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

Arliones Stevert Hoeller Junior

Famílias de Abstrações de Gerência de Memória para o EPOS

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

Prof. Dr. Antônio Augusto Medeiros Fröhlich

Florianópolis, fevereiro de 2004

Famílias de Abstrações de Gerência de Memória para o EPOS

Arliones Stevert Hoeller Junior

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciências da Computação, e aprovado em sua forma final pela Coordenadoria do Curso de Bacharelado em Ciências da Computação.

Prof. Dr. José Mazzucco Júnior

Banca Examinadora

Prof. Dr. Antônio Augusto Medeiros Fröhlich

Prof. Dr. José Mazzucco Júnior

B. Sc. Marcelo Triervellier Pereira

Para minha família e amigos, pessoas que me apoiaram e a quem devo o sucesso deste trabalho.

Resumo

Sistemas Embutidos e *Ambientes de Programação Paralela* têm estado cada vez mais presentes no cotidiano das pessoas. Hoje, a grande maioria dos dispositivos eletrônicos existentes utilizam algum tipo de processador ou microcontrolador em sua implementação. Além disso, o avanço de algumas ciências complexas, como a genética e a astronomia, têm exigido computadores mais potentes para realizar seus cálculos e manipular enormes volumes de dados. Contudo, tais computadores têm sido substituídos por *clusters* de computadores, ou seja, por ambientes de programação paralela.

Um dos maiores problemas enfrentados hoje pelos engenheiros de *software* que trabalham com estes sistemas é a grande diversidade de plataformas utilizadas nestes dispositivos. Esta diversidade traz implicações graves no desenvolvimento de *software*, pois comprometem a portabilidade destes sistemas, tornando seu desenvolvimento trabalhoso e caro.

Tendo como objetivo explorar os aspectos comuns destes sistemas, surgiu o *EPOS (Embedded Parallel Operating System)*, destinado, como o próprio nome diz, para plataformas embutidas e paralelas. Com o objetivo de tornar a implementação de *software* para estes sistemas menos custosa e favorecer a portabilidade do mesmo, o *EPOS* traz para o desenvolvimento de *software* básico recursos modernos de engenharia de *software*, sendo sua estrutura baseada, principalmente, em famílias de abstrações e de aspectos.

Este trabalho apresentará como foram projetadas e implementadas as famílias de abstrações relacionadas à gerência de memória do *EPOS*.

Palavras-chave: Gerência de Memória, Software Básico, Sistemas Embutidos, Paginação, Engenharia de Software, Projeto de Sistemas Orientado à Aplicação

Abstract

Embedded Systems and *Parallel Programming Environments* has been each time more present on people's daily life. Nowadays, most of the on-the-shelf electronic devices use some kind of processor or microcontroller in their implementation. Moreover, the advance of some complex sciences, such as genetics and astronomy, demands for more powerful hi-performance computers to perform calculations and to handle large amounts of data. However, such computers have been replaced by clusters of workstations, i. e., parallel programming environments.

One of the greatest problems faced in these times by software engineers working with these systems is the huge diversity of platforms being used in such devices. Such diversity brings to software development severe implications, once it compromises the system portability, making development more hard-working and expensive.

The *EPOS (Embedded Parallel Operating System)* was conceived focusing to explore the commonalities on such systems, and having as target those embedded systems and parallel programming environments. In order to achieve an easier way to develop basic software and a great level of portability, the *EPOS* system uses modern concepts of software engineering, and has its structure based in families of abstractions and aspects.

The goal of this work is to present how the families involved on the *EPOS's* Memory Management were designed and implemented.

Keywords: Memory Management, Basic Software, Embedded Systems, Paging, Software Engeneering, Application-Oriented System Design

Sumário

Lista de Figuras

Capítulo 1

Introdução

Este trabalho apresenta a implementação das famílias de abstrações envolvidas no gerenciamento de memória do Sistema Operacional *EPOS - Embedded Parallel Operating System* [dMF 02].

O trabalho é composto de duas partes, descritas a seguir:

Gerência de Memória no nível de Sistema: este é o gerente de memória física do sistema. Ele é responsável por gerenciar a alocação desta memória para os processos e/ou *threads* do sistema e pelo mapeamento de dispositivos de entrada e saída (I/O), quando necessário. Também é sua responsabilidade operar o *hardware* de gerência de memória (segmentação, paginação, etc), se existir. O gerente de memória do *EPOS* é constituído de 2 famílias de abstrações (*Address_Space* e *Segment*), e por uma família de mediadores (*MMU*)[1].

