto por um número inteiro e uma lista de *threads*. Somente duas operações são permitidas `P()` - que decrementa o valor do semáforo e bloqueia a *thread* quando este valor se torna negativo e `V()` - que incrementa o valor numérico, caso exista alguma *thread* bloqueada na lista deste semáforo, então a primeira da fila será liberada.

Deve ser observado que as operações `P()` ou `V()` não podem ser interrompidas durante a sua execução e outra operação sobre o mesmo semáforo iniciada. Elas são ditas **atômicas**. Uma simplificação desta idéia são os semáforos binários ou *mutex* que só possuem dois valores, ou ele tem valor 1 e está liberado ou ele possui valor 0 e está bloqueado. A função de criação de arquivos usando semáforos binários está exemplificada no código 4.

O trecho do código 4 agora está livre de condições de disputa, mas examinando mais detalhadamente esta função, verifica-se que a seção crítica está protegendo todo o código da função.

Mesmo correta, esta versão da função sacrifica em muito a concorrência, pois agora uma *thread* ficará aguardando bloqueada enquanto outra *thread* executa completamente a função. Isso se torna

[6]Do inglês *concurrent* - simultâneo.

```
int cria_arquivo (char *nome)
{
     int tmp = -1;
     P(semaforo);   /* Entra na secao critica */
     for (x = 0; x < MAX; x++)
     {
          if (Arquivos[x].livre) {
               strcpy(Arquivos[x].nome,nome);
               Arquivos[x].tamanho = 0;
               Arquivos[x].posicao = 0;
               Arquivos[x].status  = DEFAULT;
               Arquivos[x].data    = getTime();
               Arquivos[x].livre   = OCUPADO;
               numero_ocupados++;
               tmp = x;
               break ;
          }
     }
     V(semaforo);   /* Sai da secao critica    */
     return (tmp);
}
```

Código 4: Função que cria arquivo

ainda mais grave, pois em geral o escalonador implementa uma política de prioridade, isto é, ele atribui uma ordem na execução das *threads*. Assim, temos *threads* de prioridade 1 (no nosso exemplo a mais prioritária), *threads* de prioridade 2, prioridade 3, etc.

Quando uma *thread* de prioridade 2 ou 3 está rodando, o escalonador pode interromper sua execução para em seu lugar executar uma *thread* de prioridade mais alta. Considera-se que a *thread* que foi suspensa, recebeu interferência da *thread* de prioridade mais alta. Mas se uma *thread* de prioridade mais baixa estiver dentro de uma seção crítica, mesmo que o escalonador coloque uma *thread* de prioridade mais alta para rodar, esta não poderá executar a seção crítica, pois esta encontra-se bloqueada pela *thread* (de menor prioridade) que estava executando. Essa situação de bloqueio de uma *thread* de mais alta prioridade por uma *thread* de mais baixa prioridade é chamada de inversão de prioridade e deve ser evitada.

Evidentemente, não existe uma forma de evitar complemente esta inversão de prioridade, mas ao codificar uma função concorrente, devemos ter o cuidado de encontrar as seções que realmente precisam ser protegidas. Uma terceira versão do código da função de criação de arquivos está no Código 5.

Nesta versão, existem mais pontos de concorrência, isto é, as seções críticas ficaram mais delimitadas permitindo que mesmo que uma seção crítica esteja em uso, outra *thread* poderá estar executando outra parte do código.

Assim como políticas de alocação de recursos *priority ceiling* (Farines et al., 2000) procuram minimizar o problema da inversão de prioridade, uma granularidade menor para as seções críticas

```
int cria_arquivo (char *nome)
{
     for (x = 0; x < MAX; x++)
     {
          P(semaforo_1);        /* Entra na secao critica 1 */
          if (Arquivos[x].livre)
          {
               Arquivos[x].livre = OCUPADO;
               V(semaforo_1);   /* Sai da secao critica   1 */
               strcpy(Arquivos[x].nome,nome);
               Arquivos[x].tamanho = 0;
               Arquivos[x].posicao = 0;
               Arquivos[x].status  = DEFAULT;
               Arquivos[x].data    = getTime();
               P(semaforo_2);   /* Entra na secao critica 2 */
               numero_ocupados++;
               V(semaforo_2);   /* Sai da secao critica   2 */
               return (x);
          }
          V(semaforo_1);        /* Sai da secao critica   1 */
     }
     return ( -1 );
}
```

Código 5: Função que cria arquivo

resulta no mesmo objetivo. Ambas abordagens reduzem o tempo máximo de resposta das tarefas em questão.

3.2.3 Memória

Muitas soluções de *hardware* para sistemas embutidos possuem pouca memória, com o propósito de reduzir o custo do equipamento. Indiretamente, esse fato influencia a construção do software utilizado no dispositivo embutido.

Nitidamente soluções genéricas, muito utilizadas em computadores de propósito geral, não se aplicam. Por suas limitações de *hardware*, o software inserido nos dispositivos embutidos precisa apresentar um pequeno *footprint*.

Para o caso do sistema de arquivos proposto neste trabalho, além dos dois ítens necessários (previsibilidade e concorrência), foram adicionadas características importantes com o objetivo de torná-lo mais útil e bem aceito pela indústria de *software*. Com o intuito de criar um sistema de arquivos compacto (prevendo o seu uso em sistemas embutidos com severas restrições de memória) foi utilizado o conceito de composição. Com a composição, é possível construir um sistema de arquivos somente com os componentes que seriam utilizados pela aplicação final, dessa forma somente o código necessário é utilizado, poupando memória.

Num caso em que a aplicação final precisar usar no máximo “n” arquivos, o SA suportará somente “n”. Se o tamanho máximo destes é “x”, então o SA será otimizado para manipular arquivos de no máximo este tamanho. Se a aplicação faz uso somente de rotinas de `read()` e `write()`, então somente essas serão suportadas.

Assim, diferentemente de um sistema de arquivos convencional, onde mesmo que a aplicação final utilize uma pequena parcela dos recursos do SA, este estará presente em sua totalidade, ocupando espaço em memória inutilmente, na composição, podemos escolher quais requisitos o SA deve atender e somente gerar o código necessário para atendê-los. O sistema de arquivos neste caso é dito **personalizado**.

Na figura 3.5 temos ilustrado um diagrama da memória com a divisão explícita entre o código do SA e o código da aplicação. No ítem **a)** temos uma grande área de código para o SA, grande parte dela não será utilizada pela aplicação, mas permanecerá ocupando espaço. No ítem **b)** a área de código do SA é reduzida, pois este está somente fornecendo os recursos que a aplicação utiliza, reduzindo assim o *overhead* de memória.

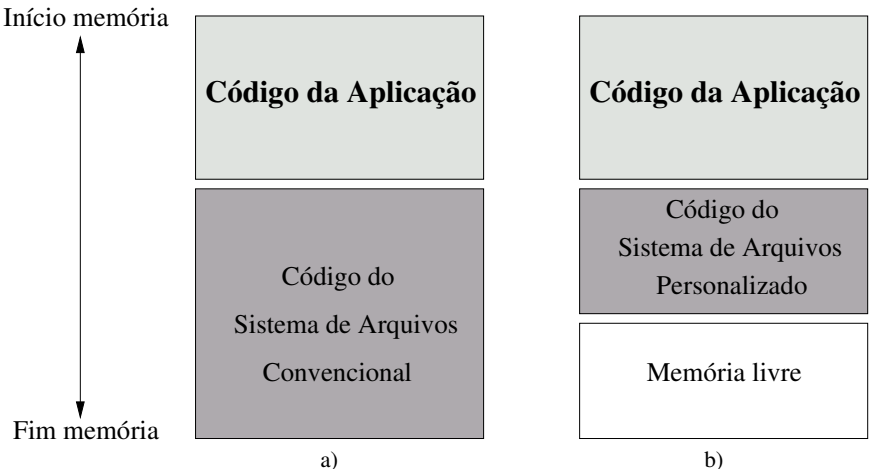


Figura 3.5: Diagramas de alocação de memória

3.3 Conclusão

Neste capítulo foi dada uma visão de sistemas embutidos de tempo real com ênfase nos aspectos e problemas que devem ser levados em consideração na construção de *software* para esses sistemas, tais como concorrência interna, seções críticas, granularidade das seções críticas, análise de algoritmos, utilização de memória, etc.

Capítulo 4

Sistema de arquivos desenvolvido

Conforme descrito no capítulo 2, obtém-se um maior aproveitamento do Sistema de Arquivos quando este está plenamente adequado a uma aplicação alvo.

O mesmo raciocínio pode ser aplicado para a área de sistemas embutidos de tempo real, onde existem muitos problemas que não são contemplados por um sistemas de arquivos convencional e precisam ser tratados por um sistema de arquivos de tempo real.

Neste capítulo será apresentado um módulo chamado sistema de arquivos abstrato *(SAA)* para produzir sistemas de arquivos voltados para aplicações de tempo real e com algumas características que o tornam altamente personalizável e propício a ser utilizado em aplicações embutidas *(embedded systems)*.

É suposto que o sistema de arquivos não será implementado através de um disco magnético, mas sim através de memória *RAM* mantida com uso de baterias, ou memória *Flash*. Essas são as soluções mais freqüentes em equipamentos de baixo custo.

4.1 Visão geral

O sistema de arquivos desenvolvido neste trabalho foi projetado para tratar os problemas apresentados no capítulo 3. Para tanto, em sua implementação foram utilizados algoritmos com tempos de pior caso explícitos e convenientes (para tratar o aspecto da previsibilidade). Já as implementações das funções do sistema de arquivos foram feitas de modo a minimizar as inversões de prioridade e maximizar a concorrência interna[1].

O aspecto de economia de memória é obtido com a composição de um sistema de arquivos utilizando somente os componentes necessários e será mostrado em mais detalhes no decorrer deste capítulo.

[1] No apêndice B temos diagramas que mostram as funções disponíveis para o programador, e as chamadas que estas fazem para funções internas do sistema de arquivos, com a lista dos semáforos que estas utilizam.

Esquematicamente o sistema de arquivos pode ser vislumbrado como um sistema em camadas, como na figura 4.1. Nesta figura, temos na camada **1** a parte de nível mais alto (com a qual o programador tem contato) e nos níveis mais baixos (camadas **2**, **3** e **4**[2]) temos as funções que dão suporte a funcionalidade necessária.

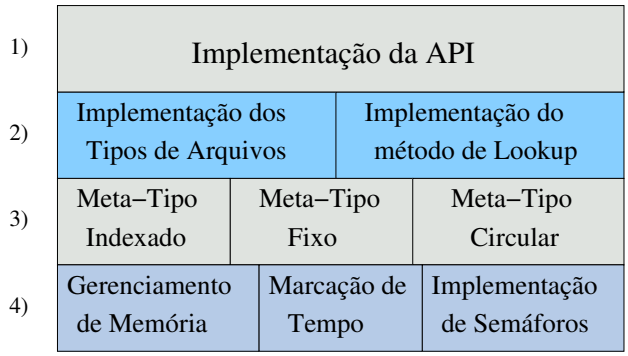


Figura 4.1: Sistema de Arquivos visto em camadas

Esse módulo (chamado de Sistema de Arquivos Abstrato SAA) possui uma grande quantidade de código com a implementação de vários componentes internos. Dependendo das necessidades do sistema de arquivo a ser gerado, partes desse código serão utilizadas e outras não. Além de selecionar essas partes (componentes) do SAA, o programador do SA pode também escolher entre diferentes implementações destas.

Essa liberdade em selecionar e juntar componentes do SAA, personalizar ao máximo e ainda assim manter as características de tempo real e baixa utilização de memória representam um grande salto de produtividade na construção de aplicações embutidas.

4.2 Projeto do sistema de arquivos

O projeto de um sistema de arquivos pode ser visto como um diagrama esquemático de seus componentes fundamentais. Na figura 4.2 temos o diagrama do sistema de arquivos proposto neste trabalho e abaixo, uma explicação detalhada dos seus componentes.

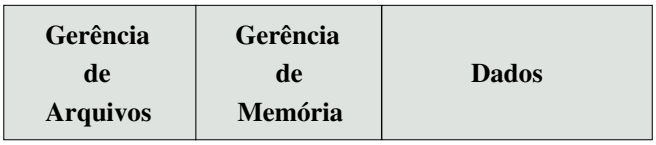


Figura 4.2: Diagrama esquemático do sistema de arquivos proposto

[2]Alguns componentes, como por exemplo “Marcação de tempo” e “Implementação de semáforos” não são realmente implementados dentro do SAA, eles implementam somente interfaces para as funções respectivamente de tempo e de manipulação de semáforos existente no SO/microkernel.

4.2.1 Gerência de arquivos

Responsável pela manipulação e armazenamento de arquivos, esta parte pode ser decomposta em duas: Diretório e descritor de arquivos.

Responsável pela manipulação e armazenamento de arquivos, esta parte pode ser decomposta em duas, gerenciamento de arquivos e estruturas para representar arquivos.

Diretório é a estrutura responsável por manter as entradas de arquivos. Nesta estrutura são inseridos, pesquisados e eliminados arquivos com base em seu nome. Diferentemente de um sistema de propósito geral, num sistema embutido não existe a necessidade de um sistema de diretórios complexo, pois em geral existirão poucos arquivos armazenados. Em virtude disto, foi implementado um diretório linear. O sistema de arquivos abstrato provê dois componentes para esta função. O objetivo de possuir dois componentes com a mesma função é que estes possuem implementações que lidam com requisitos diferentes. Podemos ter uma funcionalidade implementada com um algoritmo eficiente mas muito complexo e com grande utilização de memória, ou outro algoritmo simples, menos eficiente mas que não utiliza muitos recursos de memória. Em situações onde o número máximo de arquivos é grande, é mais interessante utilizar um algoritmo eficiente, mesmo incorrendo num maior uso de memória. Já nos demais casos, o algoritmo mais simples pode ser utilizado.

O componente mais simples é implementado com uma lista seqüencial (Santos e Azevedo, 2001), (Horowitz e Sahni, 1984), nela os nomes dos arquivos estão dispostos numa tabela como na figura 4.3. Para encontrar um arquivo, deve-se percorrer a tabela até encontrar o elemento em questão ou até o final da mesma. A operação de criação simplesmente percorre a tabela procurando por uma posição marcada como disponível, após encontrá-la esta é marcada como ocupada e utilizada. Para a eliminação basta encontrar o nome do arquivo na tabela e marcar esta posição como disponível. Verifica-se que as operações (inserção, pesquisa e eliminação) tem complexidade $O(n)$, onde "n" é o número máximo de arquivos.

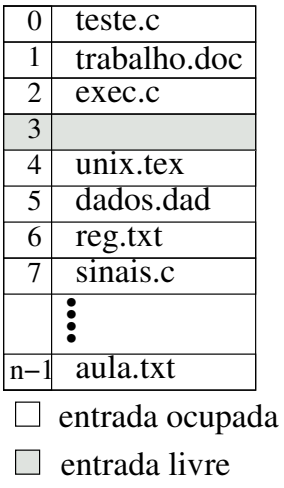


Figura 4.3: Nomes de arquivos dispostos seqüencialmente

O segundo algoritmo utiliza uma árvore vermelha-preta (Cormen, 1989) (*red-black tree*, também conhecida como *Symmetric binary B Trees*) para armazenar os nomes dos arquivos, como na figura 4.4. A escolha deste algoritmo foi motivada pela sua complexidade computacional e características de implementação (simplicidade em relação as árvores *AVL* e economia de memória em relação as árvores B e B+, (Santos e Azevedo, 2001), (Kruse, 1984)).

Este algoritmo desenvolvido em 1972 por Rudolf Bayer é uma árvore binária de pesquisa que é mantida balanceada dinamicamente e cuja complexidade de pior caso para as operações de inserção, pesquisa e eliminação é $O(log(n))$. Esta árvore possui um atributo de cor que é adicionado a cada nodo (podendo seu valor ser vermelho ou preto, daí o nome *red-black*). O atributo de cor é utilizado para especificar o critério segundo o qual a árvore é considerada balanceada e pela definição temos:

I. Todo o nodo possui um valor;

II. O valor de um nodo é maior que o de seu filho da esquerda e menor que o de seu filho da direita;

III. A raiz é preta;

IV. Todo o nodo da árvore é vermelho ou preto;

V. Todo o nodo folha é preto;

VI. Todo o nodo vermelho tem dois nós filhos pretos;

VII. Qualquer caminho simples de um nodo até uma folha descendente tem o mesmo número de nodos pretos.

A árvore é construída com base nesta definição, sempre que um elemento novo não está de acordo com as regras de balanceamento, a árvore deve ser re-estruturada por uma rotação de alguns nodos. O processo de balanceamento é uma operação local e possui uma complexidade $O(1)$.

Diferentemente de uma árvore totalmente balanceada, nas árvores *red-black* o critério de balanceamento é relaxado, isto é, não é necessário re-estruturar a árvore a cada alteração desta.

Descritor de arquivo é a estrutura responsável por descrever o arquivo, isto é, fazer referência ao nome, tamanho, posição atual e dados. Como o objetivo do sistema de arquivos abstrato (módulo a partir do qual será criado o SA) é prover duas formas de implementação de arquivos (arquivos fixos e indexados), escolheu-se uma única estrutura de dados que se adaptasse aos dois casos. O fragmento de código 6, mostra esta estrutura.

Para o caso de arquivos indexados, temos uma estrutura baseada nos *i-nodes* do *FFS* (Vahalia, 1996), com a qual temos blocos diretos, indiretos simples e duplos. Para arquivos fixos, temos apenas um ponteiro.

Diferentemente dos *i-nodes* do *FFS*, neste sistema de arquivos o tamanho desta estrutura não é fixo, ele irá adaptar-se ao tamanho máximo do arquivo suportado. Se o sistema de arquivos for gerado considerando arquivos pequenos, digamos 4KB, não serão geradas as estruturas de indireção, apenas a estrutura de blocos diretos pois ela é suficiente para representar arquivos

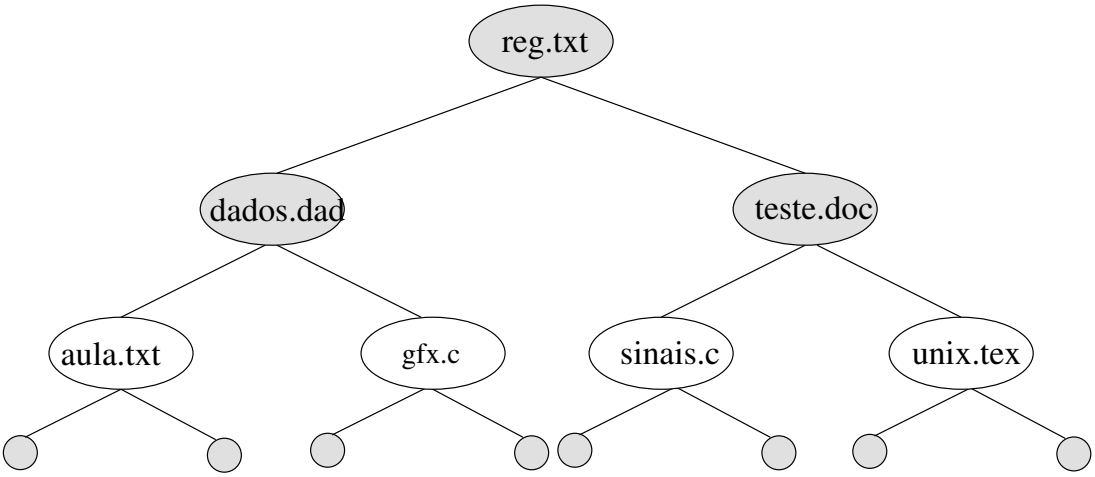


Figura 4.4: Nomes de arquivos dispostos em árvore

deste tamanho. Para arquivos pouco maiores, serão utilizados blocos diretos e indireção simples e somente com arquivos ainda maiores serão utilizadas todas as estruturas.

O número de blocos diretos também é ajustado para um tamanho específico para cada sistema de arquivos, gerando um código otimizado para cada caso, diminuindo-se assim o gasto de memória.

4.2.2 Gerência de memória

É a parte responsável pela alocação/desalocação de porções de memória que serão utilizadas por outros componentes do sistema. Na literatura de ciência da computação encontram-se alguns algoritmos clássicos para lidar com este problema, tais como listas ligadas (*first-fit*, *best-fit* e *worst-fit*) (Tanenbaum, 1992), listas de lacunas por tamanho (Johnstone e Wilson, 1999), buddy system (Tanenbaum, 1992), etc.

Em alguns sistemas de gerenciamento de memória, são utilizados uma combinação de algoritmos para resolver o problema. Um bom exemplo está no gerenciamento de memória utilizado para implementar o *malloc()*, *realloc()* e *free()* da biblioteca padrão do C. Este algoritmo é conhecido como *Doug Lea's Malloc* (Lea, 1996) em homenagem ao seu criador (Doug Lea) e faz uma combinação de dois dos algoritmos mencionados (*best-fit* e listas de lacunas por tamanho).

Muitas vezes quando se conhece a forma que uma aplicação aloca memória, pode-se implementar métodos mais eficientes e com uma menor taxa de fragmentação para essa aplicação específica. Isto é, se a aplicação somente utiliza blocos de uma mesmo tamanho, então o algoritmo de gerenciamento de memória não precisa lidar com tamanhos variáveis, e tudo fica mais simples. Se a aplicação aloca memória no inicio de sua execução e utiliza a memória até o fim, então não é necessário gerenciar os blocos liberados. Essas são apenas algumas pistas que poderão levar a um método ótimo para um caso específico.

```
struct  Iaddress {
        void  * direto [ N_ZONES ];  /* Blocos  diretos     */
        #ifdef _IN_SIMPLES
        void  * ind_simples;         /* Indiretos  simples  */
        #endif
        #ifdef _IN_DUPLA
        void  * ind_duplo;           /* Indiretos  duplos   */
        #endif
};
union  address {
        struct  Iaddress iaddress;   /* Arq. Indexados      */
        void  * fixo;                /* Arq. Fixos          */
};

struct  file_struct{
        #ifdef USA_NOMES_PARA_ARQUIVOS
        char    name [ NAME_SIZE + 1 ];
        #endif
        byte errno;
        #ifdef LOOKUP_SEQU
                byte used;
        #endif
        struct  file_status status;
        union  address storage;
};
```

Código 6: Estruturas para representar arquivos

Para este trabalho, a unidade de gerenciamento de memória deve prover suporte para alocação contígua (utilizada em arquivos fixos), alocação em blocos de um mesmo tamanho (utilizada em arquivos indexados) e alocação mista (para o caso em que os arquivos fixos coexistem com os arquivos indexados).

- Indexado - A alocação indexada é bastante simples, pois o tamanho de cada bloco é sempre fixo. Uma forma bastante eficiente de implementá-la é fazendo que os blocos sejam dispostos numa lista encadeada, cada um deles aponta para o bloco seguinte e o último bloco aponta para nulo (indicando que não existem mais blocos na lista), como na figura 4.5.

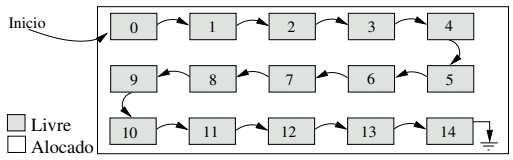


Figura 4.5: Toda a memória livre

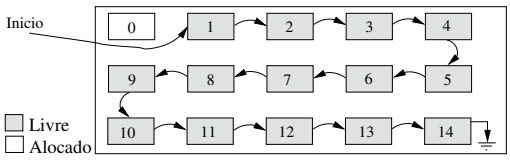


Figura 4.6: Após uma alocação

Na inserção, basta retornar o endereço do bloco indicado por “livre” e atualizar a variável “livre” para apontar para o próximo bloco na lista, como na figura 4.6. Na liberação de memória, o bloco liberado é inserido como novo início da lista e feito apontar para o endereço armazenado

em “livre”, logo a seguir, “livre” é feito apontar para este bloco. Estas operações são realizadas com complexidade $O(1)$.

- Fixo - A alocação fixa (ou contígua) é uma alocação seqüencial de blocos de memória de tamanho variável na qual as alocações iniciam-se num dos extremos da memória e progridem em direção ao outro extremo, figura 4.7. Para satisfazer uma requisição de memória de tamanho “X”, inicialmente verifica-se se existe uma lacuna maior ou igual a “X”.

  Caso exista espaço, a lacuna é dividida, parte dela será alocada (com tamanho “X”) e o restante continuará disponível. Um marcador que indica o início da lacuna livre será deslocado para o início da nova área de memória livre e alocações subseqüentes iniciarão nesta marca, conforme figura 4.8.

  No caso de uma nova requisição (desta vez de tamanho “Y”) novamente a lacuna será dividida, parte dela será alocada e o restante continuará disponível, caso mostrado na figura 4.9. A desalocação é um pouco mais complexa, no exemplo da figura 4.9 a desalocação da primeira região (região de tamanho “X”) tornará esta área disponível novamente, como na figura 4.10.

  Além disso, na desalocação de uma região verifica-se o estado (alocado ou disponível) dos vizinhos da esquerda e da direita da região alvo. Na figura 4.10, pela esquerda temos o início da memória e pela direita existe uma região alocada, então nada pode ser feito.

  Quando a região de tamanho “Y” for desalocada, esta será marcada como disponível e os seus vizinhos serão examinados, figura 4.11. Examinado o vizinho da esquerda, encontramos uma região marcada como disponível e pela direita temos a lacuna restante, também marcada como disponível. Neste caso, o algoritmo deve unir as regiões disponíveis e deslocar o marcador de início da lacuna para o extremo inferior, como na figura 4.12

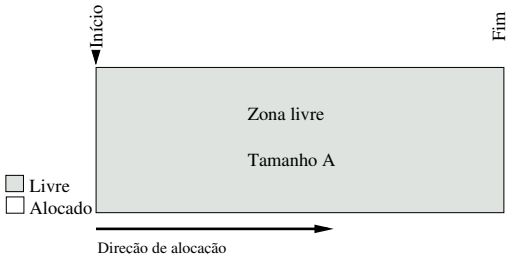


Figura 4.7: Estado inicial

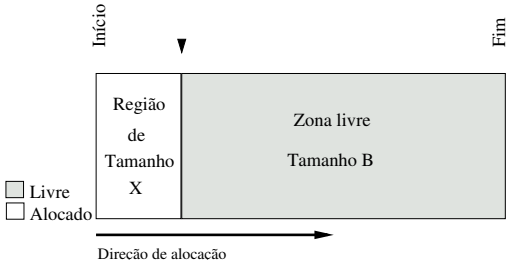


Figura 4.8: Após uma alocação

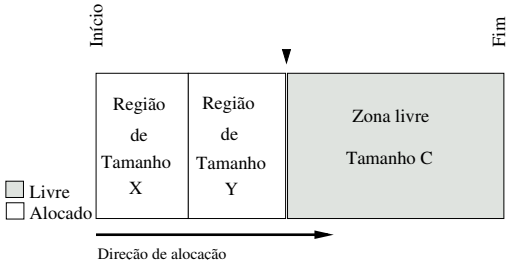


Figura 4.9: Após duas alocações

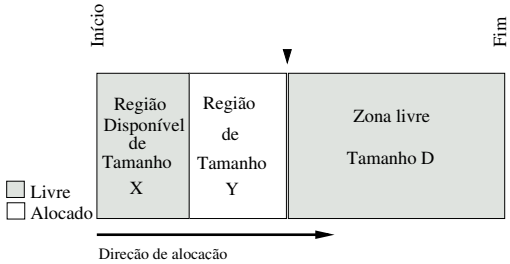


Figura 4.10: Após uma desalocação

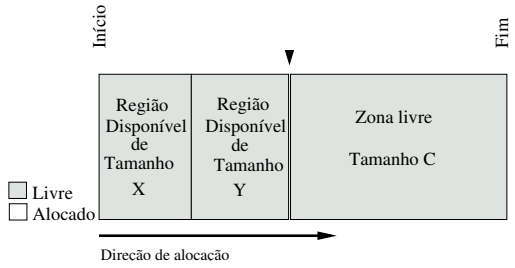


Figura 4.11: Após duas desalocações

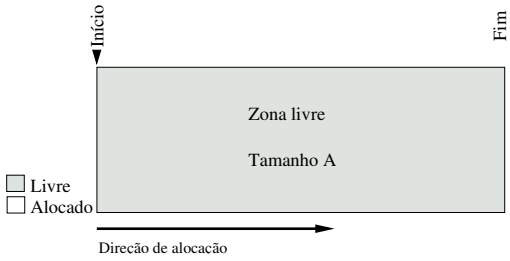


Figura 4.12: Após união de segmentos

- Misto - Para a alocação mista é necessária a funcionalidade obtida com a alocação fixa e com a alocação indexada, para tanto os dois algoritmos anteriores são utilizados com pequenas modificações. Agora, cada um deles iniciará num dos extremos da memória como na figura 4.13. Inicialmente a área disponível para alocação indexada possuí tamanho nulo. Quando for requisitado a alocação de um bloco da região indexada, uma falta de memória será retornada. Neste caso, o algoritmo tentará re-estruturar a memória, isto é, ele irá reduzir o tamanho da região de memória para alocação fixa e esse espaço será utilizado para a alocação indexada. Sempre que a memória para blocos indexados se tornar insuficiente, será feita uma tentativa de empréstimo de espaço da outra região, como na figura 4.14.

  Esse sistema misto funciona bem para o caso em que os arquivos fixos sejam criados no início da aplicação e permaneçam até o seu final, isto é, que não ocorram freqüentes alocações e desalocações de blocos fixos visto que esta estratégia de alocação gera muita fragmentação externa que só é eliminada quando as regiões adjacentes são reunidas. Tanto o algoritmo de alocação indexada, alocação fixa e alocação mista somente realizam poucas operações e possuem uma complexidade $O(1)$.

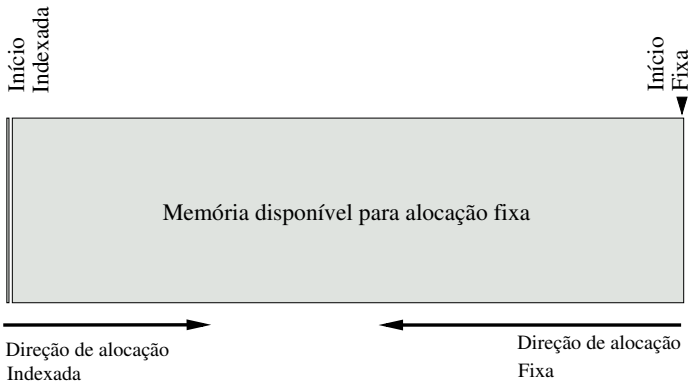


Figura 4.13: Estado inicial

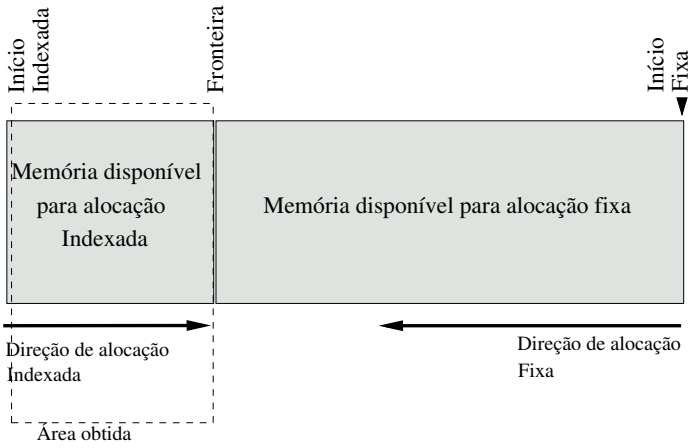


Figura 4.14: Após a reestruturação

4.3 Personalização do sistema de arquivos

4.3.1 Tipos e Meta-Tipos

No capítulo sobre SA foram analisadas as maneiras de se implementar arquivos (alocação fixa e indexada) bem como as formas de utilizá-los. Neste trabalho essas informações foram utilizadas para construir a unidade básica de composição do sistema de arquivos personalizado. Desta forma, foi instituída como unidade básica de composição o **tipo** de arquivo.

Um tipo de arquivo pode ser descrito como um objeto que herda a semântica de operação de um **Meta-Tipo** e alguns métodos que este implementa. A relação meta-tipo x tipo está ilustrada na figura 4.15

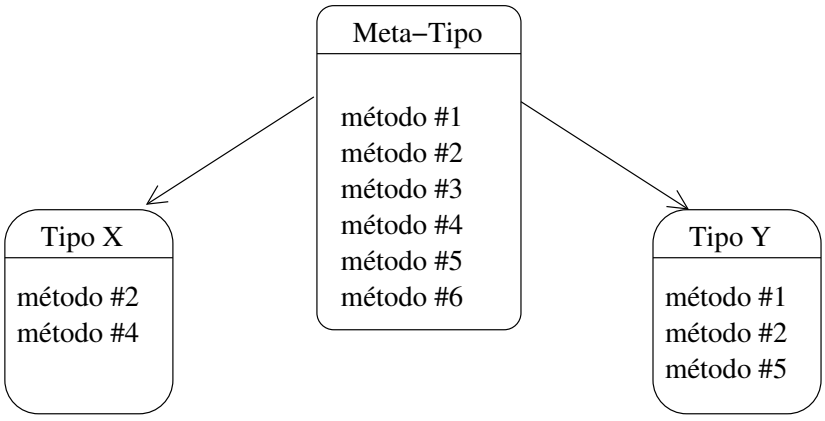


Figura 4.15: Relação Meta-tipo e Tipo

Na figura 4.15 pode-se ver que foram criados dois tipos que herdam a semântica de funcionamento de um Meta-Tipo e cada um desses tipos utiliza somente alguns métodos. Então o **Tipo X** possui agora os métodos 2 e 4, enquanto que o **Tipo Y** possuí os métodos 1, 2 e 5.

Neste sistema de arquivos foram criados três Meta-Tipos que estão ilustrados na figura 4.16 com

seus métodos.

Como pode se observar na figura 4.16, os Meta-Tipos possuem métodos com funções semelhantes (funções para leitura, escrita, etc.), embora a implementação desses métodos mude para cada um dos Meta-Tipos, objetivando usar os recursos das diferentes formas de se implementar arquivos. Desta forma, uma chamada a `read()` sobre um arquivo baseado no Meta-Tipo Fixo irá funcionar de uma forma e a mesma chamada sobre um arquivo baseado no Meta-Tipo Indexado funcionará de outra forma. Mesmo com diferentes implementações, a interface pela qual o programador vê o SA é a mesma. A seguir, temos em detalhe a semântica e função de cada um destes.

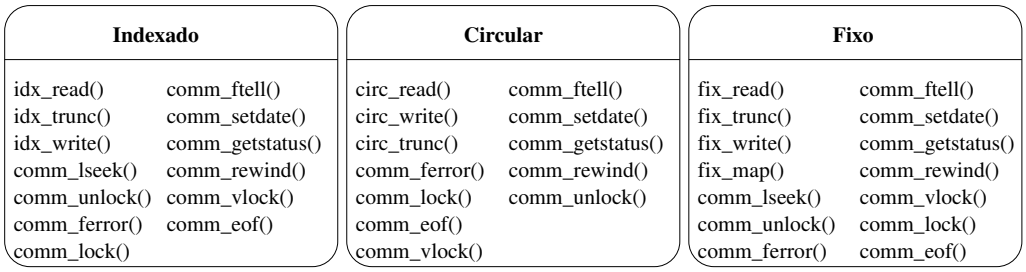


Figura 4.16: Os três Meta-tipos existentes

Indexado Este Meta-Tipo implementa arquivos usando uma estrutura Indexada, isto é, quando o arquivo é criado ele possui tamanho nulo, a cada operação de adição de dados seu tamanho é aumentando. Internamente as escritas e leituras são realizadas em blocos de dados cujo tamanho é especificado pelo SA induzindo um custo para estes arquivos, mas possuindo o benefício de tamanho variável. Deve ser utilizado somente quando for necessário um arquivo de tamanho que não é conhecido no momento da criação.

Funções:

1. `idx_read (int fd, void * buffer, int size)` - Lê *size* bytes de um arquivo indexado e armazena esses bytes em um *buffer*. A cada operação de leitura o indicador de posição relativa do arquivo é incrementado.
2. `idx_trunc (int fd, size_t size)` - Corta o arquivo em determinado tamanho. Os dados contidos além desta posição serão apagados e a memória ocupada será desalocada.
3. `comm_unlock (int fd)` - Desbloqueia um arquivo.
4. `comm_ferror (int fd)` - Verifica o status de erro de um arquivo.
5. `idx_write (int fd, void * buffer, int size)` - Escreve *size* bytes de um buffer apontado por *buffer* em um arquivo indexado. Incrementa a posição relativa no arquivo. Se este ponteiro de posição relativa estiver no final do arquivo, uma chamada de escrita irá alocar mais memória para comportar os dados, isto é o tamanho do arquivo será incrementado.
6. `comm_vlock (int fd)` - Verifica se um arquivo está bloqueado.
7. `comm_rewind (int fd)` - Posiciona o ponteiro de posição relativa no início do arquivo.

8. comm_getstatus (int fd, struct file_status *st) - Obtém informações de status de um arquivo.
9. comm_ftell (int fd) - Retorna a posição corrente do ponteiro de posição relativa do arquivo.
10. comm_lseek (int fd, off_t offset, int whence) - Ajusta a posição relativa do arquivo.
11. comm_setdate (int fd, date_t *date) - Ajusta a data de um arquivo.
12. comm_lock (int fd) - Sinaliza o bloqueio de um arquivo.
13. comm_eof(int fd) - Testa se o ponteiro de posição relativa está apontando para o fim do arquivo.

Fixo Este Meta-Tipo implementa uma estrutura com espaço fixo pré-alocado e especificado no momento da criação do arquivo, assim não é possível aumentá-lo. Este Meta-Tipo é bastante eficiente pois o acesso aos seus dados é direto, sem o *overhead* de agrupar e desagrupar bytes em blocos de dados. Permite uma maior eficiência nas funções de read/write. Esta estrutura interna se adapta melhor a arquivos que sejam criados uma vez e nunca destruídos, tais como arquivos de configuração.

Funções:

1. fix_read (int fd, void * buffer, int size) - Lê *size* bytes de um arquivo fixo e armazena esses bytes em um *buffer*. A cada operação de leitura o indicador de posição relativa do arquivo é incrementado.
2. fix_trunc (int fd, size_t size) - Corta o arquivo em determinado tamanho. Os dados contidos além desta posição serão apagados mas a memória não será desalocada até que o arquivo seja eliminado.
3. comm_unlock (int fd) - Desbloqueia um arquivo.
4. comm_ferror (int fd) - Verifica o status de erro de um arquivo.
5. fix_write (int fd, void * buffer, int size) - Escreve *size* bytes de um buffer apontado por *buffer* em um arquivo fixo. Incrementa a posição relativa no arquivo. Se este ponteiro de posição relativa estiver no final do arquivo nenhum dado será armazenado
6. comm_vlock (int fd) - Verifica se um arquivo está bloqueado.
7. comm_rewind (int fd) - Posiciona o ponteiro de posição relativa no início do arquivo.
8. comm_getstatus (int fd, struct file_status *st) - Obtém informações de status de um arquivo.
9. comm_ftell (int fd) - Retorna a posição corrente do ponteiro de posição relativa do arquivo.
10. comm_lseek (int fd, off_t offset, int whence) - Ajusta a posição relativa do arquivo.
11. comm_setdate (int fd, date_t *date) - Ajusta a data de um arquivo.

12. comm_lock (int fd) - Sinaliza o bloqueio de um arquivo.
13. comm_eof (int fd) - Testa se o ponteiro de posição relativa está apontando para o fim do arquivo.
14. fix_map (int fd, int position, void *address) - Mapeia um arquivo para um endereço de memória e retorna um ponteiro para esse endereço. Acessos de leitura/escrita ao endereço associado ao arquivo serão acessos aos dados deste arquivo.

Circular Este Meta-Tipo implementa uma estrutura com espaço fixo pré-alocado e especificado no momento da criação do arquivo, assim não é possível aumentá-lo. Funciona como um buffer circular cujo tamanho é pré-determinado na criação do arquivo. Operações de leitura e escrita incrementam a posição atual dentro deste buffer circular. Quando este chega na sua posição final, retorna ao início.

Funções:

1. circ_read (int fd, void * buffer, int size) - Lê *size* bytes de um arquivo circular e armazena esses bytes em um *buffer*. A cada operação de escrita o indicador de posição relativa do arquivo é deslocado segundo o funcionamento de uma lista circular.
2. circ_trunc (int fd, size_t size) - Corta o arquivo em determinado tamanho. Os dados contidos além desta posição serão apagados mas a memória não será desalocada até que o arquivo seja eliminado.
3. comm_unlock (int fd) - Desbloqueia um arquivo.
4. comm_ferror (int fd) - Verifica o status de erro de um arquivo.
5. circ_write (int fd, void * buffer, int size) - Escreve *size* bytes de um buffer apontado por *buffer* em um arquivo circular. Desloca a posição relativa no arquivo segundo uma lista circular.
6. comm_vlock (int fd) - Verifica se um arquivo está bloqueado.
7. comm_rewind (int fd) - Posiciona o ponteiro de posição relativa no início do arquivo.
8. comm_getstatus (int fd, struct file_status *st) - Obtém informações de status de um arquivo.
9. comm_ftell (int fd) - Retorna a posição corrente do ponteiro de posição relativa do arquivo.
10. comm_setdate (int fd, date_t *date) - Ajusta a data de um arquivo.
11. comm_lock (int fd) - Sinaliza o bloqueio de um arquivo.
12. comm_eof (int fd) - Testa se o ponteiro de posição relativa está apontando para o fim do arquivo.

4.3.2 Definições do sistema de arquivos

A liberdade de composição do sistema de arquivos, não está presente somente na criação de Tipos e na escolha de suas funções internas. Para um maior controle e personalização deste, existem

outros parâmetros que são usados na geração de código. Na composição do sistema de arquivos o programador deve utilizar esses parâmetros para ajustar opções do novo sistema de arquivos. A tabela 4.2 mostra esses parâmetros, sua semântica e um exemplo de uso.

Parâmetro	Semântica	Exemplo de uso
Número de arquivos	Especifica o número máximo de arquivos que o SA é capaz de manipular.	#define N_FILES 100
Tamanho máximo	Especifica o tamanho máximo de um arquivo em bytes.	#define MAX_SIZE 4096
Tamanho bloco	Especifica o tamanho de bloco para arquivos indexados.	#define BLOCK_SIZE 1024
Identificação arquivos	Seleciona se o SA deve usar nomes de arquivos para identificar arquivos ou se deve usar números.	#define USA_NOMES_PARA_ARQUIVOS
Tamanho nome	Ajusta quantas letras serão usadas para nomear arquivos, somente funciona se o SA estiver definido para usar nomes para identificar arquivos.	#define NAME_SIZE 8
Caixa alta	Ajusta se maiúsculas e minúsculas são tratadas iguais.	#define SENSIVEL_CASO
Tamanho memória	Seleciona o tamanho da memória para o SA.	#define DATA_SIZE 100000
Gerência de memória	Seleciona o algoritmo de gerenciamento de memória. Este pode ser FIX (alocação somente para arquivos fixos e circulares), INDX (somente para arquivos indexados), MST (suporta ambos).	#define MM MST
Datas de arquivos	Ajusta de o SA deve armazenar informações de data (tais como data de criação, última leitura, última escrita), isto é usado por funções como getstatus() e setdate() .	#define USE_DATE
Meta-datas	Seleciona se a data dos arquivos deve ser simples ou avançada, isto é data simples é composta somente do momento da criação do arquivo, já data avançada é composta do momento da criação do arquivo e também do momento da última operação de leitura e de escrita sobre o arquivo.	#define METADATA ADV
Troca nome	Ajusta se o SA deve possuir uma função para mudar o nome dos arquivos *rename(nome_anterior, novo_nome)*.	#define API_RENAME
Elimina arquivos	Ajusta se o SA deve possuir uma função de eliminação de arquivos *(unlink(nome))*.	#define API_UNLINK
Lookup	Ajusta qual algoritmo de lookup será usado. O algoritmo de lookup somente será usado caso o SA for composto usando nomes para indicar arquivos. Algoritmos possíveis: Árvores *Red-Black* (LOOKUP_RB), Pesquisa Seqüencial (LOOKUP_SEQUENTIAL).	#define LOOKUP_RB

Tabela 4.2: Opções extras para personalizar o Sistema de Arquivos

4.4 Geração de código

Após a etapa de análise de requisitos para o novo sistema de arquivos vem a etapa de geração de código. Nesta etapa, será mostrado como os Tipos de arquivos apresentados no ítem 4.3.1 e as definições do sistema de arquivos (ítem 4.3.2) são criados. A figura 4.17 mostra os módulos envolvidos no processo de geração de código.

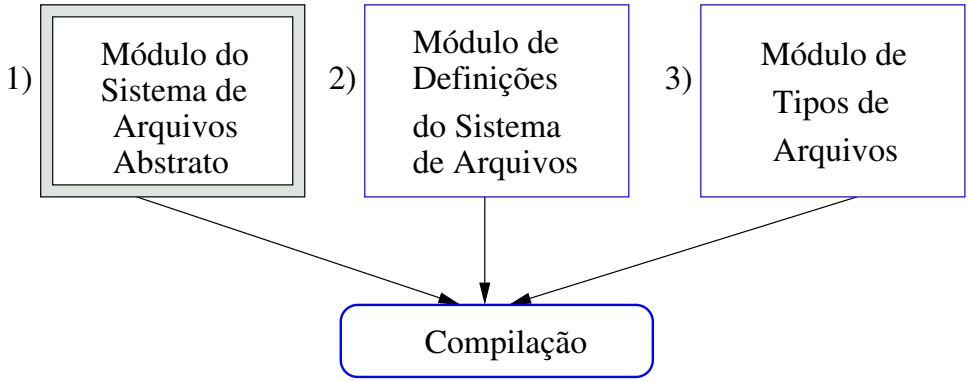


Figura 4.17: Módulos envolvidos na geração de um Sistema de Arquivos

4.4.1 Criação de tipos

A criação de tipos de arquivos é feita através da associação de funções membros herdados de um mesmo Meta-Tipo. Inicialmente este Meta-Tipo deve ser selecionado dentre os três existentes, após este passo, as funções pertinentes para compor o novo tipo de arquivo são associadas.