Gerência de Memória no nível de Usuário: a função do gerente de memória neste nível é controlar a alocação dinâmica de memória[2]. Devido ao fato de o *EPOS* não possuir uma interface de chamadas de sistema (*System Calls*), não é viável utilizar os métodos disponibilizados pela maioria dos compiladores na geração da

[1] Os termos *abstração (abstraction)* e *mediador (mediator)* têm significados específicos dentro do *EPOS*, como será discutido adiante.

[2] Alocação de memória na *Heap* de um processo ou *threads*, como as funções *malloc()* e *free()* disponíveis no *C/C++*.

aplicação, logo, é necessário que se forneça estas funcionalidades para o usuário. Aqui, este alocador dinâmico de memória constitui um utilitário[3] do sistema operacional.

1.1 Motivação

Em [TEN 00], Tennenhouse relata que, no ano de 2000, somente 2% dos processadores produzidos no mundo teriam como destino plataformas interativas (como computadores pessoais), ou seja, 98% destes processadores foram utilizados em sistemas dedicados, incluindo robôs, veículos e, principalmente, sistemas embutidos. Além disso, a grande maioria dos recursos destinados ao fomento de pesquisas nesta área têm como destino projetos que envolvem aqueles 2% de processadores.

Embora existam hoje muitos sistemas operacionais que se projetam no mercado como soluções para plataformas dedicadas, tais sistemas não provêm a mesma performance que pode ser oferecida por um sistema operacional desenvolvido especificamente para elas. Juntando a isso o fato de que, principalmente quando se trabalha com sistemas embutidos, há uma série de limitações no *hardware* utilizado (para diminuir o custo ou o espaço ocupado), precisamos sempre utilizar um sistema operacional que não prejudique o desempenho do sistema executando operações desnecessárias.

Contudo, como dito em [dMF 01] apud [AND 92a, SP 94], "*os adjetivos* genérico *e* ótimo *não podem ser atribuídos ao mesmo sistema operacional*"[4]. Assim sendo, para consiguir em um determinado sistema dedicado um conjunto de *software* básico (sistema operacional, *device drivers*, e compiladores) que satisfaça a requisitos de desempenho e que utilize todos os recursos disponíveis no *hardware*, é necessário que se desenvolva um sistema operacional diferente para cada nova plataforma, o que é, sem dúvida alguma, muito contraproducente.

Para tratar tais problemas, muitos pesquisadores da área de *software*

[3] O termo *utilitário (utility)* tem significado específico dentro do *EPOS*, como será discutido adiante.

[4] Tradução própria de: *the adjectives generic and optimal cannot be assigned to the same operating system*

básico têm dedicado seus esforços à criação e adaptação de técnicas de engenharia de *software* que possam prover ao projeto de sistemas operacionais vantagens como reusabilidade, facilidade de manutenção e desenvolvimento mais eficiente, vantagens estas que já têm ajudado projetistas e desenvolvedores de *software* aplicativo há muitos anos.

1.2 Justificativa

Nos últimos anos, o avanço tecnológico impulsionado pela necessidade de soluções práticas, baratas e de qualidade fez surgir dispositivos eletrônicos cada vez mais sofisticados. Embora as tecnologias tenham alcançado um grau de desenvolvimento muito alto, este mesmo avanço trouxe outros problemas, tais como a grande variedade de dispositivos existentes (comprometendo compatibilidade e portabilidade) e a necessidade de miniaturizar os recursos (em sistemas embutidos, principalmente).

Hoje, sempre que se cria um novo dispositivo, seja ele um forno de microondas ou um novo sistema de navegação para satélites, é preciso desenvolver um novo conjunto de *software* básico, o que encarece consideravelmente qualquer projeto. Uma das principais razões para que se faça esta "re-construção" do sistema é a *confiabilidade*.

Tendo por finalidade não só tornar o desenvolvimento de *software* básico menos desgastante e mais barato mas, principalmente, produzir sistemas operacionais que permitam que os dispositivos operem do modo mais eficiente, Fröhlich [dMF 01] sugere, através do uso de várias técnicas avançadas de engenharia de*software*, o desenvolvimento de Sistemas Operacionais Orientados à Aplicação (*AOOS - Application Oriented Operating Systems*), ou seja, sistemas operacionais que, diferentemente dos mais utilizados hoje, fazem uso do menor conjunto de *software* básico necessário para o seu funcionamento satisfatório e eficaz.