Para compreender a associação de funções, é necessário observar a estrutura de tipos, mostrada no código 7. Esta estrutura é composta por 14 campos, cada qual é um ponteiro para uma função. Quando se cria um novo tipo, o que se faz é associar um endereço onde existe uma função a um campo desta estrutura.

```
struct  file_operations {
#ifdef  API_READ
        ssize_t (*read)(int fd, void *buf, size_t count);
#endif
#ifdef  API_WRITE
        ssize_t (*write)(int fd, void *buf, size_t count);
#endif
#ifdef  API_LSEEK
        off_t (*lseek)( int fd, off_t offset, int whence);
#endif
#ifdef  API_TRUNC
        void   (*trunc)(int fd, ssize_t size);
#endif
#ifdef  API_GET_STATUS
        void   (*get_status)(int fd, struct  file_status *st);
#endif
#ifdef  API_SET_DATE
        void   (*set_date)( int fd, date_t *date);
#endif
#ifdef  API_REWIND
        void   (*rewind)( int fd);
#endif
#ifdef  API_FTELL
        off_t (*ftell)( int fd );
#endif
#ifdef  API_LOCK
        int    (*lock)(int fd);
#endif
#ifdef  API_UNLOCK
        void    (*unlock)(int fd);
#endif
#ifdef  API_VLOCK
        int    (*verify_lock)(int fd);
#endif
#ifdef  API_EOF
        int    (*eof)(int fd);
#endif
#ifdef  API_FERROR
        int    (*ferror)(int fd);
#endif
#ifdef  API_MAP
        int  (*fmap)(int fd, int position, void *address);
#endif
};
```

Código 7: Estrutura de funções membro

O fragmento de código 8 mostra esta associação através da inicialização estática da estrutura de funções membro. Neste código é mostrada a inicialização de um vetor estrutura de tamanho três, sendo assim, três tipos estão sendo definidos.

```
struct  file_operations FileTypes [ 3 ]= {
    {fix_read,fix_write,comm_lseek,fix_trunc,comm_get_status,
    comm_set_date,comm_rewind,comm_ftell,comm_lock,comm_unlock,
    comm_vlock,comm_eof,comm_ferror,fix_map },
    {idx_read,idx_write,comm_lseek,idx_trunc,comm_get_status,
    comm_set_date, comm_rewind,comm_ftell,comm_lock,comm_unlock,
    comm_vlock,comm_eof, comm_ferror,NULL },
    {circ_read,cir_write,NULL,circ_trunc,comm_get_status,
    comm_set_date,comm_rewind,comm_ftell,comm_lock,comm_unlock,
    comm_vlock,comm_eof,comm_ferror,NULL }
};
```

Código 8: Exemplo de criação dos tipos

O tipo definido na posição 0 é derivado do Meta-Tipo fixo, pois utiliza funções deste Meta-Tipo como *fix_read*, *fix_write* e *fix_trunc*, o tipo definido na posição 1 utiliza funções *idx_read*, *idx_write* entre outras, então ele é derivado do Meta-Tipo indexado. O último tipo é derivado do Meta-Tipo circular, pois ele utiliza funções deste Meta-Tipo.

Para facilidar o uso, o programador deve definir nomes para utilizar os arquivos ao invés de usar os números diretamente. Essas definições de nomes para posições da estrutura de funções membro são incluídas no arquivo de definições do SA são mostradas no código 9.

```
#define  MEU_TIPO_FIXO 0
#define  MEU_TIPO_CIRCULAR 2
#define  MEU_TIPO_INDEXADO 1+INDEX_BASE
/* Por questoes de implementacao , os tipos baseados       */
/* em Meta-Tipo indexado devem ser declarados desta forma */
```

Código 9: Definição de nomes

4.4.2 Criação do arquivo das definições

Para criar um arquivo de definições, deve-se definir os parâmetros mostrados na tabela 4.2 para valores consistentes com o objetivo do sistema de arquivos, criando um arquivo de cabeçalho (`.h`). Assim, nesse arquivo pode-se especificar qual a quantidade de memória, número máximo de arquivos, etc. Além dessas informações, outras também devem ser inseridas neste arquivo, tais como definições de algumas palavras reservadas, definições de quais funções são usadas (para cada um dos tipos de arquivos) e definição de quais rotinas da *API* (*Application Program Interface*) estarão disponíveis.

Como já foi comentando anteriormente, o SAA é composto por uma grande quantidade de código com a implementação de vários componentes. Cada função presente no SAA foi escrita dentro de trecho de compilação condicional, usando o #ifdef e #endif do pré-compilador do C, tal como no exemplo do código 10.

```
#ifdef  FIX_READ
PUBLIC ssize_t fix_read ( int fd, void *buf, size_t count )
{
    .
    .
    .
}
#endif
#ifdef  FIX_WRITE
PUBLIC ssize_t fix_write ( int fd, void *buf, size_t count )
{
    .
    .
    .
}
#endif
```

Código 10: Fragmento de código do SAA

Quando cria-se um arquivo de definições, o que se faz é definir os trechos de código que serão compilados. Definindo-se o nome FIX_READ e FIX_WRITE faz com que o código das funções fix_read e fix_write seja incluído para compilação.

Um exemplo de um arquivo (.h) com as definições de parâmetros de um sistema de arquivos pode ser visto no código 11.

```
#ifndef  _CUSTOM
#define  _CUSTOM
/* Definicao  para  uso  interno  */
#define  INDEX_BASE 1000
#define  SEEK_SET 0
#define  SEEK_CUR 1
#define  SEEK_END 2
#define  SIM 1
#define  ADV 2
#define  NONE 0
#define  O_CREAT 2048
#define  O_FIX 2
```

```
#define  O_IDX 4
#define  O_CIRC 8
#define  FIX 0
#define  INDEX 1
#define  MST 2
/* Definicao  dos nomes dos tipos  */
#define  TIPO_FIX 0
#define  TIPO_INDEX INDEX_BASE+1
#define  TP_CIRC 2
/* Lista  das funcoes  utilizadas   */
#define  FIX_READ
#define  FIX_WRITE
#define  COMMON_SEEK
#define  FIX_TRUNC
#define  COMMON_GET_STATUS
#define  COMMON_SETDATE
#define  COMMON_REWIND
#define  COMMON_FTELL
#define  COMMON_LOCK
#define  COMMON_UNLOCK
#define  COMMON_VLOCK
#define  COMMON_EOF
#define  COMMON_FERROR
#define  FIX_MAP
#define  CIRC_READ
#define  CIRC_WRITE
#define  IDX_TRUNC
#define  IDX_READ
#define  IDX_WRITE
/* Rotinas  da API que estarao  disponiveis   */
#define  API_READ
#define  API_WRITE
#define  API_LSEEK
#define  API_TRUNC
#define  API_GET_STATUS
#define  API_SET_DATE
#define  API_REWIND
#define  API_FTELL
#define  API_LOCK
#define  API_VLOCK
#define  API_EOF
#define  API_FERROR
#define  API_MAP
/* Definicoes  do SA personalizado   */
#define  USE_DATE
#define  N_FILES 10
```

```
#define  METADATA ADV
#define  USA_NOMES_PARA_ARQUIVOS
#define  NAME_SIZE 8
#define  LOOKUP_RB
#define  DATA_SIZE 100000
#define  BLOCK_SIZE 1024
#define  MM MST
#define  MAX_FILE_SIZE 4096
#endif
```

Código 11: Arquivo de definição do Sistema de Arquivos.

4.4.3 Compilação

No módulo do SAA estão implementados toda a funcionalidade do Sistema de Arquivos, os vários algoritmos para gerenciar memória (alocação e liberação), algoritmos para manipular arquivos (tanto a estrutura de arquivos quanto o gerenciamento), para lidar com os Meta-Tipos, etc.

Para gerar um sistema de arquivos, devemos combinar todos os módulos, compilar e gerar o arquivo objeto final. Para facilitar esta tarefa, é recomendado que se utilize um *Makefile*. Um *makefile* é um arquivo que possui uma listagem de todos os arquivos de um projeto que devem ser compilados e estabelece algumas regras de dependência entre estes, de forma a otimizar o tempo de sucessivas compilações.

Uma ferramenta chamada *make*[3] utiliza o arquivo *Makefile* para poder executar esta tarefa. O *Makefile* para compilar o sistema de arquivos personalizado é mostrado 52 e possui referência aos dois arquivos necessários (definições e de tipos).

Depois de compilar todos os módulos, o *make* irá unir estes arquivos objeto e criar uma biblioteca estática.

```
# Written by Fabio Rodrigues de la Rocha <frr@das.ufsc.br>
# ------------------------------------------------------
# Project               :novo.proj
# Working Path          :/home/mestrado/frr/tool/classes
# Data                  :Mon Nov 04 15:01:49 GMT 2002

# Compiler name (ie gcc, tcc)
CC=gcc
# shell command to remove files (ie: rm, del)
RM=rm
```

[3] Mais informações sobre o *Make* podem ser obtidas em http://www.gnu.org/software/make/make.html.

```
# remove flags
# linker
BIB=ld
# Linker flags
#target path
PROJ_DIR=/home/mestrado/frr/tool/classes
SOURCE_DIR=/home/mestrado/frr/mestrado/implementacao/ram

ARQ_DE_DEFINICOES=minhas_definicoes  #Arquivo com as definicoes
ARQ_DE_TIPOS=meus_tipos    #Arquivo com os tipos suportados pelo SA


BAR=/
CFLAGS = -o$@ -DCUSTOM_H=\"$(PROJ_DIR)/$(ARQ_DE_DEFINICOES).h\" \
-Wall -ansi -pedantic

BIBFLAGS=-o$@ -r
LOOKUP = lookup.obj

# Link all object files in one.
novo.obj:  memory.obj  mm.obj  time.obj\
$(LOOKUP)  function.obj  $(ARQ_DE_TIPOS).obj  api.obj semaphore.obj
        $(BIB) $(BIBFLAGS)   memory.obj  mm.obj  time.obj\
$(LOOKUP)  function.obj  $(ARQ_DE_TIPOS).obj  api.obj semaphore.obj \

api.obj: $(SOURCE_DIR)$(BAR)api$(BAR)api.c $(ARQ_DE_TIPOS).obj \
$(SOURCE_DIR)$(BAR)types$(BAR)types.h  mm.obj \
$(SOURCE_DIR)$(BAR)fstruct$(BAR)fstruct.h \
$(SOURCE_DIR)$(BAR)api$(BAR)api.h \
memory.obj $(LOOKUP)  time.obj
        $(CC) $(CFLAGS) -c $(SOURCE_DIR)$(BAR)api$(BAR)api.c

$(LOOKUP): $(SOURCE_DIR)$(BAR)lookup$(BAR)lookup.c \
$(SOURCE_DIR)$(BAR)types$(BAR)types.h\
 $(ARQ_DE_TIPOS).obj $(SOURCE_DIR)$(BAR)fstruct$(BAR)fstruct.h\
 $(SOURCE_DIR)$(BAR)api$(BAR)api.h  memory.obj \
$(SOURCE_DIR)$(BAR)lookup$(BAR)lookup.h
        $(CC)  $(CFLAGS) -c  $(SOURCE_DIR)$(BAR)lookup$(BAR)lookup.c

$(ARQ_DE_TIPOS).obj:  $(ARQ_DE_TIPOS).c \
$(SOURCE_DIR)$(BAR)types$(BAR)types.h \
 $(ARQ_DEFINICOES).h \
$(SOURCE_DIR)$(BAR)fstruct$(BAR)fstruct.h \
$(SOURCE_DIR)$(BAR)api$(BAR)api.h function.obj
        $(CC) $(CFLAGS) -c  $(ARQ_DE_TIPOS).c
```

```
function.obj: $(SOURCE_DIR)$(BAR)function$(BAR)function.c\
 $(SOURCE_DIR)$(BAR)types$(BAR)types.h  $(ARQ_DEFINICOES).h \
$(SOURCE_DIR)$(BAR)fstruct$(BAR)fstruct.h\
 $(SOURCE_DIR)$(BAR)function$(BAR)function.h mm.obj time.obj
        $(CC)  $(CFLAGS) -c \
$(SOURCE_DIR)$(BAR)function$(BAR)function.c

memory.obj: $(SOURCE_DIR)$(BAR)memory$(BAR)memory.c \
$(SOURCE_DIR)$(BAR)types$(BAR)types.h\
 $(ARQ_DEFINICOES).h $(SOURCE_DIR)$(BAR)super$(BAR)super.h \
$(SOURCE_DIR)$(BAR)fstruct$(BAR)fstruct.h \
$(SOURCE_DIR)$(BAR)api$(BAR)api.h
        $(CC)  $(CFLAGS) -c $(SOURCE_DIR)$(BAR)memory$(BAR)memory.c

time.obj: $(SOURCE_DIR)$(BAR)time$(BAR)time.c \
$(SOURCE_DIR)$(BAR)time$(BAR)time.h \
$(SOURCE_DIR)$(BAR)types$(BAR)types.h
        $(CC)  $(CFLAGS) -c $(SOURCE_DIR)$(BAR)time$(BAR)time.c

mm.obj: $(SOURCE_DIR)$(BAR)mm$(BAR)mm.c \
$(SOURCE_DIR)$(BAR)types$(BAR)types.h  $(ARQ_DEFINICOES).h\
$(SOURCE_DIR)$(BAR)fstruct$(BAR)fstruct.h \
$(SOURCE_DIR)$(BAR)mm$(BAR)mm.h
        $(CC)  $(CFLAGS) -c $(SOURCE_DIR)$(BAR)mm$(BAR)mm.c
semaphore.obj: $(SOURCE_DIR)$(BAR)semaphore$(BAR)semaphore.c  \
$(ARQ_DEFINICOES).h\
$(SOURCE_DIR)$(BAR)semaphore$(BAR)semaphore.h
        $(CC)  $(CFLAGS) -c \
$(SOURCE_DIR)$(BAR)semaphore$(BAR)semaphore.c

clean:
        $(RM) *.obj
```

Código 12: Makefile para compilar o SA personalizado.

4.5 Visão do programador de aplicação

A interface de programação é a camada mais externa do módulo de SAA (Sistema de Arquivos Abstrato). Através dela, o programador interage com o sistema de arquivos, criando, apagando, lendo e escrevendo arquivos. Por se tratar de uma parte na qual o programador da aplicação final tem contato, é importante que as interfaces da *API* sejam simples e bem conhecidas para evitar o desperdício de tempo com o aprendizado de uma nova *API*. Por esta razão, foram implementadas rotinas de acesso aos arquivos que se aproximam do padrão POSIX 1003.1(IEEE, 1996), dada a sua grande

aceitação. As diferenças em relação ao POSIX são encontradas nas funções `open()` e `creat()`, que neste sistema devem trabalhar com tipos de arquivos e que podem utilizar números para referenciar arquivos. Também existe uma função para inicialização do SAP, chamada de `build_rtfs()`..

4.5.1 Resumo da interface de programação em C

Esta seção apresenta um resumo da interface de programação em C. A descrição completa da *API* aparece no anexo A.

Já no anexo B, temos diagramas da invocação de funções. Estes diagramas mostram quais são as funções internas ao sistema de arquivos que são chamadas por cada uma das funções disponíveis para o programador. Também estão destacados quais são os semáforos utilizados.

---

sintaxe:

void **build_rtfs** (*void*);

descrição:

A função **build_rtfs** prepara o Sistema de Arquivos para uso. Ela deve ser chamada somente uma vez na aplicação para inicializar os componentes do sistema antes de utilizar qualquer outra função da API.

---

sintaxe:

int **open** (*const char** name, *int* flags, *int* mode);
int **open** (*const char** name, *int* flags);

descrição:

A função **open** abre um arquivo, ou cria um novo arquivo de nome *name* de acordo com o tipo passado em *flags* definido durante a concepção do sistema de arquivos. Caso o arquivo não exista, ele deve ser criado fazendo uma máscara do tipo de arquivo com *O_CREAT*, como por exemplo: $TYPE\_RECORD|O\_CREAT$.
O parâmetro *mode* pode conter informações necessárias para um dado tipo de arquivo, mas não é sempre aplicável. Arquivos de tamanho fixo, ou que fazem uso de *buffers* circulares utilizam este campo para ajuste do tamanho do arquivo no momento da criação.

---

sintaxe:

int **open** (*int* fd, *int* flags, *int* mode);
int **open** (*int* fd, *int* flags);

descrição:

A função **open** abre um arquivo, ou cria um novo arquivo identificado pelo número *fd* de acordo com o tipo passado em *flags* definido durante a concepção do sistema de arquivos. Caso o arquivo não exista, ele deve ser criado fazendo uma máscara do tipo de arquivo com *O_CREAT*, como por exemplo: $TYPE\_RECORD|O\_CREAT$. O parâmetro *mode* pode conter informações necessárias para um dado tipo de arquivo, mas não é sempre aplicável. Arquivos de tamanho fixo, ou que fazem uso de *buffers* circulares utilizam este campo para ajuste do tamanho do arquivo no momento da criação.

---

sintaxe:

int **creat** (*const char* * name, *int* flags);

descrição:

A função **creat** cria um arquivo com o nome *name*, de acordo com o tipo passado em *flags* definido durante a concepção do sistema de arquivos.

---

sintaxe:

int **creat** (*int* fd, *int* flags);

descrição:

A função **creat** cria um arquivo de número *fd*, de acordo com o tipo passado em *flags* definido durante a concepção do sistema de arquivos.

---

sintaxe:

int **close** (*int* fd);

descrição:

A função **close** remove a associação do descritor de arquivos com o arquivo em questão. Liberando recursos associados com o uso do arquivo.

sintaxe:

ssize_t **write** (*int* fd, *const void** buffer, *size_t* count);

descrição:

A função **write** escreve até *count* bytes de dados a partir do endereço dado por *buffer* no do arquivo referenciado por *fd*.

sintaxe:

ssize_t **read** (*int* fd, *void** buffer, *size_t* count);

descrição:

A função **read** lê até *count* bytes de dados do descritor de arquivos *fd* para a posição apontada por *buffer*.

sintaxe:

int **unlink** (*const char* * name);

descrição:

Esta função elimina um arquivo especificado pelo parâmetro *name*. O arquivo deve estar fechado antes de tentar removê-lo.

sintaxe:

int **unlink** (*int* fd);

descrição:

Esta função elimina um arquivo especificado pelo parâmetro *fd*. O arquivo deve estar fechado antes de tentar removê-lo.

sintaxe:

void **get_status** (*int* fd, *struct file_status* *st);

descrição:

Esta função retorna o status de um arquivo especificado pelo parâmetro *fd* e armazena essas informações na estrutura *file_status*.

sintaxe:

void **set_date** (*int* fd, *date_t* *date);

descrição:

Esta função ajusta a data de um arquivo especificado pelo descritor *fd* utilizando o tipo *date*. Note que o tipo *date* possui um número variável de campos de acordo com a composição do SA.
Se o SA foi composto usando informações avançadas (METADATA==ADV) então todas as informações estã disponíveis, mas se ele foi composto com (METADATA==SIM), então apenas algumas informações estarão disponíveis.

---

sintaxe:

void **rewind** (*int* fd);

descrição:

Esta função muda o valor da posição relativa do arquivo especificado pelo descritor *fd* para o início deste.

---

sintaxe:

int **rename** (*const char* * source, *const char* * target);

descrição:

Esta função muda o nome do arquivo especificado por *source* para *target*.

---

sintaxe:

int **trunc** (*int* fd, *ssize_t* new_size);

descrição:

Esta função reduz o tamanho de um arquivo especificado por *fd* para um novo tamanho *new_size*, que deve ser menor que o tamanho anterior do arquivo.

sintaxe:

off_t **lseek** (*int* fd, *off_t* offset, *int* whence);

descrição:

Esta função muda o valor do indicador de posição relativa do arquivo especificado por *fd* para uma nova posição especificada pelo deslocamento *offset* de acordo com *whence* como o seguinte:
SEEK_SET - O *offset* é em relação ao início do arquivo
SEEK_CUR - O *offset* relação a posição atual do arquivo
SEEK_END - O *offset* relação ao fim do arquivo
whence pode ser um valor positivo ou negativo.

---

sintaxe:

int **eof** (*int* fd);

descrição:

A função **eof** testa o indicador de final do arquivo *fd*.

---

sintaxe:

int **ferror** (*int* fd);

descrição:

Esta função testa o indicador de erro do arquivo especificado por *fd*.

---

sintaxe:

ssize_t **ftell** (*int* fd);

descrição:

Esta função retorna o valor do ponteiro de posição relativa do arquivo especificado por *fd*.

4.6 Conclusão

Neste capítulo, o sistema de arquivos desenvolvido foi apresentado em sua totalidade. O capítulo foi ricamente ilustrado por diversas figuras e seções de código descrevendo os algoritmos utilizados. No final, foi descrito um resumo da camada mais externa do SA (*API*) através da qual o programador utiliza o sistema de arquivos. A *API* completa do SA está no apêndice A.

Além disso, o sistema de arquivos descrito neste capítulo foi construído para tratar os problemas de tempo real (fazendo uso de algoritmos com tempos de pior caso convenientes e minimizando inversões de prioridade) e problemas de desperdício de memória (compondo o sistema de arquivos somente com o código necessário).

Todo o esforço em desenvolver esse sistema foi motivado pelos benefícios que ele provê, tais como aumento de produtividade, adequação a uma aplicação alvo e redução do tempo de conclusão de projetos.

Capítulo 5

Ferramenta desenvolvida

O sistema de arquivos descrito no capítulo anterior representa um passo importante para a construção rápida de *software* personalizado. Ele fornece subsídios para que o programador com pouco esforço desenvolva aplicações que utilizem arquivos, de uma forma transparente.

Infelizmente, a etapa de montagem do SA não é trivial. Vários são os pontos chave nos quais o programador deve estar atento para evitar problemas, forçando este a conhecer o SAA (Sistema de Arquivos Abstrato) mais a fundo e levando a uma questão: "*Deveria o programador do sistema de arquivos conhecer os detalhes da implementação para compor o SA personalizado ?*"

Esta questão não possui uma resposta fácil, ela irá depender fundamentalmente das necessidades do *hardware* e da aplicação. Se considerarmos que o programador necessita de um SA que se enquadre perfeitamente a um dispositivo ou a uma aplicação que possua outros requisitos que não são tratados por este projeto, neste caso, o programador necessita conhecer o SAA mais a fundo para escolher quais algoritmos e estruturas de dados poderão ser inseridos no seu novo sistema de arquivos. Ele pode inclusive, modificar o código fonte do SAA para se adaptar perfeitamente às suas necessidades.

Para este usuário, a ferramenta mais poderosa é a que permite o maior controle sobre o código gerado. Essa é uma situação rara e neste caso, o código fonte do SAA é o suficiente para dar total controle sobre a geração de SAs. No entanto, para a grande maioria dos usuários, a etapa de montagem do SA poderia ser facilitada. Muitos passos e decisões tomadas pelo usuário poderiam ser feitos automaticamente ou guiados por uma ferramenta de *software* visando um aumento de produtividade, qualidade e redução no tempo de conclusão de projetos.

Com essa idéia em mente, foi criada uma ferramenta de integração de componentes do SAA. A função desta ferramenta é funcionar como um *front-end* para este. Os arquivos que anteriormente eram criados manualmente pelo programador agora serão gerados, através da simples seleção de componentes utilizando uma interface visual.

5.1 Proposta de uma ferramenta

Neste trabalho, propõe-se um ambiente de concepção de Sistemas de Arquivos de Tempo Real, uma ferramenta para facilitar a criação do *software* que será executado em dispositivos embutidos. Nele, são tratados vários problemas que existem na programação de embutidos, tais como:

- dispositivos embutidos possuem uma pequena quantidade de memória, logo não pode existir desperdício inserindo código que não será utilizado (otimização da memória).
- as soluções específicas geram um melhor aproveitamento dos recursos de *hardware* (tais como memória e processador), então devem ser utilizadas sempre as estruturas de dados que melhor se adaptem a cada caso (otimização de algoritmos).

Partindo destes pontos como requisitos de uma ferramenta, projetou-se um ambiente de concepção de sistemas de arquivos, isto é, um *software* através do qual um projetista de *software* pode criar um sistema de arquivos, somente informando quais recursos ele necessita para esse sistema de arquivos (número de arquivos, tamanho máximo, operações de leitura, escrita, posicionamento, truncamento, etc.), sem a necessidade de codificar diretamente. Os recursos selecionados, dependem fundamentalmente do uso que a aplicação final pretende fazer do SA, dessa forma existe um relacionamento íntimo entre a aplicação final e os recursos providos. Em uma aplicação simples, somente alguns recursos do sistema de arquivos são utilizados, então não é necessário que este possua recursos que a aplicação não faz uso e que somente representam um *overhead* para o sistema.

Como resultado, a ferramenta produz um código fonte em *ANSI* C com a implementação de todas as funções necessárias para satisfazer as requisições do projetista de *software*, tal como na figura 5.1.

A implementação utiliza as estruturas de dados que melhor se enquadram para representar os arquivos e permitir que as operações selecionadas sejam executadas no menor tempo possível e com um comportamento temporal conhecido. Esta ferramenta também permite outras funções interessantes, tais como acesso fácil a informações de ajuda e geração automática de documentação (*API* do sistema de arquivos personalizado) fornecendo um manual descrevendo as rotinas disponíveis para o programador da aplicação final.

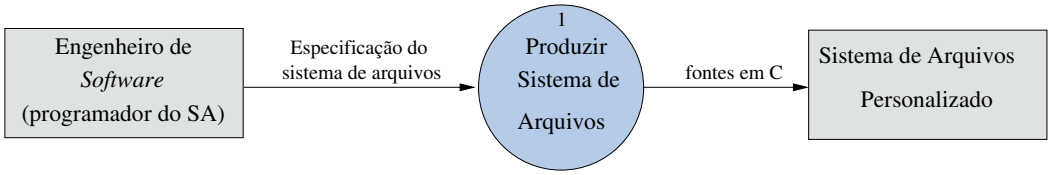


Figura 5.1: Diagrama de fluxo de dados

Uma outra característica interessante é a portabilidade da ferramenta, isto é, a capacidade de se utilizar o mesmo *software* em diferentes plataformas de *hardware* e sistemas operacionais. Para alcançar esta característica, escolheu-se codificar em Java [1] para permitir sua execução em qualquer

[1] Java é produzido e mantido pela Sun http://www.java.sun.com

ambiente computacional para o qual exista uma máquina virtual. Além disso, a ferramenta gera código (fonte) para suportar diversas arquiteturas. Na figura 5.2 contempla-se o ambiente de concepção de sistemas de arquivos executando numa arquitetura H_1 e gerando código do sistema de arquivos para outras tantas (caso 1, 2 e 3).

Além de portabilidade de código, a linguagem Java possui outros pontos que justificaram sua escolha neste trabalho, tais como:

Custo Um conjunto de ferramentas de desenvolvimento está disponível gratuitamente para *download* no *site* da Sun[2] para várias plataformas. Neste *kit* existe um compilador, um interpretador (máquina virtual), uma ferramenta para geração automática de documentação, exemplos de programas, etc.

***Interface* Gráfica** Java utiliza uma *GUI* (*Graphic User Interface*) simples e poderosa. Em aplicações onde a *interface* é um requisito fundamental, a facilidade de programação é sempre um ponto forte.

Exceções Java permite a utilização de exceções para tratar erros e separa o código do tratador de erros do código da aplicação.

Moderna Java utiliza conceitos modernos de linguagens de programação, tais como orientação a objeto, tipagem forte, eventos, coleta de lixo, etc.

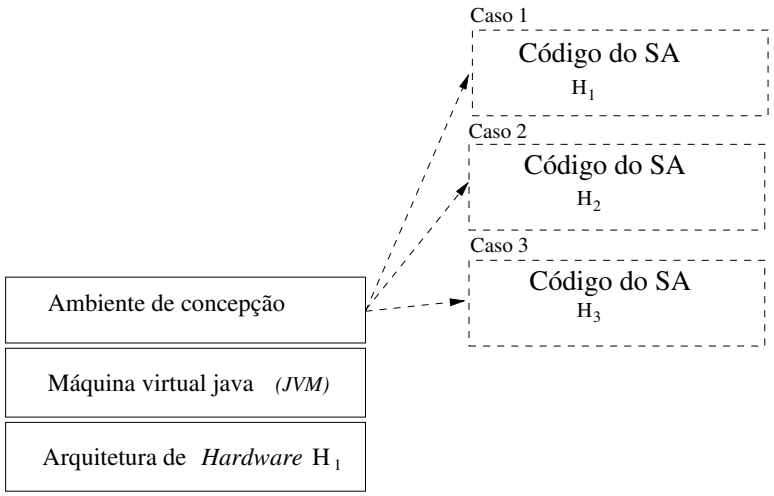


Figura 5.2: Geração de código

5.1.1 Ferramentas de desenvolvimento

Como já descrito, neste trabalho foi utilizado Java na codificação da ferramenta proposta. Além disso foram utilizados os seguintes *softwares*, motivados principalmente pela qualidade e disponibilidade sem ônus:

[2]Recentemente a IBM produziu o seu próprio conjunto de ferramentas de desenvolvimento, com os mesmos recursos das ferramentas da Sun. O *kit* de desenvolvimento da IBM pode ser obtido em http://www.ibm.com

- Sistema Operacional Linux[3];
- Editores de texto Jedit[4] e Nedit[5];
- Livros *on-line*[6].

5.2 Exemplos de uso

Nesta seção, a ferramenta de concepção de sistemas de arquivos será apresentada através de exemplos simples. Os exemplos produzidos têm o objetivo de apenas demonstrar o uso da ferramenta sem ter pretensões maiores em relação a criação de sistema de arquivos perfeitamente enquadrados aos casos propostos.

Mesmo assim, eles tentam mostrar situações nas quais um sistema de arquivos poderia ser utilizado. O primeiro exemplo é de uma câmera digital (mostrado passo a passo) e o segundo exemplo é de uma máquina injetora de plástico.

5.2.1 Exemplo 1: Câmera fotográfica digital

Descrição do problema

Uma câmera digital X possui um diagrama de blocos mostrado na figura 5.3. Neste diagrama temos um controlador central (microprocessador) que comanda um Capturador de frames (*frame grabber*) e este obtém uma imagem de um sensor eletrônico. Esta imagem possui um tamanho fixo e é armazenada numa região de memória (de tamanho fixo).

Após esta etapa, o controlador comanda o compactador para que este compacte a imagem adquirida e armazene junto com as demais fotos. Como o método de compactação possui uma taxa variável, um número variável de fotos pode ser obtido. Também devem ser armazenadas algumas informações de Opções/status da câmera (tais como número de fotos já tiradas).

Resolução

Com base na descrição do problema e utilizando os conceitos de Meta-Tipos descritos neste trabalho, pode-se mapear as memórias utilizadas neste exemplo para arquivos, conforme a tabela 5.1.

Nesta tabela, temos o tipo TIPO_BITMAP que é fixo e possui apenas a função `map()`. O uso da função `map()` será bastante útil neste caso, pois o algoritmo de compactação poderá operar sobre este arquivo simplesmente acessando posições de memória. Os demais tipos (TIPO_FOTOS e TIPO_OPCOES) foram escolhidos para descender de Meta-tipos INDEXADO e FIXO respectivamente.

[3] Linux está disponível em http://www.linux.org .
[4] Jedit está disponível em http://www.jedit.org .
[5] Nedit está disponível em http://www.nedit.org .
[6] Alguns livros, tutoriais e listas de erros estão disponíveis em http://www.java.sun.com e em http://www.oreilly.com .

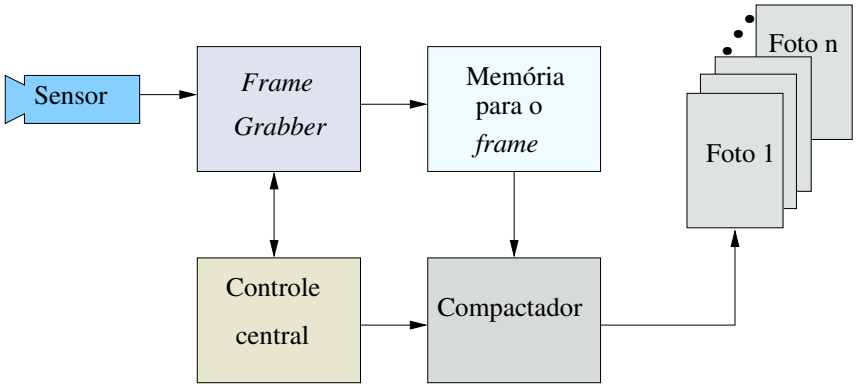


Figura 5.3: Diagrama de blocos de uma câmera digital

Descrição	Nome do tipo	Meta-Tipo base	Funções utilizadas
Memória para a imagem capturada	TIPO_BITMAP	FIXO	map()
Memória para as fotos	TIPO_FOTOS	INDEXADO	read(), write(), eof(), rewind()
Opções/status da câmera	TIPO_OPCOES	FIXO	read(), write(), rewind()

Tabela 5.1: Mapeamento de tipos de arquivos para o exemplo 1

Uso da ferramenta

A primeira tela da ferramenta está mostrada na figura 5.4. Nela, pode-se ver menus no lado direito da tela e na parte de baixo. O menu da direita, controla a janela de tipos (Área branca ao lado) e o menu abaixo, controla as ações da ferramenta.

Inicialmente deve-se criar um projeto (que irá manter informações sobre o trabalho no novo SA), para isso usa-se o menu **Projeto**>**Cria Projeto** (Figura 5.5). Feito isso, devem ser criados os tipos, conforme a tabela 5.1. Para tanto, usa-se o botão **Novo** no menu da direita (que controla a janela de tipos) e insere-se o nome do primeiro tipo, conforme a figura 5.6. Repetindo-se o processo para todos os tipos de arquivos, obtém-se os três tipos conforme ilustrado na figura 5.7 (deve ser notado que os tipos são criados por padrão descendendo do Meta-Tipo Fixo).

Após esta etapa, deve-se personalizar os tipos, isto é especificar de qual Meta-Tipo eles descendem e especificar quais funções eles suportam. Para tanto, deve-se selecionar um tipo e usar o botão **Edita**. Dentro desta opção, seleciona-se as opções para o tipo em questão e repete-se a mesma ação para todos os tipos do projeto, conforme a figura 5.8. O resultado final está ilustrado na figura 5.9.

Com os tipos criados e ajustados corretamente, pode-se passar para o ajuste das definições do SA. A janela de ajuste das definições é acessada através do botão **Definições** na parte inferior da tela. Pressionado este botão, obtém-se a janela ilustrada na figura 5.10, onde devem ser selecionadas as características do novo SA, tais como tamanho máximo dos arquivos, tamanho de bloco, algoritmos utilizados (podem ser deixados no modo automático), configurações do compilador, etc.

Deve se notar que nas definições do sistema de arquivos, existem campos tais como estratégia de

alocação de memória e algoritmo de *lookup* que podem ser selecionados pelo usuário da ferramenta ou podem ser deixados no modo automático. Neste modo, a ferramenta irá verificar os demais tipos de arquivos criados pelo usuário, o número de arquivos e automaticamente escolherá os melhores algoritmos.

A decisão do algoritmo de *lookup* é feita com base nesse número de arquivos. Para um sistema de arquivos com mais de 10 arquivos e com uma grande quantidade de memória, ela irá selecionar o algoritmo de *red-black trees* (pois esse método é mais eficiente para muitos arquivos, mas consome mais memória). Já para poucos arquivos ou quando não existe memória em abundância o método de pesquisa seqüencial será utilizado.

A seleção do método de gerenciamento de memória irá depender dos outros tipos de arquivos. Se o usuário da ferramenta está utilizando tipos baseados somente no Meta-Tipo FIXO ou CIRCULAR então a ferramenta usará somente partições fixas, se os tipos são baseados somente no Meta-Tipo INDEXADO, então será utilizada alocação indexada. No último caso, em que existe uma combinação deles, o método utilizado será MISTO, isto é suportará ambos.

Também existem campos onde são especificadas a localização dos fontes em C do sistema de arquivos e o diretório onde a documentação será produzida, os demais campos são referentes as definições que já apareceram no capítulo anterior sobre o sistema de arquivos desenvolvido.

Finalmente, grava-se o projeto (usando o menu **Projeto**>**Grava Projeto**) e usa-se o botão **Código** para gerar o código do SA. Uma janela de relatório será apresentada, informando quantas funções foram inseridas, quais são seus nomes, etc. Este relatório é mostrado na figura 5.11 e pode ser simples ou completo (dependendo se a opção **Gera relatório detalhado estiver marcada**).

Figura 5.4: Tela principal da ferramenta

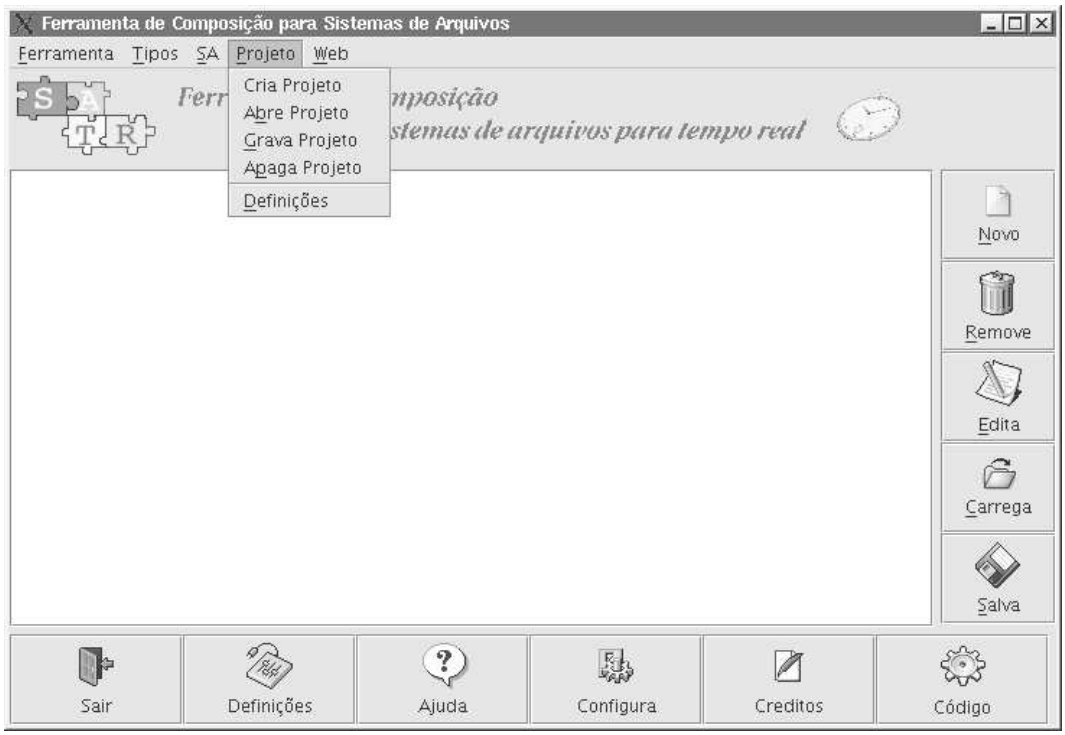


Figura 5.5: Menu de Projeto

Figura 5.6: Adição de um tipo de arquivo

Figura 5.7: Depois de adicionar os três tipos

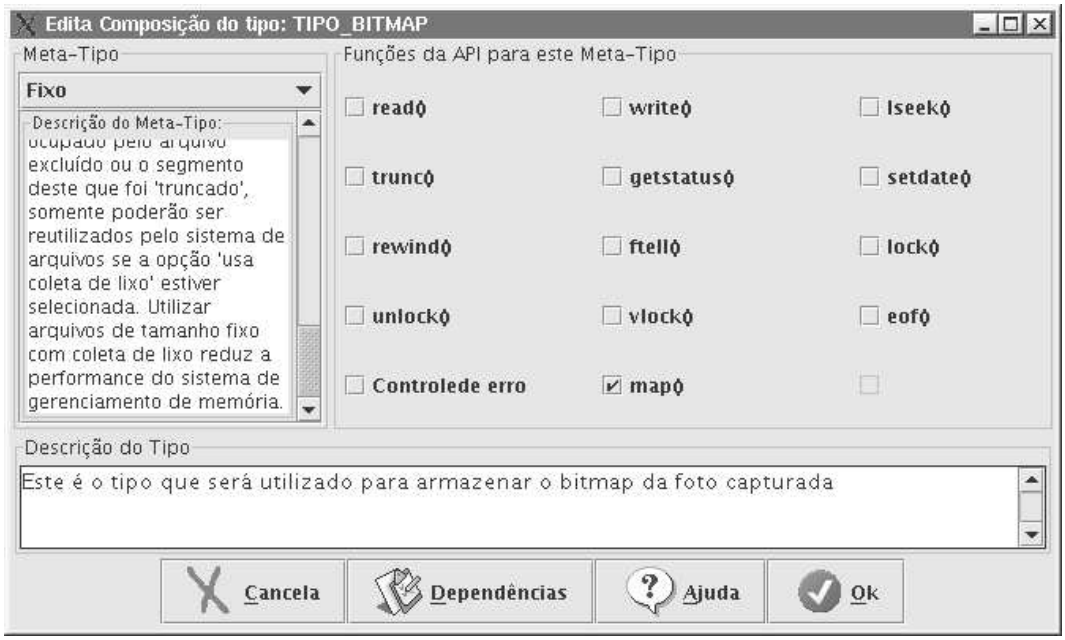


Figura 5.8: Editando o tipo TIPO_BITMAP

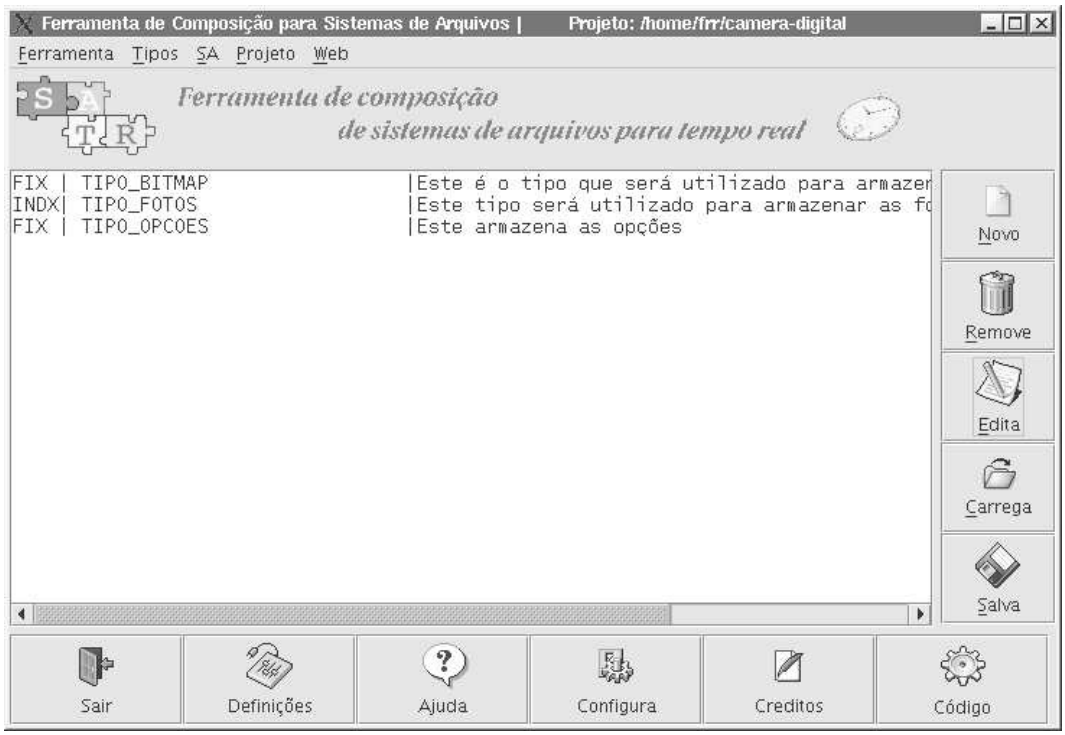


Figura 5.9: Depois de editar os três tipos

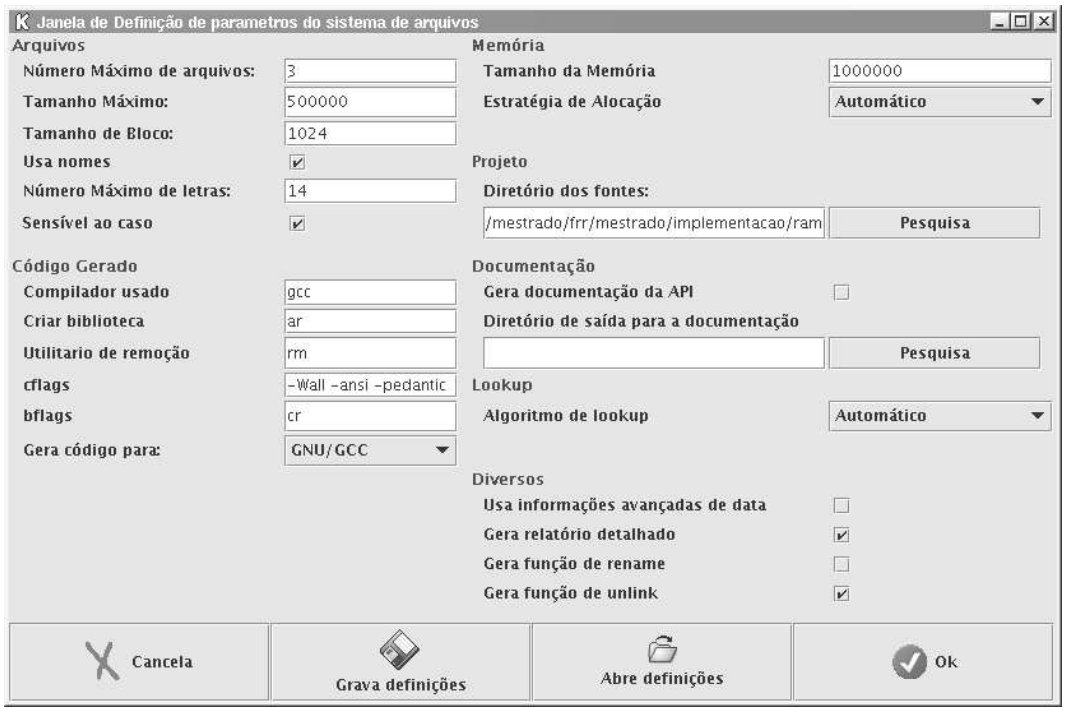


Figura 5.10: Cria as definições do SA

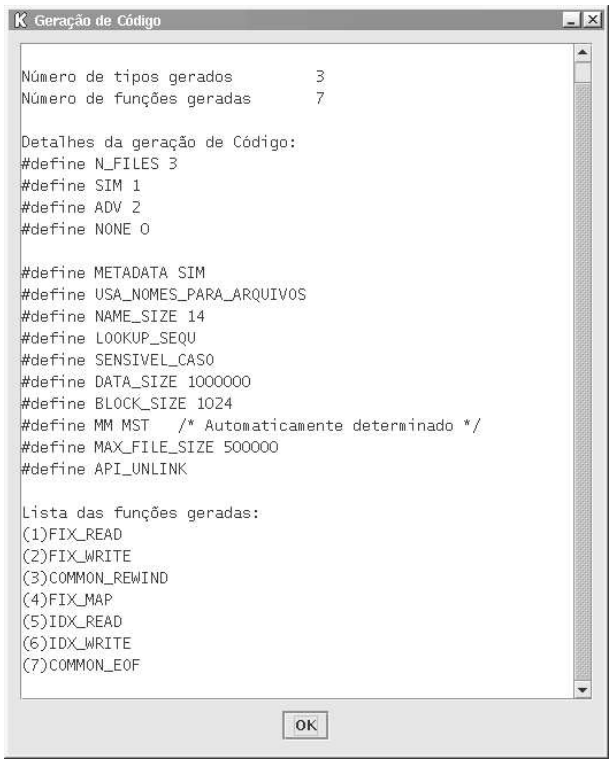


Figura 5.11: Relatório (detalhado) do SA gerado

5.2.2 Exemplo 2: Máquina injetora de plástico

Descrição do problema.