Objetivando a validação destes conceitos, os pesquisadores do *LISHA* (Laboratório de Integração *Software/Hardware*) [HTT 04c], coordenados pelo Prof. Dr. Fröhlich, estão desenvolvendo o *EPOS - Embedded Parallel Operating System* [dMF 02], sistema operacional que segue os conceitos da AOOS, projetado para plataformas dedicadas embutidas, tais como vídeo-games e eletro-eletrônicos, e para ambientes de computa-

ção paralela e distribuí da.

Considerando os recursos reduzidos, fato comum neste tipo de plataforma, é necessário obter o melhor desempenho possí vel destes dispositivos. Um destes recursos escassos na maioria das plataformas embutidas é a memória, que precisa ser gerenciada de forma efi ciente e de modo a evitar desperdí cio.

1.3 Objetivos

Este texto tem como objetivo fi nal demonstrar como foram desenvolvidas as famí lias de abstrações envolvidas na gerência de memória do EPOS (*Embedded Parallel Operating System*) [dMF 02].

1.3.1 Objetivo Geral

Como já dito, o objetivo deste trabalho é projetar e desenvolver, através do emprego das técnicas descritas por Fröhlich em [dMF 01] e baseado no projeto original do EPOS [dMF 01, dMF 02], um gerente de memória para este sistema operacional. Tal sistema deverá satisfazer requisitos de portabilidade e performance condizentes à sua aplicação (Sistemas Embutidos e Paralelos).

1.3.2 Objetivos Especí fi cos

Os objetivos especí fi cos foram divididos em três:

Pesquisa: levantamento do estado da arte das áreas de *software* básico, sistemas dedicados, e sistemas de gerência de memória através do estudo da literatura clássica e artigos cientí fi cos destas áreas, bem como através da análise dos diversos sistemas operacionais de código-aberto (*open-source*) atualmente utilizados.

Análise e Projeto: o trabalho foi modelado de acordo com a metodologia AOSD (*Application Oriented System Design*) [dMF 01], tendo como objetivo sua integração com o Sistema Operacional *EPOS*.

Implementação, Integração e Testes: implementação do sistema modelado na fase anterior, utilizando a linguagem *C++* [STR 97] e o conjunto de ferramentas e compiladores da *GNU* [HTT 04b], integração do gerente de memória com o sistema operacional *EPOS* e teste do sistema nas plataformas disponí veis no*LISHA* (Laboratório de Integração *Software/Hardware - UFSC*).

Capítulo 2

Conceitos Básicos

O termo *gerenciador de memória* pode ser empregado sob dois pontos de vista: o ponto de vista do sistema operacional e o da aplicação.

Sob o ponto de vista do sistema operacional, o gerenciador de memória é um conjunto de funcionalidades que deve permitir às camadas de mais alto nível o acesso à memória física e aos dispositivos de entrada e saída, provendo serviços de proteção e compartilhamento. Já o ponto de vista da aplicação é diferente. Para a aplicação é necessário que os detalhes da implementação do sistema operacional e do *hardware* fiquem transparentes, ou seja, que seja definida uma interface para o gerenciador de memória do sistema que atenda às necessidades da aplicação e que, ao mesmo tempo, seja independente das camadas mais baixas do mesmo gerenciador.

A seguir serão demostrados alguns conceitos básicos de gerenciamento de memória e, após, serão descritos alguns esquemas de tradução de endereços. Todos estes conceitos apresentados a seguir foram extraídos dos principais livros de sistemas operacionais [SIL 98, TAN 92] e em alguns artigos científicos que são citados no decorrer do texto.