Criar um sistema de arquivos que será utilizado numa aplicação de controle de uma injetora plástica. O dispositivo em questão é parametrizável, isto é, ele mantém um programa de controle (mantido sempre fixo) que é responsável pelo funcionamento da máquina e este aceita parâmetros, tais como dimensões da peça, número de elementos a produzir, etc. Vários conjuntos de parâmetros podem existir e podem ser adicionados novos. A máquina também deve manter um histórico das últimas "n" operações.

Resolução.

Com esta descrição pode-se esquematizar o armazenamento de informações conforme a figura 5.12 e mapeá-los para arquivos como na tabela 5.2. O uso da ferramenta será exatamente da maneira que foi feito no exercício anterior, mas com tipos diferentes.

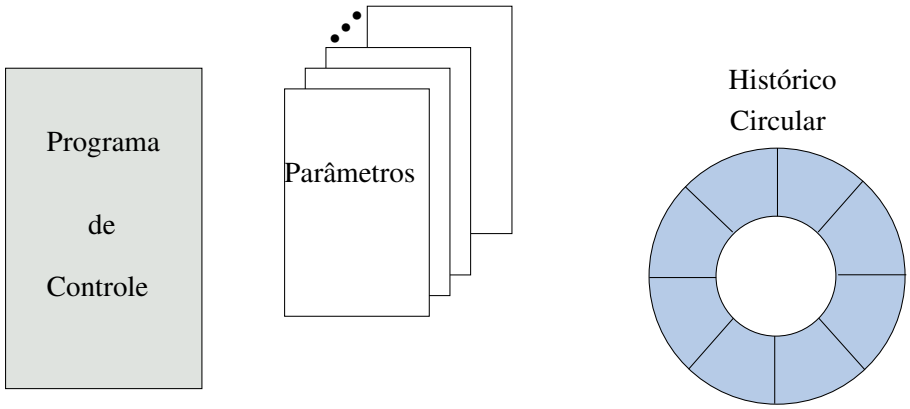


Figura 5.12: Exemplo da injetora de plástico

Descrição	Nome do tipo	Meta-Tipo base	Funções utilizadas
Memória para o programa	TIPO_PROGRAMA	FIXO	map()
Memória para os parâmetros	TIPO_PARAM	INDEXADO	read(), write(), lseek(), ftell()
Histórico	TIPO_HISTORICO	CIRCULAR	read(),write()

Tabela 5.2: Mapeamento de tipos de arquivos para o exemplo 2

5.3 O projeto da ferramenta

Nesta seção serão mostrados alguns diagramas que explicam o projeto da ferramenta visual de concepção de sistemas de arquivos.

Os diagramas serão mostrados sob duas visões, quanto a especificação do sistema de arquivos e quanto a geração de código. Inicialmente será mostrado o diagrama de casos de uso para a especificação, no qual o usuário interage com a ferramenta especificando um sistema de arquivos (figura 5.13)

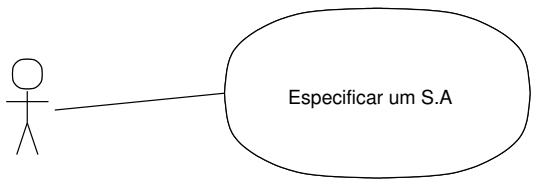


Figura 5.13: Diagrama de casos de uso - especificação do SA

Esse é um diagrama simples que não informa nada sobre as atividades envolvidas na especificação de um sistema de arquivos. Para isso, utiliza-se o diagrama de atividades mostrado na figura 5.14.

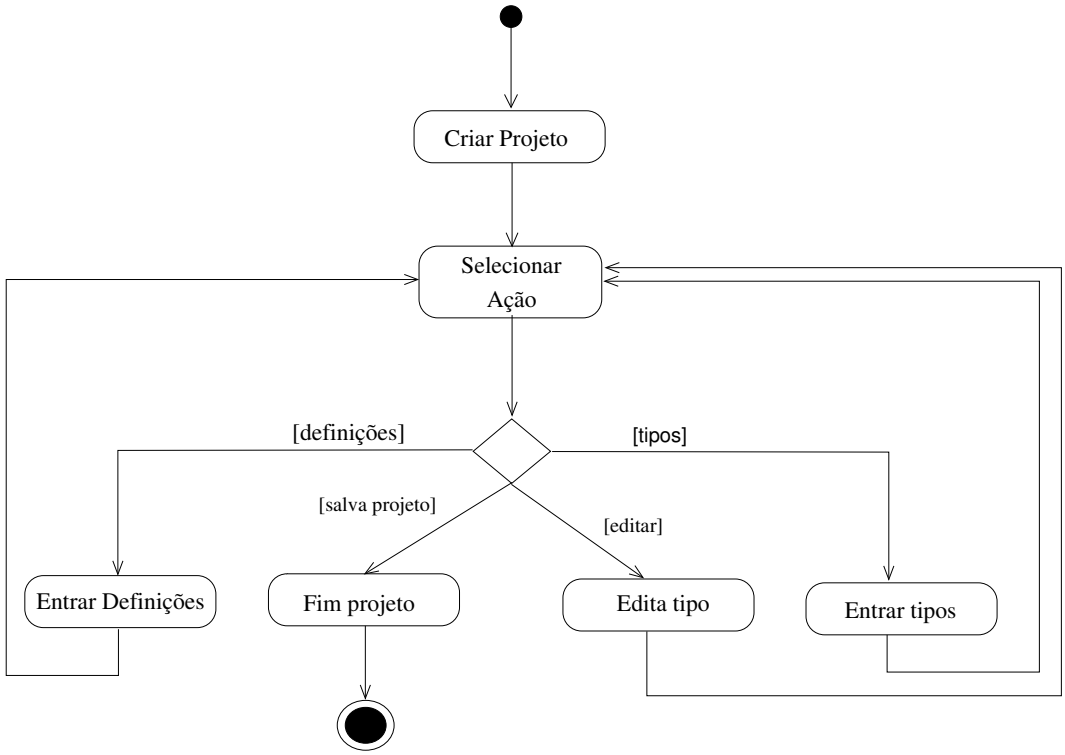


Figura 5.14: Diagrama de atividades - especificação do SA

Este diagrama ilustra basicamente os tipos de atividades possíveis: criação de tipos de arquivos, definições de características do sistema de arquivos, edição de tipos ou gravar projeto.

Na criação de tipos, o usuário insere todos os tipos de arquivos que pretende utilizar no seu projeto, na edição ele pode alterar as funções que estão disponíveis para cada tipo de arquivo e ajustar o meta-tipo no qual ele é baseado. Com as definições ele ajusta as propriedades do sistema de arquivos e gravar permite gravar o projeto.

A seguir, na figura 5.15 temos um diagrama com as classes envolvidas na especificação de um sistema de arquivos.

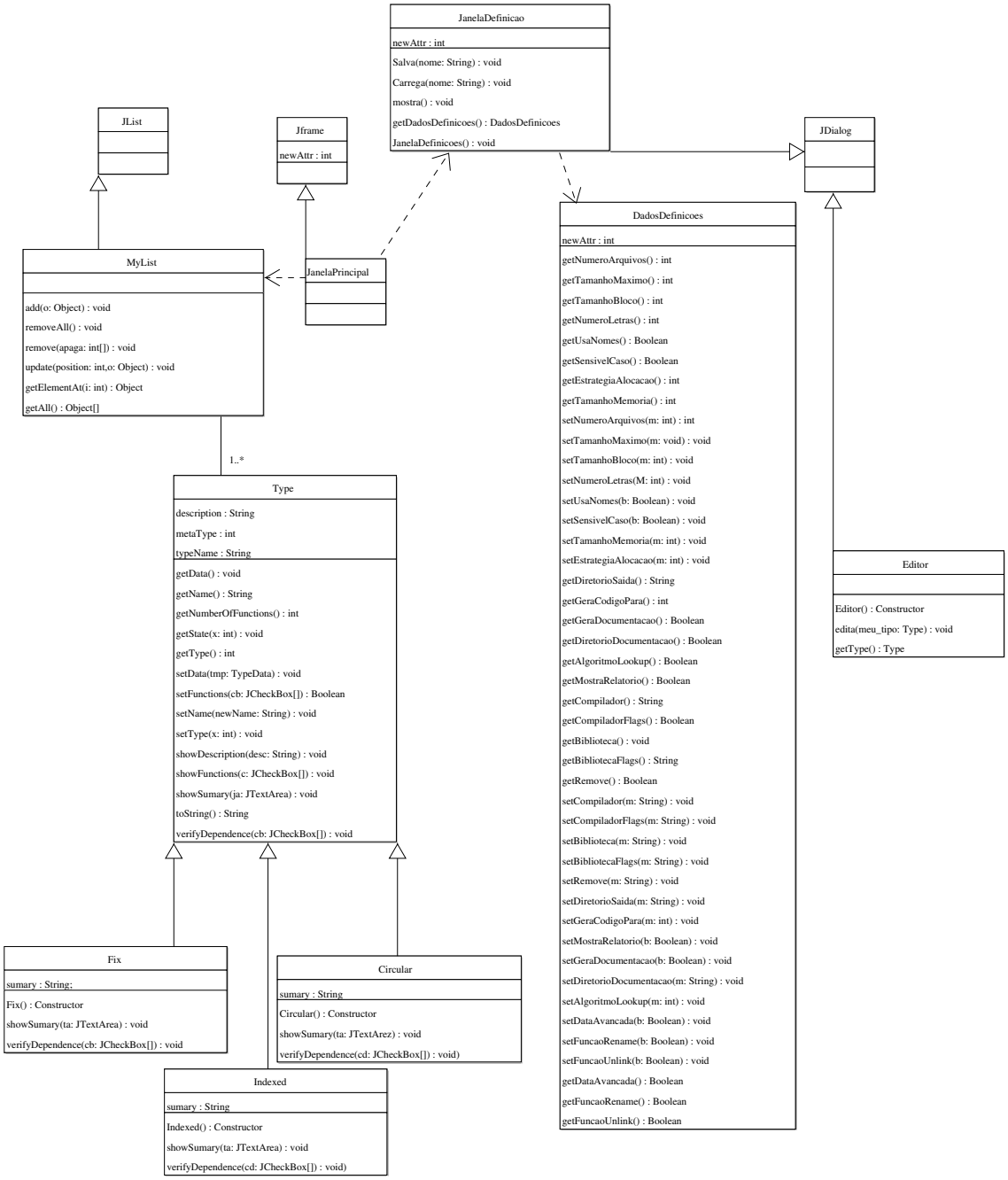


Figura 5.15: Diagrama de classes - especificação do SA

Como último diagrama desta etapa, temos um diagrama de seqüência ilustrado na figura 5.16 que mostra os eventos envolvidos na criação de um tipo de arquivo, edição e ajuste de definições.

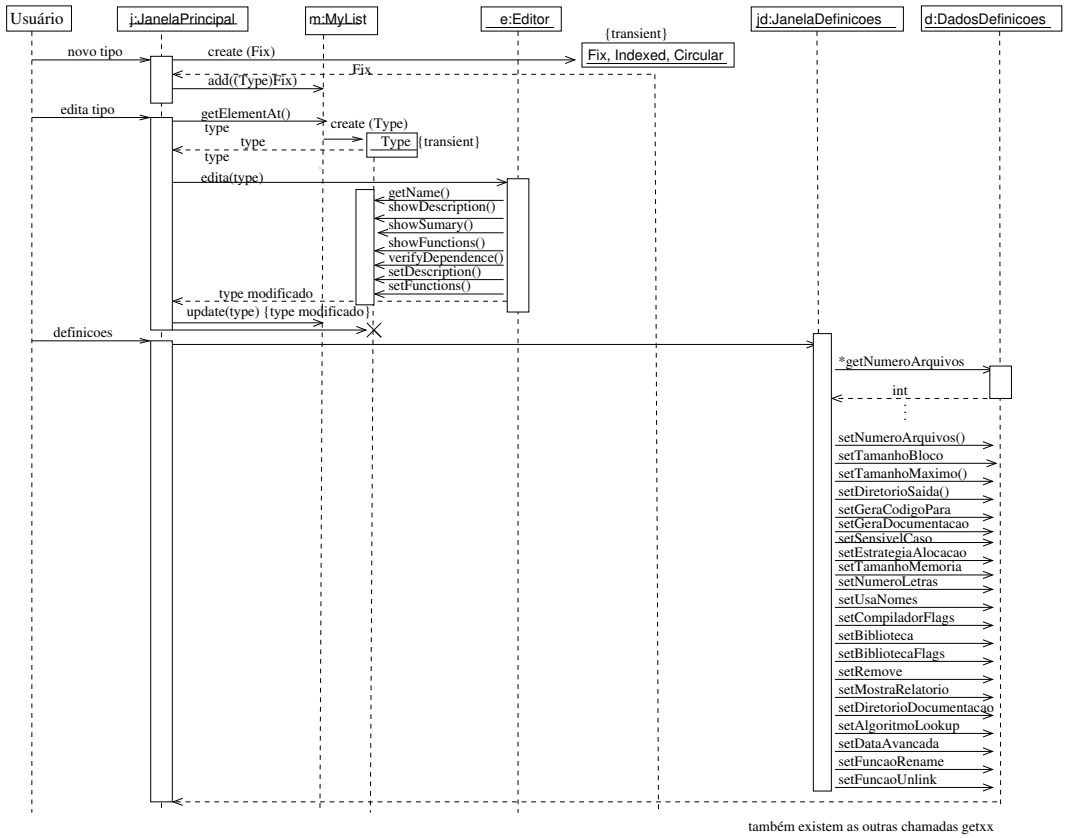


Figura 5.16: Diagrama de seqüência - especificação do SA

Na segunda visão é mostrada a geração do sistema de arquivos. Inicialmente um diagrama de casos de uso que ilustra como o usuário interage com a ferramenta (figura 5.17).

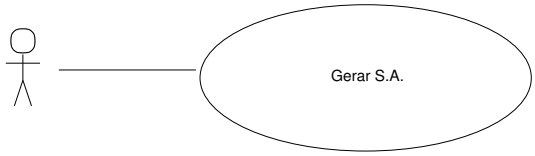


Figura 5.17: Diagrama de casos de uso - geração do SA

Após, um diagrama das atividades envolvidas na geração do SA, sistematizando as ações envolvidas. Assim, temos a criação de um projeto e seleção da opção geração de código, mostrados na figura 5.18.

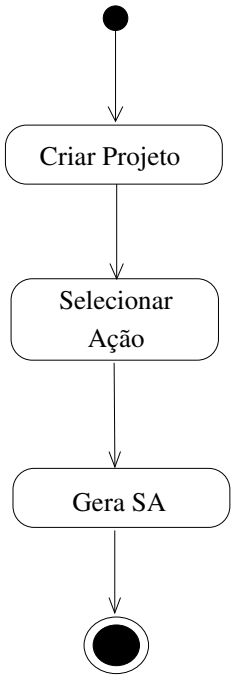


Figura 5.18: Diagrama de atividades - geração do SA

No diagrama de classes (figura 5.19), temos as principais classes envolvidas: Type que mantém as informações básicas sobre os tipos, MyList que apresenta uma lista onde os tipos ficam armazenados, CodeManager que é o responsável pela geração de código que seja independente dos tipos de arquivos. Ele se utiliza de alguns objetos (CodigoCircular, CodigoFixo e CodigoIndexado) para obter informações de como gerar código para cada um dos tipos de arquivos criados.

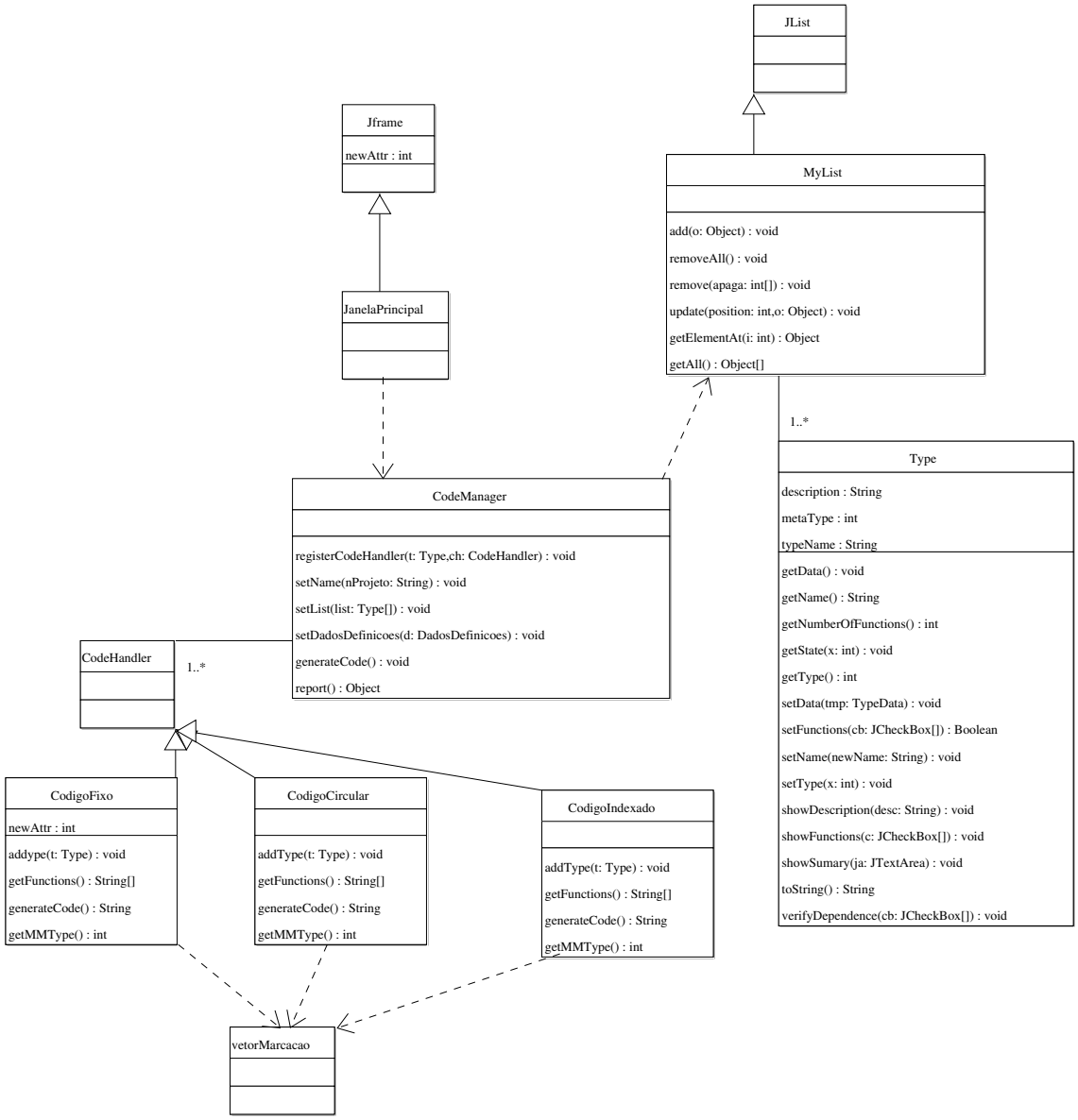


Figura 5.19: Diagrama de classes - geração do SA

Como último diagrama temos um diagrama de seqüência com os eventos envolvidos na criação do SA (figura 5.20).

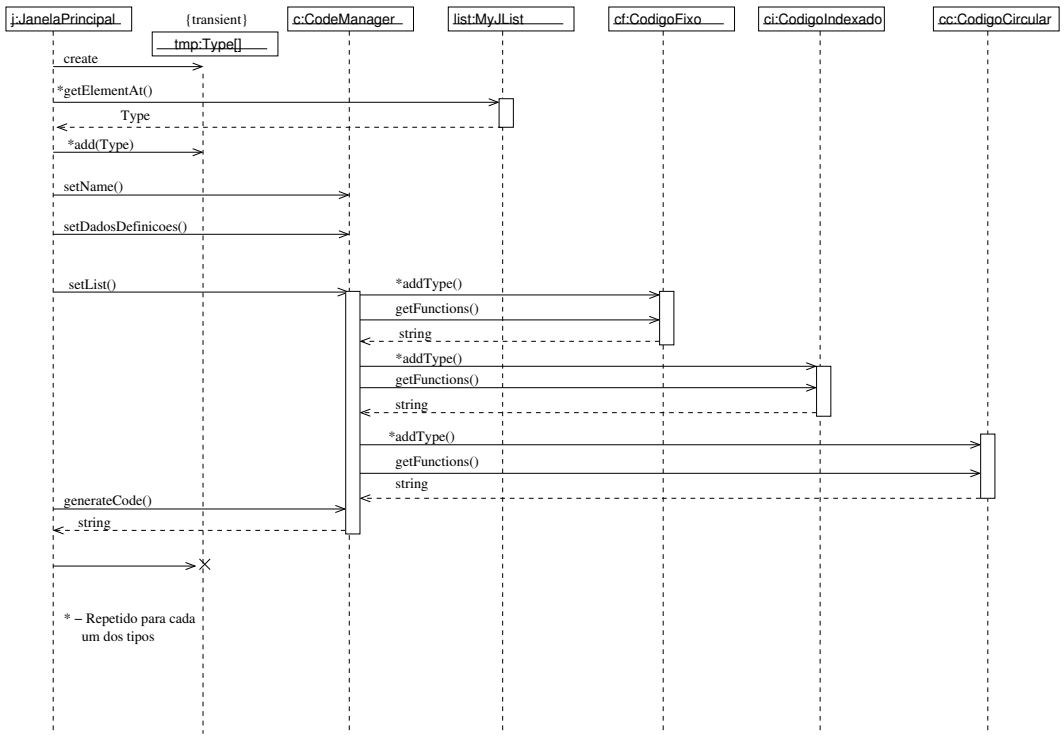


Figura 5.20: Diagrama de seqüência - geração do SA

5.4 Conclusão

Ferramentas de desenvolvimento baseadas na seleção de componentes já são bastante populares. Mesmo fazendo um bom trabalho na escolha dos componentes a utilizar, essas ferramentas possuem problemas quanto a integração desses componentes. O problema existe quando se quer integrar num projeto componentes que são mutuamente exclusivos ou que possuem restrições de uso e requisitos não fornecidos. Na maioria dos casos, o usuário fica com a responsabilidade de conhecer os componentes a fundo e saber quando um dado componente pode ser combinado com outro dentro de um projeto.

Neste trabalho, foi despendido esforço para prover a ferramenta com a capacidade de automaticamente fazer a melhor escolha de quais algoritmos utilizar (para gerenciamento de memória e *lookup*) baseado nas seleções de componentes feitos pelo usuário. Também é possível verificar em tempo de montagem do sistema de arquivos se um certo tipo de arquivo possui restrições não satisfeitas.

A disponibilidade dessa ferramenta representa uma redução no esforço de desenvolvimento da aplicação e, portanto, do seu custo. Em muitas oportunidades o projetista poderá em questão de minutos gerar um sistema de arquivos sob medida, com otimização no consumo de memória e utilizando algoritmos com tempo de pior caso controlado. Fazer o mesmo desenvolvendo diretamente sobre o código C do SAA demandaria um esforço e um custo maior.

Capítulo 6

Experiências e resultados

Neste capítulo são apresentados os resultados obtidos de experiências realizadas com a ferramenta de composição e sobre sistemas de arquivos gerados por ela.

Além disso, serão apresentados os resultados de uma experiência na qual dois sistemas de arquivos são criados para o mesmo projeto, um deles será criado sem a preocupação do espaço ocupado pelo sistema de arquivos. O outro será composto, tendo em mente uma aplicação específica e usando somente as funções necessárias.

6.1 Testes com a ferramenta

O exemplo ou cenário de uso escolhido é o de projetar um sistema de arquivos para uma câmera fotográfica digital (citado na seção 5.2.1 do capítulo 5). Naquele ítem, foram identificados os tipos de arquivos necessários para compor o SA e estes foram criados utilizando a ferramenta, resultando em:

- arquivo de **Makefile** para automatizar o processo de compilação;
- arquivo **camera.c**[1] com a associação das funções membro e inicialização da estrutura de funções;
- arquivo **camera.h** com as definições do SA e de nomes de tipos.

O processo de composição transcorreu corretamente (criação do projeto, criação dos tipos, edição dos tipos, edição das definições) e, após isso, a geração de código. A função de geração de código apresenta um relatório simples ou completo das ações tomadas pela ferramenta. Neste momento, a atenção será dada às listagens das funções inseridas pela ferramenta e mostradas no relatório completo gerado.

[1] Os nomes de arquivos **camera.c** e **camera.h** são gerados baseados no nome do arquivo de projeto, neste caso **camera.project**.

Pela tabela 5.1 verificamos que existem 8 funções, considerando todos os tipos criados pelo usuário. Mas os dados do relatório informam que foram criados 3 tipos de arquivos e inseridas 7 funções. Para compreender melhor essa situação, temos um diagrama (na figura 6.1) que mostra os tipos de arquivos e as funções que cada um desses utiliza.

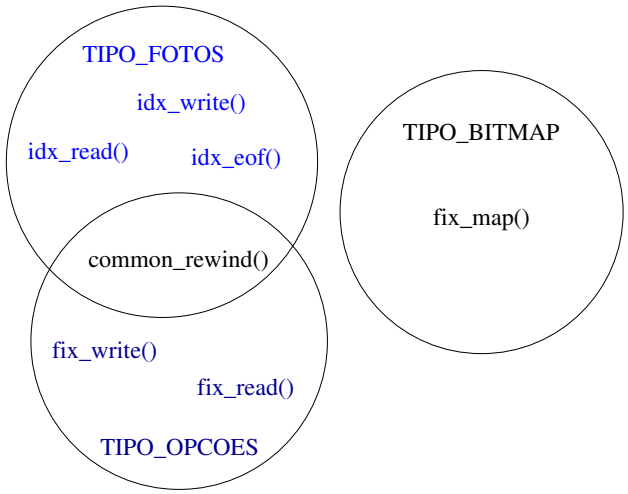


Figura 6.1: Diagrama das implementações das funções utilizadas

Neste diagrama, temos os 3 tipos criados, 2 desses derivando do mesmo Meta-tipo. Podemos ver que existe uma função que o tipo TIPO_FOTOS compartilha com o TIPO_OPCOES (common_rewind())[2].

Nos casos como este em que existem funções que são utilizadas por vários tipos, somente uma cópia do código da função é inserida. Se por acaso mais um tipo fosse criado e se este tipo utilizasse funções que já estão presentes nos demais tipos, nenhum código extra seria adicionado ao SA.

Com essa idéia em mente, podemos adicionar um tipo, por exemplo TIPO_AUXILIAR que é baseado no meta-tipo fixo e possui as funções read , write , rewind e map (funções já presentes nos demais tipos). Gerando o código desse novo sistema de arquivos, verifica-se pelo relatório (gerado da mesma maneira que ilustrado na seção 5.2.1 do capítulo 5) que o mesmo número de funções foi gerado, como seria esperado.

Neste caso, somente será adicionada uma linha no arquivo de associação de funções membro (arquivo **camera.c**) e uma linha de definição de nome no arquivo **camera.h**.

6.1.1 Definições do SA

Até agora, os sistemas de arquivos gerados utilizaram algumas definições automáticas que estão pré-ajustadas na ferramenta de composição. As definições automáticas referem-se aos algoritmos de *lookup* empregados e a estratégia de gerenciamento de memória para os arquivos.

[2]Os nomes das funções mostrados no diagrama referem-se as funções implementadas, de acordo com a figura 4.16.

A ferramenta permite que seja selecionada a estratégia de gerenciamento de memória entre as seguintes :

- Partições fixas;
- Alocação Indexada;
- Alocação Mista;
- Modo Automático.

Conforme já mencionado no capítulo 5, o modo automático irá escolher, baseado nos tipos de arquivos do usuário, qual o melhor algoritmo de alocação de memória a utilizar, isto é, o algoritmo mais eficiente em termos de ocupação de memória que provê a funcionalidade necessária.

O diagrama ilustrado na figura 6.2 mostra como os algoritmos de alocação se apresentam em relação a sua generalidade. Podemos ver que o método de partições fixas e o método de alocação indexada são isolados, isto é, o método (ou estratégia) de alocação fixa irá funcionar somente com os tipos de arquivos baseados nos meta-tipos Fixo ou Circular. A estratégia de alocação indexada irá funcionar somente com os tipos de arquivos baseados no meta-tipo indexado. Já a alocação mista, é mais geral e mais complexa, tratando arquivos baseados em todos os 3 meta-tipos.

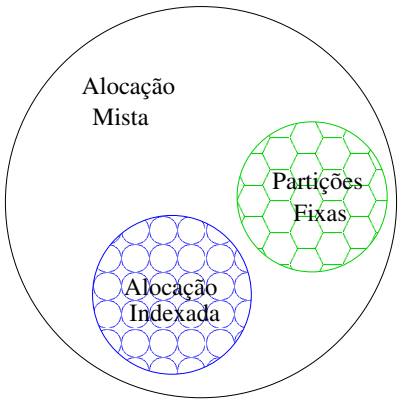


Figura 6.2: Diagrama das estratégias de alocação de memória

Para a aplicação exemplo, são utilizados os tipos baseados no meta-tipo indexado (TIPO_FOTOS) e baseados no meta-tipo fixo (TIPO_BITMAP e TIPO_OPCOES). Neste caso, a estratégia de alocação de memória deve ser ajustada para alocação mista ou ajustada para o modo automático, pois já que os tipos de arquivos criados necessitam de alocação indexada e de partições fixas, necessita-se de uma estratégia que possua ambos. Caso esta seja ajustada para partições fixas somente ou para alocação indexada somente, um erro em tempo de compilação irá ocorrer causado pela inexistência de código necessário para gerenciar a memória de tipos de arquivos baseados em um dado meta-tipo.

Um usuário experiente, poderia ter esse problema em mente quando da criação dos tipos, fazendo que todos eles possam ser utilizados com um algoritmo de gerenciamento de memória simples (partições fixas ou alocação indexada). Fazendo isso se reduz o gasto de memória.

A outra definição é em relação a operação de *lookup*. As seleções possíveis são árvores *red-black*, pesquisa seqüencial ou modo automático. Neste caso o modo automático irá escolher entre um ou outro, baseado no número máximo de arquivos e do tamanho da memória, conforme descrito no capítulo 5.

Caso o usuário ajuste um modo que não seria ideal para o seu projeto, este será compilado corretamente. Os problemas somente existirão em relação ao desempenho da aplicação (pois os métodos possuem diferentes complexidades computacionais) e em relação ao uso de memória. Em algumas situações onde a memória é bastante reduzida ou o desempenho deve ser melhorado, uma escolha sensata do algoritmo utilizado deve ser feita.

Ainda nas definições, temos algumas funções que podem ser selecionadas e disponibilizadas para todos os tipos do SA. Essas funções são:

- informações avançadas de data;
- função de *rename*;
- função de *unlink*.

Se selecionarmos o uso de informações avançadas para data, a estrutura que armazena a data será composta por mais campos, tais como data de última escrita e data de última leitura. Esta seleção também faz que a cada chamada de rotina sobre um arquivo, essas informações sejam atualizadas, o que pode ser um desperdício de tempo em alguns casos. Uma forma mais simples é usar somente informações simples de data (deixando não marcado o ítem informações avançadas), na qual a data é composta apenas pela data de criação do arquivo.

A seleção de uso da função *rename*, permite que o nome associado a um arquivo seja mudado. Essa função só estará disponível no caso em que os arquivos sejam identificados por nomes e não números. Já a seleção de uma função de *unlink* permite que se apague um arquivo, liberando a memória ocupada. Todas essas seleções irão aparecer no relatório de geração de código, se este foi produzido com a opção **Gera relatório detalhado** marcada (conforme ilustrado na figura 5.10).

6.1.2 Experiência - economia de memória

Para ilustrar a potencial economia de memória que pode ser obtida com o uso da ferramenta de composição de sistemas de arquivos, criou-se um outro sistema de arquivos para o mesmo projeto da câmera fotográfica digital, considerando-se os mesmos 3 tipos iniciais.

Nesse sistema de arquivos cada tipo foi construído com a totalidade das funções e as definições do sistema de arquivos foram relaxadas, isto é, não houve preocupação em economia de memória para sistema de arquivos. No total, foram inseridas 17 funções no sistema de arquivos novo (sistema de arquivos **[B]**). Na seção 6.3 será feita uma comparação de tamanho entre o código compilado dos dois sistemas resultantes.

6.2 Testes sobre o sistema de arquivos

Nesta seção, o exemplo do sistema de arquivos composto para a câmera fotográfica digital será compilado usando o *make*[3] e o *gcc*[4] no sistema operacional Linux[5] e será escrita a aplicação principal que fará chamadas às rotinas do SA. Antes de examinar detalhadamente a codificação desta aplicação, devem ser explicitadas algumas diferenças entre a compilação de uma aplicação em um sistema convencional (como a que será feita nesta seção) e a compilação para uma plataforma embutida com um *hardware* específico.

6.2.1 Código de inicialização

Para automatizar e acelerar o desenvolvimento de *software*, os compiladores modernos executam um conjunto de ações sobre um arquivo fonte `.c`. Automaticamente, eles produzem arquivos objeto baseados nos arquivos fontes e executam um *linker* sobre esses para produzir um arquivo executável. Durante a etapa de link-edição, as bibliotecas e o código de inicialização também são unidos para constituir o arquivo executável.

Esse código de inicialização (*startup code*), mostrado no código 6.3, é sempre executado antes da função `main()` e tem a função de:

- desabilitar as interrupções;
- copiar dados necessários da *ROM* para a *RAM*;
- preencher com zeros a memória não inicializada;
- alocar memória para a pilha e inicializar o ponteiro de pilha;
- alocar e inicializar a memória *heap*;
- habilitar interrupções;
- chamar `main`.

Por padrão, um compilador para um sistema convencional utiliza com arquivo objeto (geralmente chamado `startup.o` ou `crt0.o` [6]) pré-concebido para a arquitetura alvo.

[3] GNU Make versão 3.79.1
[4] GCC versão 3.2
[5] Linux Mandrake 9.0- kernel 2.4.19-16mdk
[6] Abreviatura de C runtime

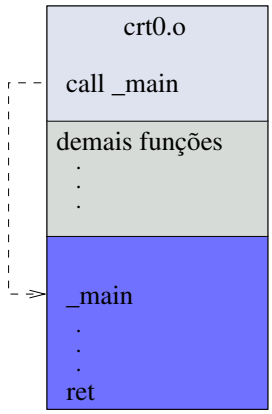


Figura 6.3: Código de inicialização

Neste caso, o *gcc* irá automaticamente adicionar o seu arquivo de inicialização que faz alguns ajustes para determinada versão do Linux executando sobre uma certa plataforma de *hardware*, como por exemplo um processador Intel compatível.

Diferentemente de um sistema convencional, quando se produz *software* para um sistema embutido, não se dispõe de um arquivo de inicialização pré-concebido. Cada dispositivo embutido é diferente e o código de inicialização geralmente leva em consideração algumas dessas diferenças. Neste caso, o programador deve implementar o seu próprio arquivo de inicialização e manualmente chamar o *linker* para unir todos os arquivos objeto (Berger, 2002; Barr, 1999).

6.2.2 Implementação de semáforos binários

No capítulo 3 foi comentado que o sistema de arquivos é um sistema que permite que várias linhas de execução estejam em andamento num dado instante e que este utiliza semáforos binários (ou mutex) para proteger áreas de código contra acesso concorrente. Internamente, o sistema de arquivos utiliza funções tais como `P()` e `V()` para este fim.

Uma fonte de informações importantes aparece no apêndice B, com o diagrama de invocação de funções e lista de semáforos utilizados em cada função. Com esses diagramas e os comentários existentes, pode-se conhecer o comportamento das funções quanto aos bloqueios dos semáforos binários e determinar que funções utilizar na programação da aplicação.

A implementação de semáforos não faz parte do módulo do sistema de arquivos e desse modo não foi incluída nele. O código responsável pelos semáforos deve ser implementado pelo programador, mantendo sempre a mesma interface padrão esperada pelo sistema de arquivos. Para facilitar os testes no sistema operacional Linux, a implementação de semáforos foi feita utilizando-se a biblioteca *pthreads*[7]. Com essa biblioteca, as rotinas `P()` e `V()` foram implementadas facilmente.

[7] Uma biblioteca que possui rotinas para criação e manipulação de *threads* e semáforos.

6.2.3 Bibliotecas estáticas x bibliotecas compartilhadas

Como última diferença importante para a compilação de *software* para sistemas embutidos temos a questão das bibliotecas estáticas e compartilhadas. Uma biblioteca estática é uma biblioteca na qual o seu código será inteiramente adicionado junto com os demais objetos para formar um arquivo executável. Se imaginarmos que para cada programa que faz chamadas a uma biblioteca de rotinas incluirá internamente uma biblioteca, teremos um grande desperdício de espaço.

Uma forma de minimizar esta situação, é utilizar bibliotecas compartilhadas. Neste caso, o compilador não irá inserir o código relativo a biblioteca no arquivo executável, ele irá simplesmente colocar uma referência ao nome da biblioteca. Quando o programa for executado, ele irá precisar do código dessa biblioteca e irá pedir ao sistema que a encontre. Após o sistema carregar essa biblioteca para a memória principal, o programa executará chamadas às rotinas dessa biblioteca como no caso das bibliotecas estáticas, mas sem incorrer no *overhead* de espaço no arquivo[8].

Em um sistema convencional, o compilador C sempre que possível irá utilizar bibliotecas compartilhadas para evitar o desperdício de memória. Infelizmente, para um sistema embutido sem muitos recursos de *software* é quase uma unanimidade que a compilação deve ser sempre feita com bibliotecas estáticas, pois nestes sistemas geralmente não existe suporte para bibliotecas compartilhadas (Barr, 1999).

6.3 Aplicação final

Terminada a etapa de uso da ferramenta, utiliza-se o comando *make* para compilar os sistemas de arquivos produzidos. Este comando irá invocar o compilador nativo do Linux *gcc* para produzir os arquivos objeto para cada um dos componentes do sistema de arquivos. Após o termino dessa compilação, ele irá invocar o aplicativo *ar* para criar uma biblioteca com todos estes objetos.

No código 13 temos a biblioteca resultante da compilação do sistema de arquivos específico para este projeto e no código 14 temos a biblioteca para o sistema de arquivos **[B]**, conforme experiência comentada.

No primeiro, a biblioteca foi gerada com a visão mais focada no problema da economia de espaço, incluindo somente as funções que serão necessárias e usando as definições do sistema de arquivos para esse caso específico.

No segundo, temos a biblioteca resultante da criação dos 3 tipos de arquivos com a inclusão de todas as funções destes. Também foram utilizadas definições mais gerais para o sistema de arquivos (suporte a grande número de arquivos, nomes de arquivos com muitas letras, inclusão de funções de *rename*, *unlink* e informações avançadas para a data). Como resultado, pode-se perceber uma diferença que chega a quase 50% entre o tamanho das bibliotecas produzidas nos dois casos. Além disso, a memória usanda em tempo de execução também é minimizada.

[8]Existem outras vantagens na utilização de bibliotecas compartilhadas, mas por saírem do escopo deste trabalho não serão mostradas.

O código 15 temos a listagem dos objetos presentes no código 13.

```
tool/classes>    ls -la camera.a
-rw-------       1 frr       mestrado      19908  Jan  22 13:39  camera.a
tool/classes>
```

Código 13: Biblioteca do sistema de arquivos

```
tool/classes>    ls -la camera_b.a
-rw-------       1 frr       mestrado      29822  Jan  22 14:00  camera_b.a
tool/classes>
```

Código 14: Segunda biblioteca produzida (sistema de arquivos **[B]**)

```
tool/classes>    ar tv camera.a
rw-------    2548/203      844 Jan  22 13:39  2003 memory.obj
rw-------    2548/203     3024 Jan  22 13:39  2003 mm.obj
rw-------    2548/203      948 Jan  22 13:39  2003 time.obj
rw-------    2548/203     2188 Jan  22 13:39  2003 lookup.obj
rw-------    2548/203     4640 Jan  22 13:39  2003 function.obj
rw-------    2548/203     1108 Jan  22 13:39  2003 custom.obj
rw-------    2548/203     4744 Jan  22 13:39  2003 api.obj
rw-------    2548/203     1112 Jan  22 13:39  2003 semaphore.obj
tool/classes>
```

Código 15: Arquivos objeto presentes na biblioteca

Com o sistema de arquivos pronto, resta desenvolver a aplicação principal. Deve ser enfatizado que, neste teste, a aplicação será apenas uma simulação executando como um programa normal Linux. Muitas das funções responsáveis pela leitura dos sensores, compactação das fotos, entre outras, foram deixadas em branco e também muitas das funções existentes no sistema de arquivos que foi composto não foram utilizadas, para manter o exemplo simples e pequeno.

Um dos problemas em escrever esse tipo de *software* é a ausência de dispositivos de saída, para verificar se o comportamento da aplicação está correto. Inicialmente poderia-se pensar em incluir o arquivo `stdio.h` para ter acesso as rotinas padrão de I/O da biblioteca do C. Infelizmente essa solução não é satisfatória, pois esse arquivo irá incluir também as funções de acesso a arquivos, as quais possuem o mesmo nome que as do sistema de arquivos personalizado, resultando num erro em tempo de compilação.

Para lidar com esse problema, existem algumas possibilidades:

- trocar o nome de todas as funções do sistema de arquivos, por exemplo prefixando as rotinas com o nome `rtfs_` e re-compilando. Assim, pode-se incluir as bibliotecas de I/O. Outro ponto

problemático são defines e structs que existem nas bibliotecas de I/O e que também existem no código do sistema de arquivos, tais como a definição dos tipos size_t , ssize_t , etc;

- escrever e utilizar funções de I/O externas;

Pela primeira alternativa, fica claro que seria trabalhoso modificar o fonte do sistema de arquivos inteiro para evitar problemas de compilação, isso nos leva a segunda alternativa. Assim, foram implementadas algumas funções de I/O, que são: my_read() - para ler números do teclado, my_print() - para imprimir cadeias de caracteres e my_printint() - para imprimir números inteiros. Essas funções são suficientes para que o usuário possa verificar o funcionamento da aplicação (Código 16).

A aplicação principal é composta por duas *threads*, sendo uma para o controle do dispositivo (botões) e outra para fazer a atualização do *display*. A *thread* para controle do dispositivo possui cinco funções (obter uma foto, gravar a foto, ajustar sensor, apagar a memória e descarregar as fotos). A função de obter uma foto constitui-se de configurar o sensor e ler uma imagem de tamanho fixo. Após obter essa imagem, pode-se escolher compactar e gravar a imagem junto com as demais.