2.1 Fragmentação

Fragmentação é o problema que ocorre quando a memória livre (seja para o usuário ou para o sistema) não está contiguamente posicionada. Este problema se torna grave quando uma requisição de alocação de memória é feita e existe memória livre suficiente para atendê-la mas, devido à fragmentação, é impossível proceder a alocação. Existem dois tipos de fragmentação:

Fragmentação Externa: este é o pior tipo de fragmentação, pois é impossível prever o tamanho da perda de memória que pode ser ocasionada. Este é um problema perigoso e difícil de lidar. Para inibir este tipo de fragmentação, geralmente, quando há suporte em *hardware*, utiliza-se paginação[1]. Quando não existe suporte em *hardware*, há dois modos para resolver este problema: (1) partições de tamanho fixo, que provê fragmentação interna, ou (2) o uso de algoritmos de compactação (discutidos adiante), o que implica em um sério *overhead*. A figura 2.1(a) demonstra a ocorrência de fragmentação externa, uma vez que há memória livre suficiente para alocar o processo X, mas como esta memória está fragmentada não é possível realizar a alocação.

Fragmentação Interna: este tipo de fragmentação está presente apenas em sistemas que fazem uso de partições de tamanho fixo para alocar memória, ou costumam alocar mais memória que o solicitado. Um exemplo bastante usado de esquema de alocação de memória que ocasiona fragmentação interna é a paginação. Esta fragmentação ocorre quando um bloco de memória alocada é maior do que precisaria ser, ou seja, há memória alocada que não está sendo usada. Esta situação está representada na figura 2.1(b), onde existe memória não usada dentro dos segmentos de memória alocados. Caso esta memória estivesse livre e contiguamente posicionada, a requisição X poderia ser alocada. Quando o esquema de gerência de memória não utiliza *hardware* para permitir tradução de endereços, e divide a

[1] Note que paginação provê fragmentação interna, mas não há a possibilidade de ocorrer fragmentação externa neste esquema, assim como em nenhum outro esquema que utiliza partições de memória com tamanho fixo. Paginação será discutida em maiores detalhes mais adiante.

memória em partições de tamanho fixo, a fragmentação interna pode se tornar um problema grave, dado que a perda de memória para fragmentação interna pode variar de 0% até quase 100% da partição, já que existe a possibilidade de se alocar uma partição para armazenar apenas um *byte*. A escolha de um algoritmo[2] apropriado para realização da alocação pode suavizar este problema. Em sistemas como paginação (*hardware* para tradução de endereço só necessário) este problema pode ser contornado de um modo bastante limpo e eficiente, já que, graças ao *hardware* para tradução de endereços, as partições de memória (chamadas então de *páginas*) não precisam ser contíguas para satisfazer as requisições de alocação.

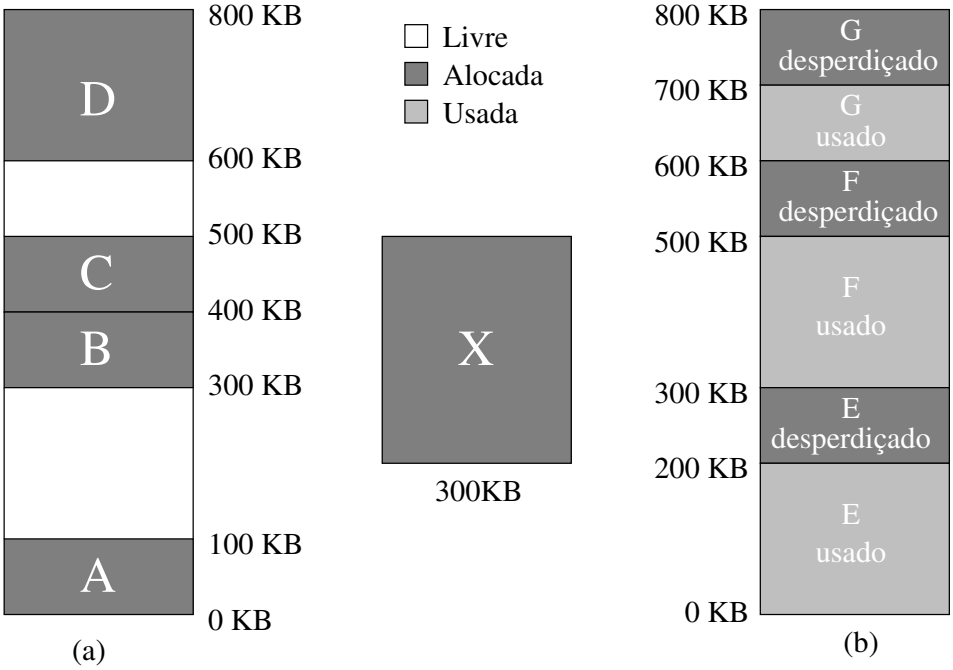

Figura 2.1: A fragmentação impede que o processo X seja alocado devido a fragmentação externa (a) e fragmentação interna (b).

Finalmente, se a opção no momento de projetar um gerente de memória estiver entre um esquema que provoque fragmentação externa e outro que provoque fragmentação interna, existe uma grande possibilidade de que, optando-se pelo último, a perda de memória seja menor.

[2] Algoritmos de alocação serão discutidos adiante.

2.2 Alocação de Memória

A definição de um esquema de alocação de memória é composta por três entidades: *estratégia de alocação*, *política de alocação* e *mecanismo de alocação*. Este último também referenciado como *algoritmo de alocação*. Esta seção irá descrever estes conceitos.

2.2.1 Estratégias de Alocação

Estratégia de alocação é uma modelagem de alto nível das características que se deseja em um gerente de memória. Esta descrição é constituída de observações e teorias para justificar a escolha de uma *política de alocação*, que é finalmente implementada por um *mecanismo de alocação*.

Dada a incomensurável variedade de aplicações existentes, é fácil deduzir que, ao longo dos anos, tenham sido desenvolvidas várias políticas e inúmeros algoritmos de alocação para atender às mais diversas estratégias definidas. Portanto, torna-se difícil, ou praticamente impossível, apresentar todas neste trabalho. Contudo, foram selecionadas algumas das mais utilizadas, que serão discutidas a seguir.

2.2.2 Políticas de Alocação

Uma política de alocação é definida através da análise da estratégia de alocação desejada para um determinado gerente de memória. Contudo, mesmo sendo estes os passos mais corretos para a implementação de um gerente de memória, hoje em dia são raros os casos em que se cria uma nova política. O mais comum é que se faça a definição da estratégia de alocação e se escolha a política que satisfaça a maior parte dos requisitos desejáveis. A seguir serão descritas algumas das mais utilizadas políticas de alocação:

Address-Ordered First-Fit: esta política de alocação sempre aloca o bloco capaz de satisfazer a alocação que tenha o menor endereço. Esta é a política de alocação de memória mais comumente utilizada. Normalmente ela é implementada através de

um algoritmo de busca do tipo *First-Fit*[3], que realiza sua busca sobre uma lista encadeada de blocos livres. Algumas vezes esta política é referenciada simplesmente como *First-Fit*[4].

LIFO-Ordered First-Fit: LIFO significa *Last In First Out*, ou seja, é o conceito de *Pilha*, onde o último elemento "empilhado" é sempre o primeiro a ser "desempilhado". Nesta política, o bloco escolhido é aquele que, além de ser grande o suficiente para satisfazer a requisição, tenha sido mais recentemente liberado. Geralmente, esta política é implementada anexando blocos liberados na frente de uma lista de blocos livres, e então usando um algoritmo *First-Fit* sobre esta lista. Aqui o processo de liberação de memória pode ser muito rápido, dependendo da política utilizada para realizar a fusão de blocos livres contíguos, e o processo de alocação de memória não deve ser mais lento que o da política *Address-Ordered First-Fit*. Contudo, esta política pode sofrer fragmentação externa quando ocorrerem alocações de grandes blocos de memória de mesmo tamanho que são rapidamente liberados. Neste caso, conforme vão sendo realizadas alocações de pequenos blocos, a lista de blocos livres é preenchida com fragmentos um pouco menores que o tamanho dos grandes blocos.

Worst-Fit: aqui, sempre é alocado o maior bloco livre. Normalmente, esta política é implementada através de uma lista de blocos livres ordenada pelo tamanho dos blocos, utilizando-se de um algoritmo *First-Fit* para realizar a escolha. O processo de alocação de memória é rápido, já que basta verificar se o primeiro bloco da lista é capaz de atender a solicitação, se sim, aloca-se o bloco, senão, ocorre uma falha. O processo de liberação é similar ao da política *Address-Ordered First-Fit*, já que também implementa uma lista ordenada de blocos livres, neste caso ordenada por tamanho. No entanto, esta política tende a não operar muito bem, já que ela elimina

[3] Mecanismos de alocação serão discutidos mais adiante.

[4] Não confundir com o algoritmo First-Fit. Neste caso, o termo First-Fit refere-se à mais conhecida política de alocação. Esta política faz uso do algoritmo First-Fit, mas envolve muitos outros conceitos além deste.

rapidamente os blocos grandes de memória, fazendo com que requisições maiores não possam ser devidamente atendidas após um certo tempo.