Outras funções possíveis são: Configurar o sensor, apagar a memória (eliminando todas as fotos) e descarregar as fotos (possivelmente para um computador).

```
#include  "camera.h"
#include  <pthread.h>
#include  "aplicacao.h"

void *ptr;
char buffer[BITMAP_MAX];
pthread_t thread1, thread2; /* Identificador das threads */
pthread_mutex_t foto        = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t todas_fotos = PTHREAD_MUTEX_INITIALIZER;
int arq_foto, arq_opt , arq_todas_fotos;

/* Identificadores de arquivos                    */

extern void my_print ( char *st );
extern void my_printint ( int x );
extern int my_read ( void );

void read_sensor ( void *ptr )
{
        /* Simula uma leitura de sensor           */
        int x;
        char *cptr = (char *)ptr;
        for (x =0;x < BITMAP_MAX; x++)  cptr[x]=4;
}
void set_sensor_opt ( int opt )
```

```
{

}
int read_button ( void )
{
        /* Simula leitura do painel de controle     */
        /* da camera digital                        */
        return (my_read());
}
int compress_bitmap ( void *ptr )
{
        /* Compacta a foto antes de gravar          */
        return BITMAP_MAX;
}
void mostra_foto( void *ptr )
{
        /* Mostra a foto no display                 */
}
void limpa_visor( void )
{
        /* Limpa o display                          */
}
int read_new_opt ( void )
{
        /* Obtem novos parametros de configuracao   */
        /* do sensor                                */
        return 0;
}
void bate_foto ( void *ptr , int opt)
{
        /* Captura uma foto                         */

        pthread_mutex_lock (&foto);
        set_sensor_opt(opt);
        map(arq_foto,0,ptr);
        read_sensor(ptr);
        pthread_mutex_unlock (&foto);
}
void grava_foto ( void *ptr )
{
        /* Grava uma foto                           */

        int size;
        pthread_mutex_lock (&foto);
        pthread_mutex_lock (&todas_fotos);
        /* Antes de gravar , compacta a foto        */
```

```
        size = compress_bitmap(ptr);
        write (arq_todas_fotos,ptr,size);
        pthread_mutex_unlock (&todas_fotos);
        pthread_mutex_unlock (&foto);
}
void apaga_memoria ( void )
{
        /* Apaga todas as fotos armazenadas          */

        pthread_mutex_lock ( &todas_fotos );
        close ( arq_todas_fotos );
        unlink ( "arquivo_fotos" );
        arq_todas_fotos=open("ARQUIVO_FOTOS",O_CREAT|TIPO_FOTOS);
        pthread_mutex_unlock ( &todas_fotos );
}
void descarrega_fotos ( void )
{
        int x,y;
        char mbuffer[BITMAP_MAX];

        pthread_mutex_lock (&todas_fotos);
        rewind(arq_todas_fotos);
        while (!eof(arq_todas_fotos))
        {
                x = read(arq_todas_fotos,mbuffer,sizeof (mbuffer));
                for (y = 0; y < x; y++) {
                        my_print("mbuffer[");
                        my_printint(y);
                        my_print("]=");
                        my_printint(mbuffer[y]);
                        my_print("\n");
                }
        }
        pthread_mutex_unlock (&todas_fotos);
}
void *controla_botoes ( void )
{
        int b, opt;
        ptr = (void *)&buffer;
        opt = OPT_DEFAULT;
        my_print("Tecle:");
        my_print("[0] para tirar foto");
        my_print("[1] gravar foto");
        my_print("[2] para ajustar sensor");
        my_print("[3] para apagar a memoria");
        my_print("[4] para descarregar as fotos");
```

```
        while (1)
        {
                b = read_button();
                switch (b) {
                        case TIRA_FOTO:
                                my_print(" TIRA_FOTO");
                                bate_foto(ptr,opt);
                                mostra_foto(ptr);
                                b = read_button();
                                limpa_visor();
                                if (b == GRAVA_FOTO) {
                                        my_print(" GRAVA_FOTO");
                                        grava_foto(ptr);
                                }
                                break ;
                        case AJUSTA_OPT_SENSOR:
                                my_print(" AJUSTA_OPT_SENSOR");
                                opt = read_new_opt();
                                break ;
                        case APAGA_MEMORIA:
                                my_print(" APAGA_MEMORIA");
                                apaga_memoria();
                                break ;
                        case DESCARREGA_FOTOS:
                                my_print(" DESCARREGA_ARQUIVOS");
                                descarrega_fotos();
                                break ;
                }
        }
}
void cria_arquivos ( void )
{
        /* Cria os arquivos                                    */
        arq_foto=open("FOTO_BITMAP",O_CREAT|TIPO_BITMAP,BITMAP_MAX);
        arq_opt =open("FOTO_OPT",O_CREAT|TIPO_OPCOES,OPT_SIZE);
        arq_todas_fotos = open("ARQUIVO_FOTOS",O_CREAT|TIPO_FOTOS);
}
void * atualiza_visor ( void )
{
        /* Simula a rotina de atualizacao do display  */
        while (1)
        {
        }
}
int main ( void )
```

```
{
        my_print(" Constroi o SA");
        build_rtfs();     /* Constroi  o SA  */
        my_print(" Cria os arquivos");
        cria_arquivos();  /* Cria  arquivos   */
        /* Cria  algumas  threads  para  realizar  certas  tarefas  */
        pthread_create (&thread1, 0, (void *)controla_botoes, 0);
        pthread_create (&thread2, 0, (void *)atualiza_visor, 0);
        pthread_join (thread1, 0);
        pthread_join (thread2, 0);
        return  0;
}
```

Código 16: Código fonte da aplicação

Comentários de cada função:

read_sensor É responsável por fazer uma leitura do sensor da câmera, retornando a imagem capturada. No fonte .c da aplicação, esse código foi simulado;

set_sensor_opt Ajusta opções do sensor da câmera, na aplicação essa função foi deixada em branco;

read_button Lê o painel de controle da câmera, na aplicação essa rotina foi simulada com uma leitura do teclado;

compress_bitmap Antes de uma imagem ser salva, ela deve ser compactada. Esta função é responsável por compactar uma imagem e retornar essa imagem compactada e seu tamanho. Na aplicação essa função foi deixada em branco;

mostra_foto Mostra a foto no display. Essa função não foi implementada;

limpa_visor Apaga a imagem do display. Essa função não foi implementada;

read_new_opt Função que lê as novas opções para o sensor. Essa função não foi implementada;

bate_foto É responsável por configurar o sensor e obter uma foto. Essa foto é armazenada num arquivo de tamanho fixo e que foi mapeado para uma posição de memória com a função `map()`. Utiliza uma seção crítica para proteger o acesso aos dados da imagem.

grava_foto Responsável por armazenar a foto (que estava mapeada em memória) para um arquivo indexado. Para cada foto escrita no arquivo indexado, a posição desse arquivo será incrementada. Posteriormente lendo este arquivo indexado pode-se obter todas as fotos. Utiliza seções críticas para proteger o acesso aos dados do arquivo fixo (que armazena uma foto) e o arquivo indexado (para a coleção de fotos);

apaga_memoria Elimina todas as fotos existentes do arquivo indexado. Para fazer isto, simplesmente elimina o arquivo de fotos e cria novamente;

controla_botoes Controla as ações dos botões de controle da câmera, chamando funções correspondentes a essas funções. As função são: tirar foto, gravar foto, ajustar sensor, apagar memória e descarregar fotos;

descarrega_fotos Lê todas as fotos contidas no arquivo do tipo TIPO_FOTOS e transmite para o computador. Na implementação essa função imprime os bytes do vetor que representa a imagem. Utiliza a função `rewind()` para posicionar o arquivo no início e deste ponto lê os dados até o final do arquivo (com ajuda da função `read()` e `eof()`);

cria_arquivos Responsável por criar os três tipos de arquivos utilizados.

O primeiro arquivo criado é o que armazena a foto obtida do sensor, esse arquivo possui um tamanho fixo BITMAP_MAX e somente suporta a função `map()`. O segundo arquivo é um arquivo de tamanho fixo OPT_SIZE e armazena algumas opções (que não foram implementadas nesta aplicação). Ele possui as funções `read()`, `write()` e `rewind()`;

O último arquivo é o utilizado para armazenar o conjunto de fotos, ele é baseado no meta-tipo indexado e possui as seguintes funções: `read()`, `eof()` e `rewind()`;

atualiza_visor Responsável por atualizar o visor com dados da imagem e informações de status do dispositivo. Na implementação essa função foi simulada;

main Faz chamadas à funções responsáveis pela construção do sistema de arquivos em memória, criação dos arquivos e da *threads*.

6.4 Conclusão

Neste capítulo foram feitas algumas experiências sobre a ferramenta de concepção de sistemas de arquivos, sobre a aplicação principal e também algumas considerações sobre a criação de programas para sistemas embutidos.

Desde o início, através das experiências/testes ilustrados, foi possível ver com que facilidade alterações são feitas no sistema de arquivos e verificar muitas das vantagens da utilização da ferramenta de composição, tais como criação de sistemas de arquivos conforme a necessidade e desenvolvimento rápido destes.

Capítulo 7

Conclusões e perspectivas futuras

Neste trabalho, apresentou-se um ambiente de concepção de sistemas de arquivos voltados para aplicações embutidas de tempo real. A principal motivação deste trabalho, foi a carência de ferramentas de *software* automatizadas para construir sistemas de arquivos sob medida para uma aplicação e também o grande benefício que uma ferramenta destas poderia produzir.

Para tanto, foi feito um estudo das necessidades e restrições das aplicações geralmente encontradas em sistemas embutidos. Neste estudo, também foram feitos levantamentos de sistemas de arquivos existentes, formas de implementação destes, complexidade de algoritmos, tempo real e aspectos de programação envolvidos.

Como resultado, construiu-se um sistema de arquivos composto por diversos módulos altamente configuráveis. Esse sistema de arquivos (chamado sistema de arquivos abstrato) é utilizado como base para construção de sistemas de arquivos sob medida para uma determinada aplicação (tal como um telefone celular, um *GPS*, etc). Como vantagens desse sistema de arquivos, temos:

- Facilita o desenvolvimento rápido de sistemas de arquivos personalizados. A criação de sistemas de arquivos passa a ser uma questão se escolher quais os módulos serão necessários, todo o código já está implementado;
- O código gerado utiliza algoritmos com complexidade de pior caso conveniente e explícita, desta forma, é propício a ser usado em sistemas de tempo real;
- O tamanho do código gerado varia conforme a necessidade. A idéia de criar sistemas de arquivos personalizados permite que o sistema de arquivos em questão, seja composto somente com os módulos necessários para determinada aplicação alvo, sem incorrer em *overheads* desnecessários inserindo código que nunca será utilizado. A economia de memória que vem disso, é um ponto importante para dispositivos com restrições de memória, tais como sistemas embutidos;
- Suporte a várias *threads*. O código do sistema de arquivos permite que várias *threads* estejam acessando a mesma função no mesmo momento. Ele utiliza semáforos binários para controlar o acesso a seções críticas de código;

- Minimiza as inversões de prioridade, pois os algoritmos são criados tendo em mente que as seções críticas existentes no código devem ser pequenas, minimizando o tempo de bloqueio nos semáforos e maximizando a concorrência das *threads*.

Em determinado momento deste trabalho, percebeu-se que a etapa de composição de sistemas de arquivos era constituída de vários passos que poderiam ser automatizados ou guiados, visando um aumento de produtividade.

Com essa idéia em mente, foi criada uma ferramenta visual de concepção de sistemas de arquivos. Esta ferramenta, funciona como um *front-end* para a composição de sistemas de arquivos. Com esta, pode-se em questão de minutos criar um sistema de arquivos personalizado para uma aplicação, com as mesmas vantagens em relação ao código gerado.

Todas essas vantagens reunidas neste ambiente de concepção representam um grande auxílio no desenvolvimento de *software* para dispositivos embutidos que necessitem de sistemas de arquivos. Além disso, representam uma vantagem competitiva para empresas da área, reduzindo o tempo de desenvolvimento de projetos, o tempo que um produto leva até chegar o mercado e conseqüentemente o custo final deste. A vantagem se torna mais aparente, quando leva-se em consideração a inexistência de ferramentas similares, deixando nas mãos do programador a tarefa de desenvolver todo o sistema de arquivos para uma aplicação.

Baseado em observações das mudanças no mercado ocorridas nos últimos anos, parece cada vez mais claro que as ferramentas automatizadas vieram para ficar.

Desta forma, sugere-se como continuação deste trabalho um ambiente de concepção mais geral, no qual o objeto de concepção não seja somente o sistema de arquivos mas um sistema operacional completo. Uma idéia semelhante, embora aplicada num outro contexto é projeto *OSKit*[1] que está sendo desenvolvido na universidade de UTA nos EUA. O mesmo grupo que desenvolve este projeto, também desenvolve o *Knit*[2] uma linguagem de definição de componentes de *software* para auxiliar na detecção de erros durante a integração de componentes, redução do *overhead* e promoção da reusabilidade de código.

Essa tendência mundial mostra que esta é uma área promissora para pesquisas futuras.

[1] http://www.cs.utah.edu/flux/oskit/
[2] http://www.cs.utah.edu/flux/knit/

Apêndice A

Interface de programação completa em C

---

nome: **build_rtfs**

sintaxe:

void **build_rtfs** (*void*);

descrição:

A função **build_rtfs** prepara o Sistema de Arquivos para uso. Ela deve ser chamada somente uma vez na aplicação para inicializar os componentes do sistema antes de utilizar qualquer outra função da API.

retorno:

Esta função não retorna nada.

---

nome: **open** *(versão utilizando nomes para arquivos)*

sintaxe:

int **open** (*const char** name, *int* flags, *int* mode);
int **open** (*const char** name, *int* flags);

descrição:

A função **open** abre um arquivo, ou cria um novo arquivo de nome *name* de acordo com o tipo passado em *flags* definido durante a concepção do sistema de arquivos. Caso o arquivo não exista, ele deve ser criado fazendo uma máscara do tipo de arquivo com *O_CREAT*, como por exemplo: $TYPE\_RECORD|O\_CREAT$.
O parâmetro *mode* pode conter informações necessárias para um dado tipo de arquivo, mas não é sempre aplicável. Arquivos de tamanho fixo, ou que fazem uso de *buffers* circulares utilizam este campo para ajuste do tamanho do arquivo no momento da criação.

retorno:

No caso de sucesso, esta função retorna o descritor de arquivos, caso contrário retorna **-1** e ajusta o valor da variável **errno**
A variável **errno** pode conter os seguintes valores:
ETYPE - O tipo não existe.
EMFILE - O limite do número de arquivos abertos foi alcançado.
ENOMEM - Memória insuficiente.

nome: **open** *(versão utilizando números para arquivos)*

sintaxe:

int **open** (*int* fd, *int* flags, *int* mode);
int **open** (*int* fd, *int* flags);

descrição:

A função **open** abre um arquivo, ou cria um novo arquivo identificado pelo número *fd* de acordo com o tipo passado em *flags* definido durante a concepção do sistema de arquivos. Caso o arquivo não exista, ele deve ser criado fazendo uma máscara do tipo de arquivo com *O_CREAT*, como por exemplo: $TYPE\_RECORD|O\_CREAT$.
O parâmetro *mode* pode conter informações necessárias para um dado tipo de arquivo, mas não é sempre aplicável. Arquivos de tamanho fixo, ou que fazem uso de *buffers* circulares utilizam este campo para ajuste do tamanho do arquivo no momento da criação.

retorno:

No caso de sucesso, esta função retorna o descritor de arquivos, caso contrário retorna **-1** e ajusta o valor da variável **errno**
A variável **errno** pode conter os seguintes valores:
ETYPE - O tipo não existe.
EMFILE - O limite do número de arquivos abertos foi alcançado.
ENOMEM - Memória insuficiente.

---

nome: **creat** *(versão utilizando nomes para arquivos)*

sintaxe:

int **creat** (*const char* * name, *int* flags);

descrição:

A função **creat** cria um arquivo com o nome *name*, de acordo com o tipo passado em *flags* definido durante a concepção do sistema de arquivos.

retorno:

Em caso de sucesso, esta função retorna o descritor de arquivos, caso contrário retorna -1 e ajusta o valor da variável **errno**.
A variável **errno** pode conter os seguintes valores:
ETYPE - O tipo não existe.
EMFILE - O limite do número de arquivos abertos foi alcançado.
ENOMEM - Memória insuficiente.

nome: **creat** *(versão utilizando números para arquivos)*

sintaxe:

int **creat** (*int* fd, *int* flags);

descrição:

A função **creat** cria um arquivo de número *fd*, de acordo com o tipo passado em *flags* definido durante a concepção do sistema de arquivos.

retorno:

Em caso de sucesso, esta função retorna o descritor de arquivos, caso contrário retorna -1 e ajusta o valor da variável **errno**.
A variável **errno** pode conter os seguintes valores:
ETYPE - O tipo não existe.
EMFILE - O limite do número de arquivos abertos foi alcançado.
ENOMEM - Memória insuficiente.

---

nome: **close**

sintaxe:

int **close** (*int* fd);

descrição:

A função **close** remove a associação do descritor de arquivos *fd* com o arquivo em questão. Liberando recursos associados com o uso do arquivo.

retorno:

Em caso de sucesso é retornado zero, caso contrário é retornado **EOF** e ajustado o valor da variável **errno**.

nome: **write**

sintaxe:

ssize_t **write** (*int* fd, *const void** buffer, *size_t* count);

descrição:

A função **write** escreve até *count* bytes de dados a partir do endereço dado por *buffer*, no arquivo referenciado por *fd*.

retorno:

Esta função retorna o número de bytes escritos com sucesso, em caso de erro retorna um número menor do que o esperado ou zero. Em caso de erro é retornado -1 e **errno** é ajustado
Valores possíveis de erro são:
EBADF - descritor de arquivos é inválido
EIO - Ocorreu um erro de entrada/saída.

---

nome: **read**

sintaxe:

ssize_t **read** (*int* fd, *void** buffer, *size_t* count);

descrição:

A função **read** lê até *count* bytes de dados do descritor de arquivos *fd* para a posição apontada por *buffer*.

retorno:

No caso de sucesso, retorna o número de elementos lidos com sucesso e avança o indicador de posição do arquivo, em caso de erro retorna -1 e ajusta a variável **errno**
Valores possíveis de erro são:
EBADF - descritor de arquivos inválido.
EIO - Ocorreu um erro de entrada / saída.

nome: **unlink** *(versão utilizando nomes para arquivos)*

sintaxe:

int **unlink** (*const char* * name);

descrição:

Esta função elimina um arquivo especificado pelo parâmetro **name**. O arquivo deve estar fechado antes de tentar removê-lo.

retorno:

Em caso de sucesso, retorna zero, caso contrário retorna EOF e a variável **errno** é ajustada.
A variável de erro pode conter:
ENOMEM - Memória insuficiente
EROFS - O nome se refere a um arquivo somente de leitura
EIO - Ocorreu um erro de entrada/saída

---

nome: **unlink** *(versão utilizando números para arquivos)*

sintaxe:

int **unlink** (*int* fd);

descrição:

Esta função elimina um arquivo especificado pelo parâmetro *fd*. O arquivo deve estar fechado antes de tentar removê-lo.

retorno:

Em caso de sucesso, retorna zero, caso contrário retorna EOF e a variável **errno** é ajustada.
A variável de erro pode conter:
ENOMEM - Memória insuficiente
EROFS - O nome se refere a um arquivo somente de leitura
EIO - Ocorreu um erro de entrada/saída

nome: **get_status**

sintaxe:

void **get_status** (*int* fd, *struct file_status* *st);

descrição:

Esta função retorna o status de um arquivo especificado pelo parâmetro *fd* e armazena essas informações na estrutura *file_status*.

retorno:

Esta função não retorna valores.

---

nome: **set_date**

sintaxe:

void **set_date** (*int* fd, *date_t* *date);

descrição:

Esta função ajusta a data de um arquivo especificado pelo descritor *fd* utilizando o tipo *date*. Note que o tipo *date* possui um número variável de campos de acordo com a composição do SA.
Se o SA foi composto usando informações avançadas (METADATA==ADV) então todas as informações estã disponíveis, mas se ele foi composto com (METADATA==SIM), então apenas algumas informações estarão disponíveis.

retorno:

Esta função não retorna valores.

---

nome: **rewind**

sintaxe:

void **rewind** (*int* fd);

descrição:

Esta função muda o valor da posição relativa do arquivo especificado pelo descritor *fd* para o início deste.

retorno:

Esta função não retorna valores.

nome: **rename**

sintaxe:

int **rename** (*const char* * source, *const char* * target);

descrição:

Esta função muda o nome do arquivo especificado por *source* para *target*.

retorno:

Esta função retorna zero em caso de sucesso ou EOF se um erro ocorreu.

nome: **trunc**

sintaxe:

int **trunc** (*int* fd, *ssize_t* new_size);

descrição:

Esta função reduz o tamanho de um arquivo especificado por *fd* para um novo tamanho *new_size*, que deve ser menor que o tamanho anterior do arquivo.

retorno:

Em caso de sucesso, retorna zero, caso contrário retorna EOF e a variável **errno** é ajustada.

nome: **lseek**

sintaxe:

off_t **lseek** (*int* fd, *off_t* offset, *int* whence);

descrição:

Esta função muda o valor do indicador de posição relativa do arquivo especificado por *fd* para uma nova posição especificada pelo deslocamento *offset* de acordo com *whence* como o seguinte:
SEEK_SET - O *offset* é em relação ao início do arquivo;
SEEK_CUR - O *offset* relação a posição atual do arquivo;
SEEK_END - O *offset* relação ao fim do arquivo;
whence pode ser um valor positivo ou negativo.

retorno:

Em caso de sucesso, retorna o valor em bytes do indicador de posição do arquivo, caso contrário retorna -1 e a variável **errno** é ajustada.
A variável de erro pode conter:
EBADF - Descritor de arquivo não esta aberto.

---

nome: **eof**

sintaxe:

int **eof** (*int* fd);

descrição:

A função **eof** testa o indicador de final do arquivo *fd*.

retorno:

Retorna zero se o indicador de final do arquivo não esta marcado e retorna -1 caso contrário.

---

nome: **ferror**

sintaxe:

int **ferror** (*int* fd);

descrição:

Esta função testa o indicador de erro do arquivo especificado por *fd*.

retorno:

Esta função retorna zero, caso o status de erro do arquivo não esteja marcado ou retorna um número diferente de zero para erro.

nome:	**ftell**
sintaxe:	*ssize_t* **ftell** (*int* fd);
descrição:	Esta função retorna o valor do ponteiro de posição relativa do arquivo especificado por *fd*.
retorno:	Esta função retorna a posição em bytes se não ocorreram erros, ou retorna EOF em caso de erro e ajusta a variável **errno**.

Apêndice B

Diagramas de funções do SA

Estes diagramas mostram as funções disponíveis na *API* do programador fazendo chamadas às funções internas ao sistema de arquivos. Estas informações são úteis para conhecer a complexidade em relação ao tempo dessas funções, o número de funções que são invocadas e os semáforos que elas utilizam.

Os semáforos que representam maiores problemas (pois bloqueiam recursos por mais tempo) são os que controlam o acesso a tabela de arquivos (representado pelo semáforo *semaphore_lookup*) e o acesso a memória (no caso de alocação de espaço) representado pelos semáforos *semaphore_mm* e *semaphore_mm_index*.

Nas funções, também são utilizados semáforos *semaphore_status* quando for necessário acessar ou mudar o valor de variáveis simples. Mesmo ocorrendo em todas as funções, eles não representam um grande problema para inversões de prioridades pois sempre a seção crítica que eles protegem são compostas por poucas instruções que somente fazem atribuições e leituras de valores de variáveis simples.

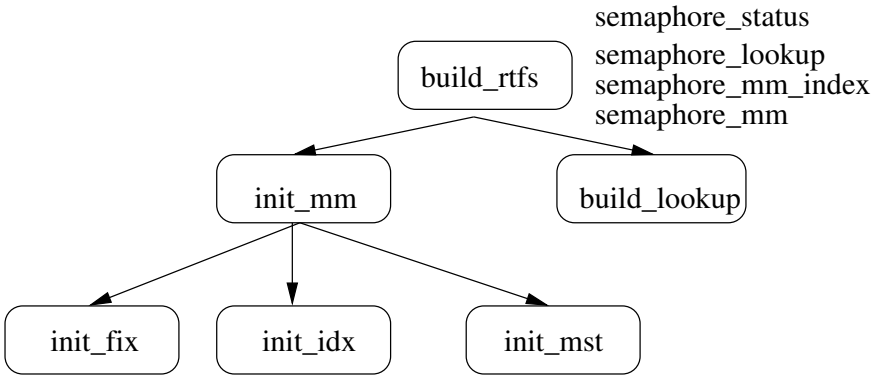
build_rtfs()

Comentário:

Esta função deve ser utilizada para criar o sistema de arquivos. As rotinas de acesso ao sistema de arquivos somente poderão ser utilizadas após a sua conclusão.

Como esta função é utilizada somente na criação do sistema de arquivos, e neste momento nenhum arquivo está em uso ela não representa maiores problemas quando a inversões de prioridade.

Utiliza os semáforos *semaphore_status*, *semaphore_mm*, *semaphore_mm_index* e *semaphore_lookup*.



`open()`

Comentário:

A função `open()` tem a função de criar ou abrir arquivos. Para o caso de arquivos cujo tamanho é fixo ela também pré-aloca memória para estes.

Esta função ajusta algumas informações de *status* do arquivo em questão e para tanto, usa o semáforo *semaphore_status*. Ela também utiliza três funções auxiliares, que são:

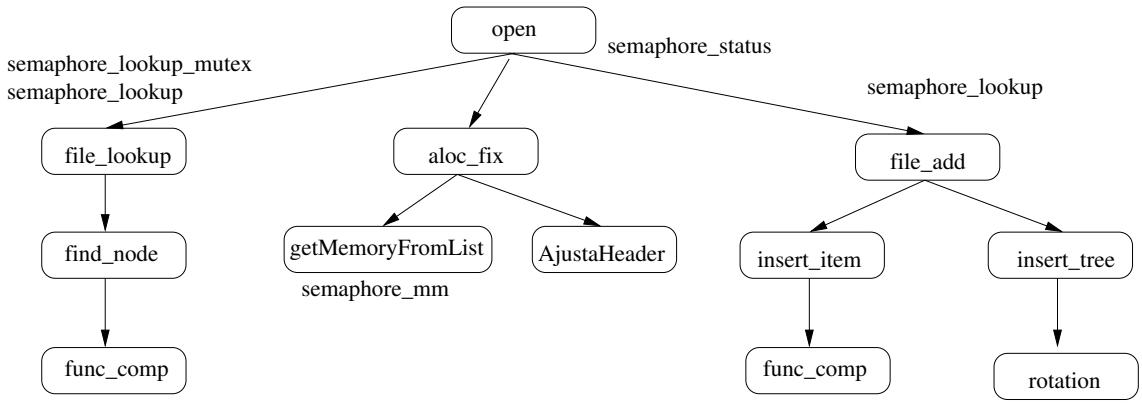
file_lookup() Retorna a posição de um arquivo na tabela de arquivos ou retorna um numero negativo caso este não exista. Esta função usa os semáforos binários *semaphore_lookup_mutex* e *semaphore_lookup* para obter acesso a estrutura de dados que representa a tabela de arquivos.