Best-Fit: esta política sempre aloca o menor bloco livre de memória capaz de satisfazer a requisição. Algoritmos utilizados nesta política incluem: (1)*Sequential-Fit*[5] buscando por um *bloco perfeito*, ou seja, um bloco do mesmo tamanho da requisição, (2) *First-Fit* em uma lista de blocos livres ordenada por tamanho dos blocos, (3) *Segregated-Fits*[6], (4) ou ainda *Indexed-Fits*[7]. Em teoria, esta política pode exibir fragmentação, porém, na prática, isto não é observado com muita frequência.

Good-Fit: *Good-Fit* é, na verdade, um termo utilizado para referenciar uma classe de políticas de alocação que tentam aproximar o *Best-Fit*. Estas políticas são bastante utilizadas porque as implementações do *Best-Fit* podem ser muito caras computacionalmente (dependendo dos detalhes do algoritmo de alocação), então resolve-se por aproximar aquela política, alocando o primeiro bloco que tiver um tamanho próximo ao da requisição. A escolha do quão próximo pode ser o tamanho deste bloco está diretamente relacionada com a quantidade extra de fragmentação que esta política gerará, e varia de acordo com a implementação.

2.2.3 Algoritmos de Alocação

O *algoritmo de alocação*, ou *mecanismo de alocação*, é responsável por implementar a política utilizada para alocar memória. A escolha de um algoritmo apropriado, como já dito, pode refletir de modo importante no desempenho e na quantidade de memória perdida para fragmentação, principalmente em esquemas de gerência de memória que não possuam suporte de *hardware*.

A literatura clássica de sistemas operacionais, outros artigos, relatórios e documentações de implementações existentes apresentam inúmeros algoritmos. A seguir serão apresentadas as principais classes de algoritmos de alocação, citando alguns

[5]Busca sequencial.

[6]Técnica que utiliza diferentes listas, cada uma com blocos de diferentes tamanhos.

[7]Onde são utilizadas estruturas de dados indexadas, como árvores ou tabelas de hash.

representantes destas classes:

Bitmaped-Fit: esta classe utiliza *bitmaps* para controlar o uso da memória. Cada *bit* no *bitmap* corresponde a um bloco de memória. O processo de alocação é feito varrendo o *bitmap* a procura de uma sequência de *bits* que satisfaça a requisição. Estes mecanismos oferecem boa localidade espacial[8], já que ele evita examinar os cabeçalhos dos blocos livres, que estão distribuídos na memória. Isto pode ocasionar um bom ganho de desempenho, pois diminui a quantidade de faltas de acesso à memória *cache*. Dependendo do tamanho dos blocos marcados por cada *bit*, pode haver um grande problema de fragmentação interna neste algoritmo. Contudo, se o tamanho destes blocos forem muito pequenos, este mecanismo pode causar um *overhead* no uso da memória que se tornaria inaceitável. Logo, é extremamente importante que se faça uma boa análise das aplicações antes de definir as características deste algoritmo.

Sequential-Fit: classe de algoritmos de alocação que utilizam uma lista encadeada simples para armazenar os blocos de memória livre. Exemplos de algoritmos desta classe são o *First-Fit* e o *Next-Fit*. Algumas implementações utilizam listas duplamente encadeadas para controle dos blocos de memória livre, o que facilita a fusão de blocos contíguos de memória livre. Este mecanismo pode não ser muito eficiente quando utilizado para gerenciar a alocação de uma grande área de memória, já que o tempo de busca sequencial em uma lista com muitos blocos livres pode se tornar muito grande. Outro problema que pode ocorrer quando houverem muitos blocos livres é a fragmentação externa, já que a existência de uma grande quantidade de blocos de memória pre-supõe o fato de estes blocos serem pequenos e não contíguos. A figura 2.2 apresenta a alocação de um bloco de memória em uma lista de memória já em uso através o algoritmo *First-Fit* (de (a) para (b)), e a liberação de outro bloco de memória com a realização de fusão do mesmo bloco com um bloco contíguo (de (b) para (c)).

[8] A localidade espacial deve-se ao fato de o bitmap ser (na grande maioria das vezes) contíguo.

Segregated-Free-List: aqui, a lista de blocos livres é dividida em várias outras listas, de acordo com o tamanho dos blocos. Os blocos livres são inseridos, de acordo com o seu tamanho, na lista apropriada e as requisições de memória são servidas pela lista que contém os menores blocos capazes de satisfazer o pedido. Estes algoritmos implementam políticas como o *Good-Fit* e o *Best-Fit*. Variações destes mecanismos incluem *Simple Segregated Storage*[9], *Segregated-Fit*[10] e *Buddy-Systems*. Note que alguns destes mecanismos utilizam tamanhos fixos de blocos, o que geralmente leva à fragmentação interna, enquanto outros utilizam técnicas de divisão e fusão de blocos nos processos de, respectivamente, alocação e liberação de memória, o que pode causar fragmentação externa. Mais uma vez se faz importante a análise da aplicação para decidir qual algoritmo será mais apropriado. O problema de fragmentação interna pode ser observado na figura 2.3, onde há uma série de alocações e liberações de memória utilizando um *Buddy System*.

Indexed-Fit: esta classe de algoritmos envolve aqueles onde a busca por blocos livres de memória é realizada em uma estrutura de dados indexada (como árvores ou tabelas de *hash*). Por exemplo, pode-se utilizar uma árvore ordenada pelo tamanho dos blocos para implementar uma política *Best-Fit*. Para usar estes mecanismos, é necessário analisar a eficiência de alguns métodos. Árvores balanceadas e ordenação de árvores podem incluir métodos custosos, e esta pode não ser uma solução eficiente para alguns casos. Além disso, dependendo da política de alocação, estes mecanismos também podem causar fragmentação.

Como visto, todos os algoritmos mencionados acima podem causar fragmentação. Na verdade, evitar fragmentação é impossível. Seja ela interna ou externa, pelo menos uma das duas sempre existirá. Contudo, a escolha dos algoritmos apropriados para um certo conjunto de aplicações pode amenizar estes problemas. Uma grande solução para estes problemas é o uso de esquemas de tradução de endereços, como

[9] Geralmente se utiliza este tipo de alocador em sistemas paginados.

[10] Onde existe um *array* de listas de blocos livres, cada uma armazenando blocos em uma certa faixa de tamanho.

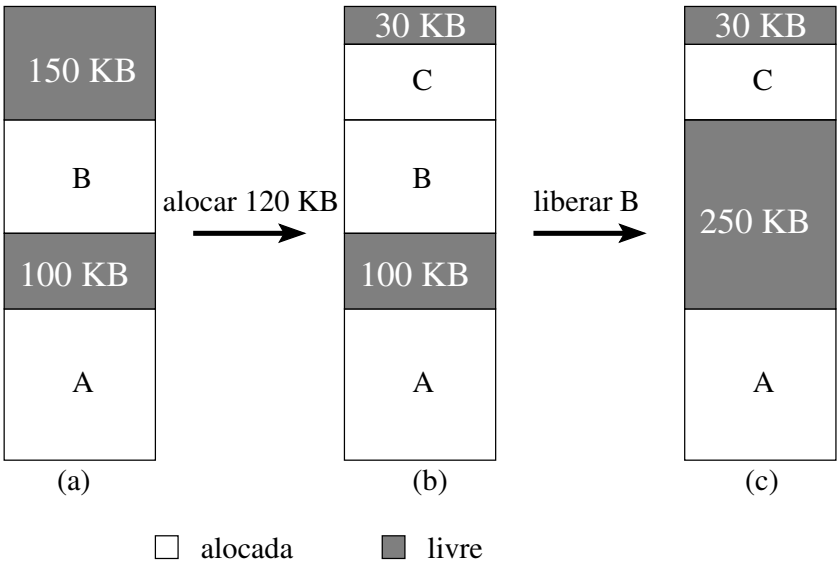

Figura 2.2: Processo de alocação utilizando listas encadeadas com o algoritmo First-Fit (a)(b) e de liberação de memória com fusão de segmentos de memória livres (b)(c).

paginação e *segmentação* (adiante será discutido como estes esquemas amenizam tais problemas). Contudo, na inexistência destes esquemas (que são, geralmente, implementados em *hardware*), as soluções existentes para amenizar a fragmentação podem tornar o sistema bastante inefi ciente.

Uma destas soluções é a compactação, e pode ser utilizada com qualquer um dos algoritmos citados. Algoritmos de compactação precisam re-arranjar a posição dos segmentos de memória alocados na memória fí sica, afi m de tornar contíguos os segmentos livres. Como este procedimento implica na realização de várias cópias de grandes segmentos de memória de uma posição para outra, ele é extremamente lento. Mesmo lento, pode ser uma boa solução em alguns casos se não for utilizado constantemente, pois, além de eliminar a fragmentação externa, ele aumenta a localidade destes blocos, diminuindo as faltas no acesso à *cache*.

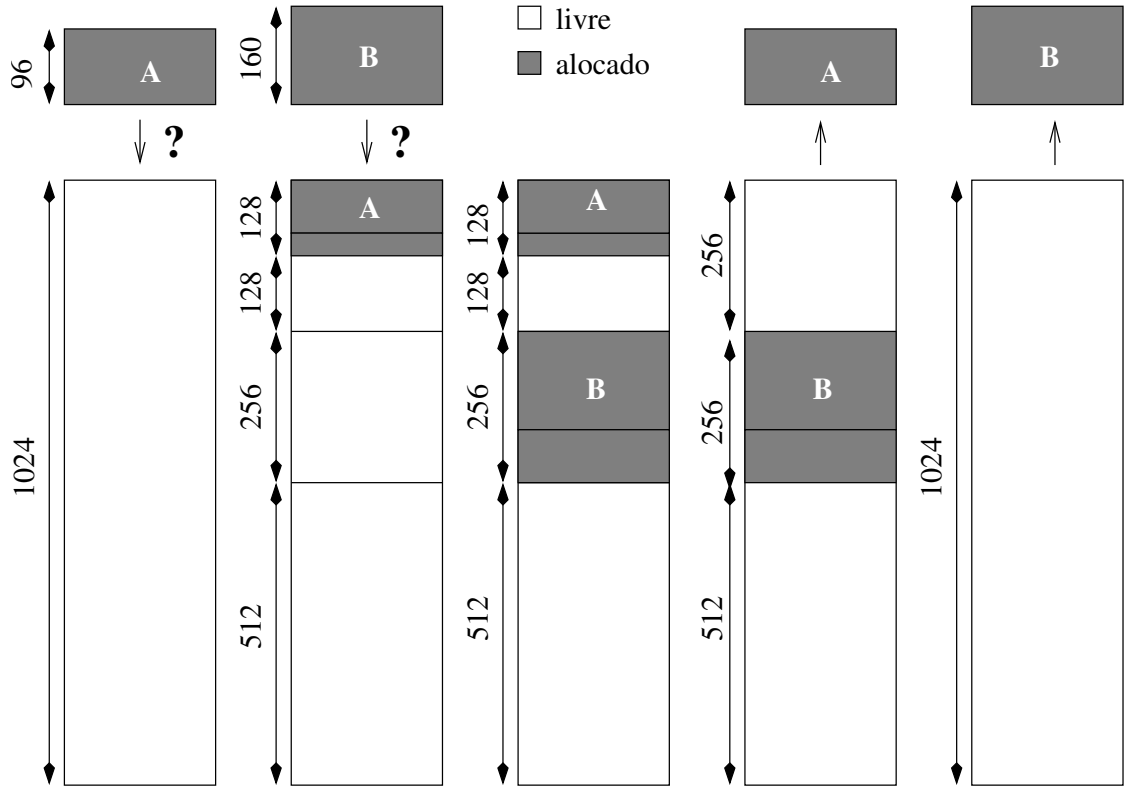

Figura 2.3: Processos de alocação e liberação de memória no sistema Buddy System.

2.3 Tradução de Endereç os

Esquemas de *tradução de endereç os* são largamente empregados hoje em dia para facilitar a implementação de sistemas multi-tarefa. Quando estes esquemas são utilizados, o processo tem a ilusão de estar sozinho na *CPU*, tendo todo o espaç o de endereç amento do processador para sí .

Boa parte das arquiteturas atualmente comercializadas oferecem algum tipo de suporte em *hardware* para tradução de endereç os. Este *hardware* recebe o nome de MMU (*Memory Management Unit* - Unidade de Gerenciamento de Memória). A MMU permite a separação entre espaç o de endereç amento real (endereç os físicos) e espaç o de endereç amento do processador/processo (endereç os lógicos). A tradução de endereç os é demonstrada na fi gura 2.4. A seguir serão apresentadas os principais equemas utilizados para tradução de endereç os.