A função `file_lookup` possui um custo alto em termos de tempo, pois enquanto um arquivo esta sendo pesquisado na tabela, esta fica inacessível para escrita (inclusão de novos arquivos). O tempo máximo dessa pesquisa está ligado com a análise de pior caso dos algoritmos que são utilizados para representar a tabela de arquivos (árvores *red-black* ou pesquisa seqüencial);

aloc_fix() Responsável por alocar a memória para arquivos que possuam tamanho fixo, como os baseados no meta-tipo FIXO e no meta-tipo CIRCULAR;

file_add Insere um arquivo na tabela de arquivos (quando a função `open()` esta sendo utilizada para criar um arquivo). Precisa ter acesso exclusivo sobre esta estrutura de dados, para tanto, usa o semáforo binário *semaphore_lookup*. Possui um alto custo em termos de tempo, pois deve encontrar uma posição livre para inserir a referência ao arquivo na estrutura de dados usada, resultando numa pesquisa.

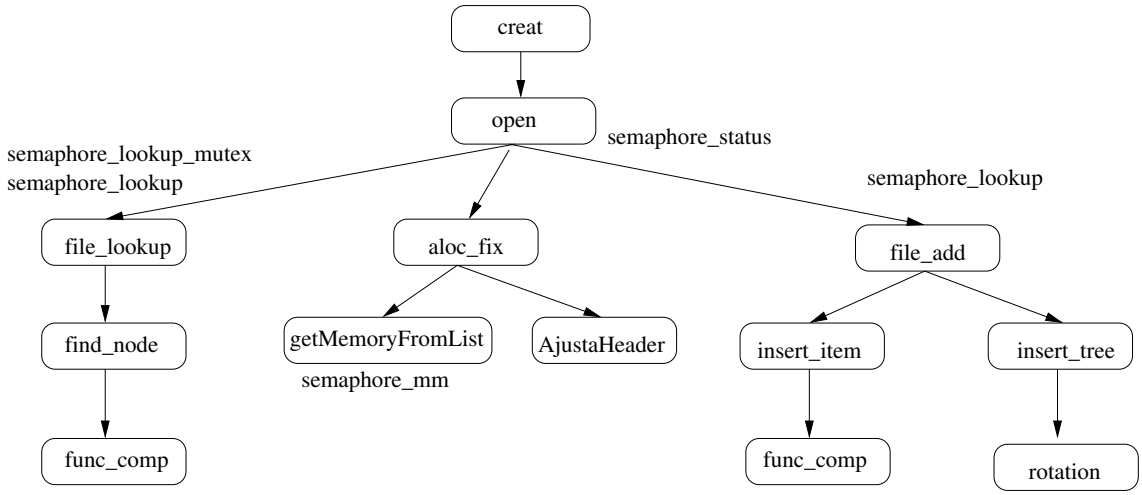
Como foi mostrado, a função `open()` é uma função complexa que deixa indisponível o acesso a estrutura de dados que representa a tabela de arquivos do sistema, enquanto um arquivo esta sendo criado. Esta é uma operação que pode ser demorada se existirem muitos arquivos no sistema e se o algoritmo utilizado para representar a tabela de arquivos for o de pesquisa seqüencial.



`creat()`

Comentário:

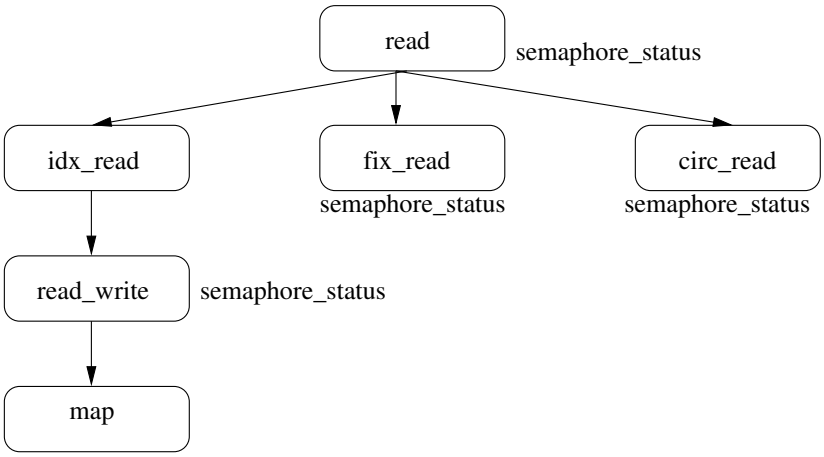
A função `creat()` utiliza a função `open()` para criar um arquivo. Sendo assim, possui as mesmas características desta.



`read()`

Comentário:

A função `read()` utiliza somente semáforos necessários para deixar o sistema de arquivos num estado consistente. Caso várias *threads* utilizem o mesmo arquivo, cabe ao programador utilizar métodos para manter a consistência das informações. Não é uma função complexa em relação ao tempo de execução e não mantêm acesso exclusivo em seções críticas por muito tempo.

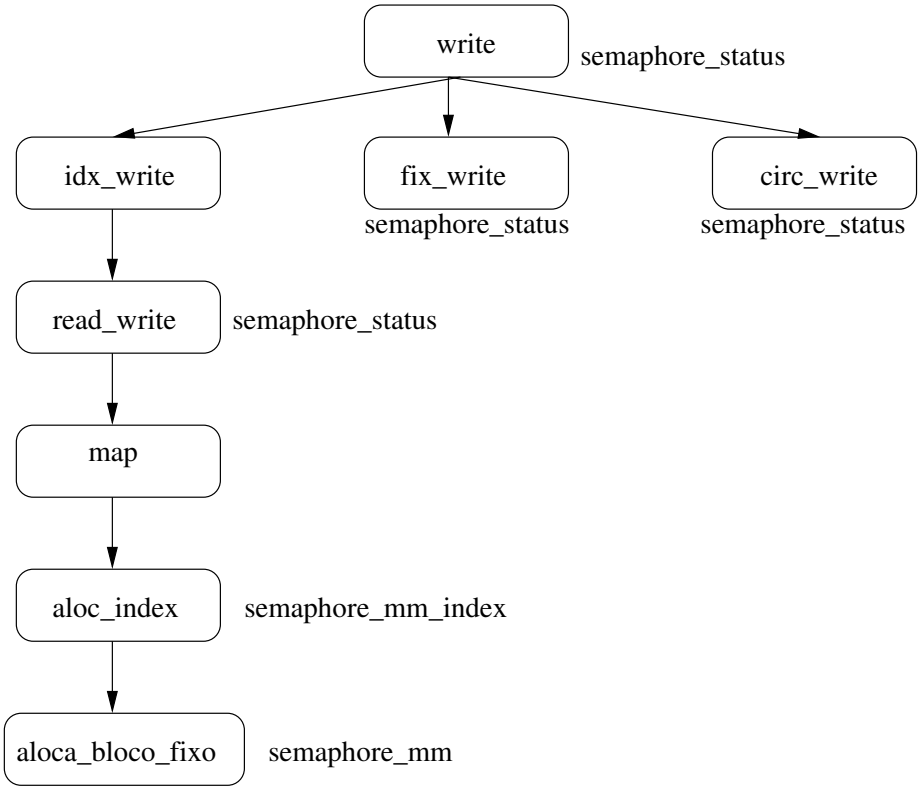


write()

Comentário:

Utiliza somente semáforos que protegem uma seção crítica pequena (apenas duas ou três linhas de código simples, com complexidade $O(1)$). Se a escrita for em arquivos baseado nos meta-tipos circular ou fixo, então é uma operação trivial.

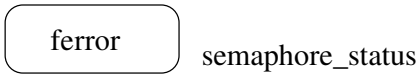
Caso a escrita seja feita em arquivos baseados no meta-tipo indexado, a operação é mais complexa, pois envolve alocar memória para os novos blocos. Caso o espaço de memória designada para alocação indexada tenha terminado, então a função aloca memória da área designada a alocação fixa (conforme descrito no capítulo 4). Os bloqueios do semáforo semaphore_mm_index ocorrem para cada novo alocado. bloco.



ferror()

Comentário:

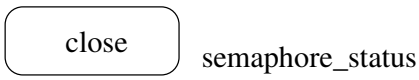
Função simples. Somente verifica o valor de uma estrutura que mantêm o status do arquivo, a seção crítica possui somente uma instrução simples. Complexidade $O(1)$.



close()

Comentário:

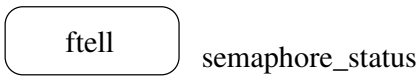
Função simples. Somente ajusta o valor de uma estrutura que mantêm o status do arquivo, a seção crítica possui somente uma instrução simples. Complexidade $O(1)$.



ftell()

Comentário:

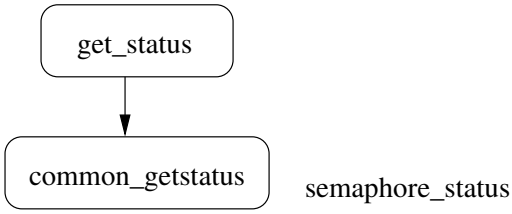
Função simples. Somente verifica o valor de uma estrutura que mantêm o status do arquivo, a seção crítica possui somente uma instrução simples. Complexidade $O(1)$.



get_status()

Comentário:

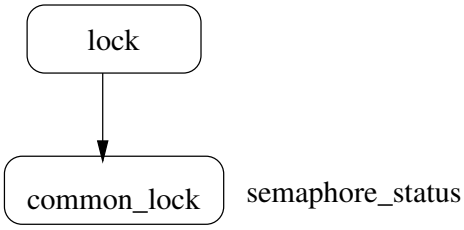
Função simples. Somente verifica o valor de uma estrutura que mantêm o status do arquivo, a seção crítica possui poucas instruções simples. Complexidade $O(1)$.



`lock()`

Comentário:

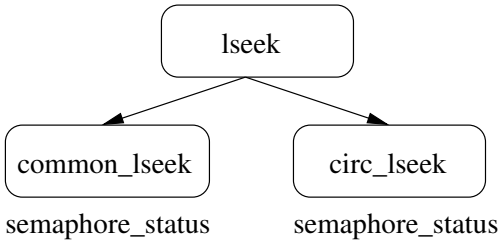
Função simples. Somente verifica o valor de uma estrutura que mantêm o status do arquivo, a seção crítica possui poucas instruções simples. Complexidade $O(1)$.



`lseek()`

Comentário:

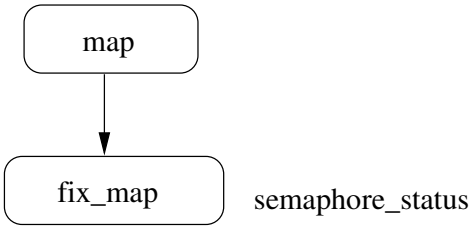
Função simples. Somente ajusta o valor de uma estrutura que mantêm o status do arquivo, a seção crítica possui somente algumas instruções simples. Complexidade $O(1)$.



`map()`

Comentário:

Função simples. Somente associa o endereço de um arquivo fixo a um ponteiro. Complexidade $O(1)$.

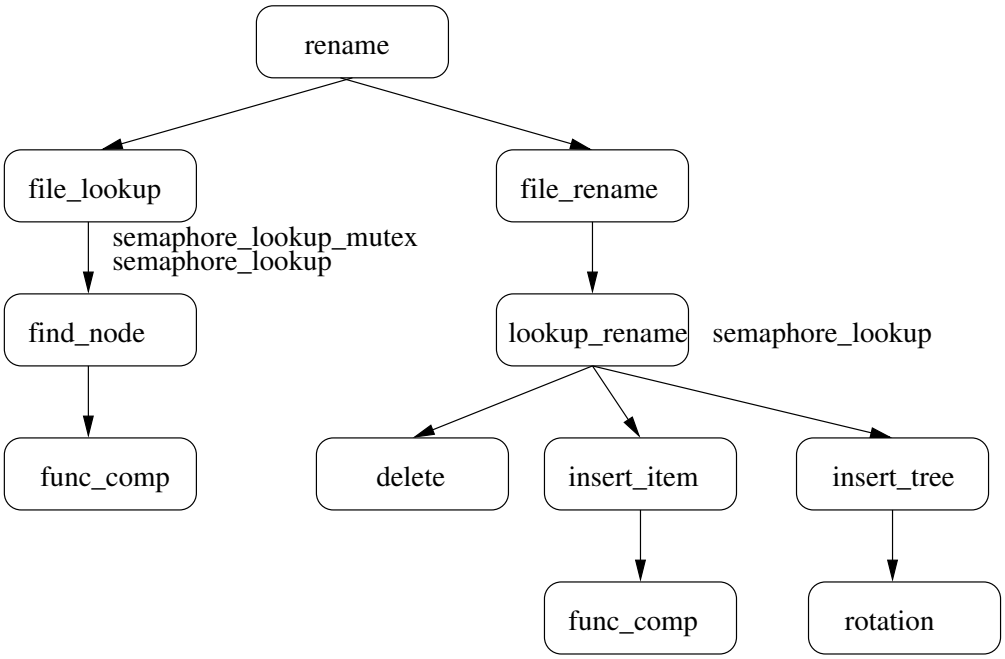


rename()

Comentários:

A função rename() troca o nome de um arquivo. Sua implementação é construída removendo o arquivo da tabela de arquivos e criando uma nova entrada.

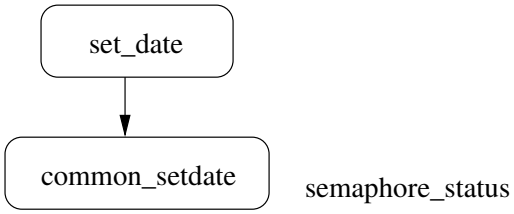
Essa função também verifica inicialmente se o arquivo existe no sistema. Por essas características, essa função é complexa em relação ao tempo.



set_date()

Comentários:

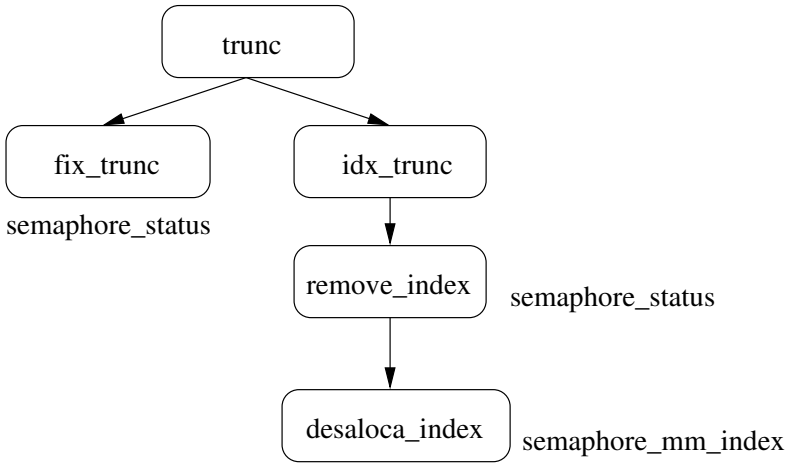
Função simples. Somente verifica o valor de uma estrutura que mantêm o status do arquivo, a seção crítica possui somente algumas instruções simples. Complexidade $O(1)$.



trunc()

Comentários:

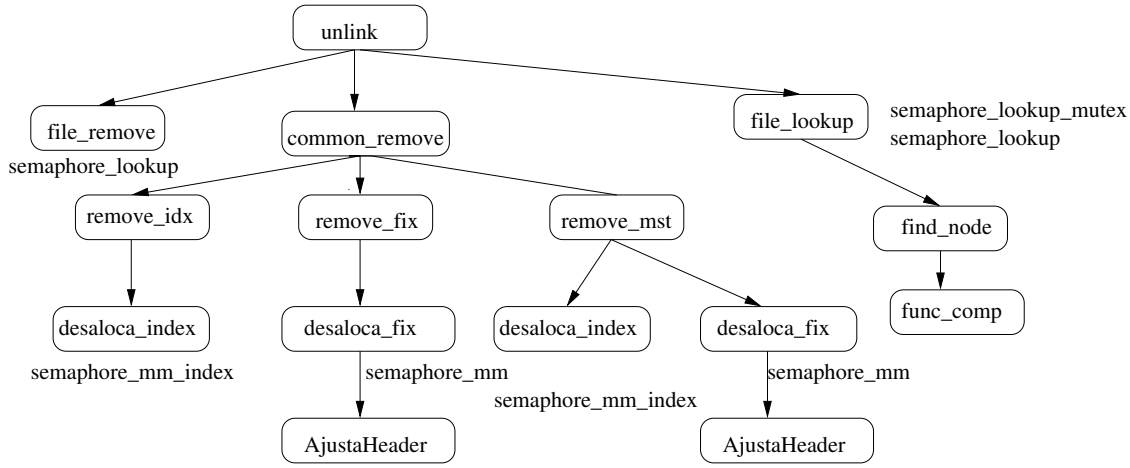
A função trunc() é simples quando utilizada em arquivos de tamanho fixo, pois nesse caso somente ajusta o campo tamanho do arquivo. Para o caso de arquivos indexados, esta função deve desalocar os blocos de memória anteriormente ocupados pelo segmento de arquivo liberado. Sendo assim, ela deve bloquear o acesso a memória durante a desalocação.



unlink()

Comentários:

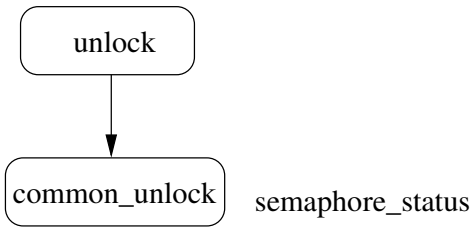
A função unlink() remove um arquivo do sistema de arquivos. É uma função complexa em relação ao tempo pois deve bloquear o acesso a memória durante a desalocação e bloquear a estrutura de dados que representa a tabela de arquivos.



unlock()

Comentários:

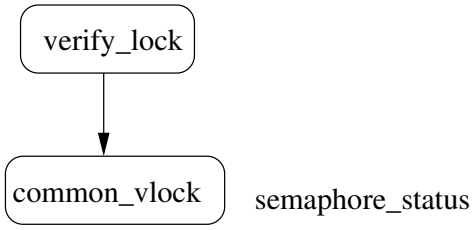
Função simples. Somente verifica o valor de uma estrutura que mantêm o status do arquivo, a seção crítica possui somente algumas instruções simples. Complexidade $O(1)$.



verify_lock()

Comentários:

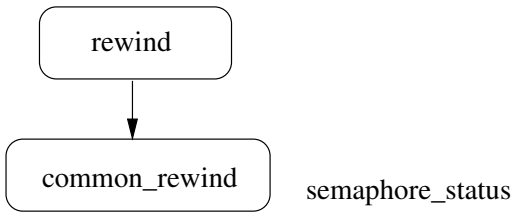
Função simples. Somente ajusta o valor de uma estrutura que mantêm o status do arquivo, a seção crítica possui somente algumas instruções simples. Complexidade $O(1)$.



rewind()

Comentários:

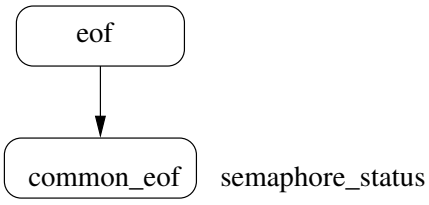
Função simples. Somente verifica o valor de uma estrutura que mantêm o status do arquivo, a seção crítica possui somente algumas instruções simples. Complexidade $O(1)$.



eof()

Comentário:

Função simples. Somente verifica o valor de uma estrutura que mantêm o status do arquivo, a seção crítica possui somente uma instrução simples. Complexidade $O(1)$.



Glossário

ANSI	American National Standards Institute.
API	Application Program Interface.
AVL	AVL Tree - Adelson-Velskii and Landis tree.
CIs	Circuitos Integrados.
DSP	Microprocessador otimizado para uso em aplicações envolvendo processamento de sinais.
EEPROM	Electrically Erasable Programmable Read-Only Memory.
EOF	End Of File.
FLASH-ROM	Dispositivo de memória não volátil que permite leitura, escrita e apagamento. Esses dispositivos são divididos em blocos (chamados setores) que são individualmente apagáveis.
FFS	Fast File System.
GPS	Global Positioning System.
GUI	Graphic User Interface.
PC	Personal Computer.
POSIX	Portable Operating System Interface.
RAM	Random-Access Memory. Dispositivo de memória volátil na qual todas as posições podem ser lidas e escritas.
RTAI	Real Time Application Interface.
ROM	Read-Only Memory. Dispositivo de memória no qual as posições de memória podem ser lidas mas não escritas.
SA	Sistema de Arquivos.
SAA	Sistema de Arquivos Abstrato.
SO	Sistema Operacional.

Referências Bibliográficas

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

Berger, A. S. (2002). *Embedded Systems Design*. CMP Books.

Cormen, T. (1989). *Introduction to algorithms*. MIT Press.

Farines, J. M., Fraga, J. d., e Oliveira, R. S. (2000). *Sistemas de Tempo Real*. Escola de Computação 2000.

Horowitz, E. e Sahni, S. (1984). *Fundamentals of Data Structures*. Computer Science Press.

Huber, B. (2002). Ramdisk: A sample user-defined C I/O driver. Technical report, Texas Instruments.

IEEE (1996). Portable operating system interface (POSIX) - part 1: System application programming interface (API). Technical report, IEEE.

Johnstone, M. S. e Wilson, P. R. (1999). The memory fragmentation problem: solved? *ACM SIGPLAN Notices*, 34(3):26–36.

Kruse, R. L. (1984). *Data Structures & Program Design*. Prentice-Hall, Englewood Cliffs, New Jersey 07632.

Lages, W. F. e Hemerly, E. M. (1999). Linux as a software plataform for mobile robots. *Submetido à IEEE Software*.

Lea, D. (1996). A memory allocator. *unix/Mail*, 6.

Martin, G. e Schirrmeister, F. (2002). A design chain for embedded systems. *IEEE Computer*, páginas 100–103.

Oliveira, R. S., Carissimi, A. d., e Toscani, S. S. (2000). *Sistemas Operacionais*. Editora Sagra Luzzato.

Ritchie, D. e Kernighan, B. (1988). *The C Programming Language*. Prentice Hall PTR.

Santos, C. S. e Azevedo, P. A. (2001). *Tabelas: Organização e Pesquisa*. Sagra Luzzatto.

Sargent III, M. e Shoemaker, R. (1995). *The personal computer from the inside out: the programmers guide to low-level PC hardware and software*. Addison wesley.

Silberschatz, A. e Galvin, P. (1997). *Operating System Concepts*. Wiley.

Stankovic, J. A. (2001). VEST — A toolset for constructing and analyzing component based embedded systems. *Lecture Notes in Computer Science*, 2211:390–??

Tanenbaum, A. S. (1992). *Modern Operating Systems*. Prentice Hall.

Toscani, L. V. e Veloso, P. A. (2001). *Complexidade de Algoritmos*. Editora Sagra Luzzato.

Vahalia, U. (1996). *Unix Internals: The new frontiers*. Prentice Hall.

Wolf, W. (2001). *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA.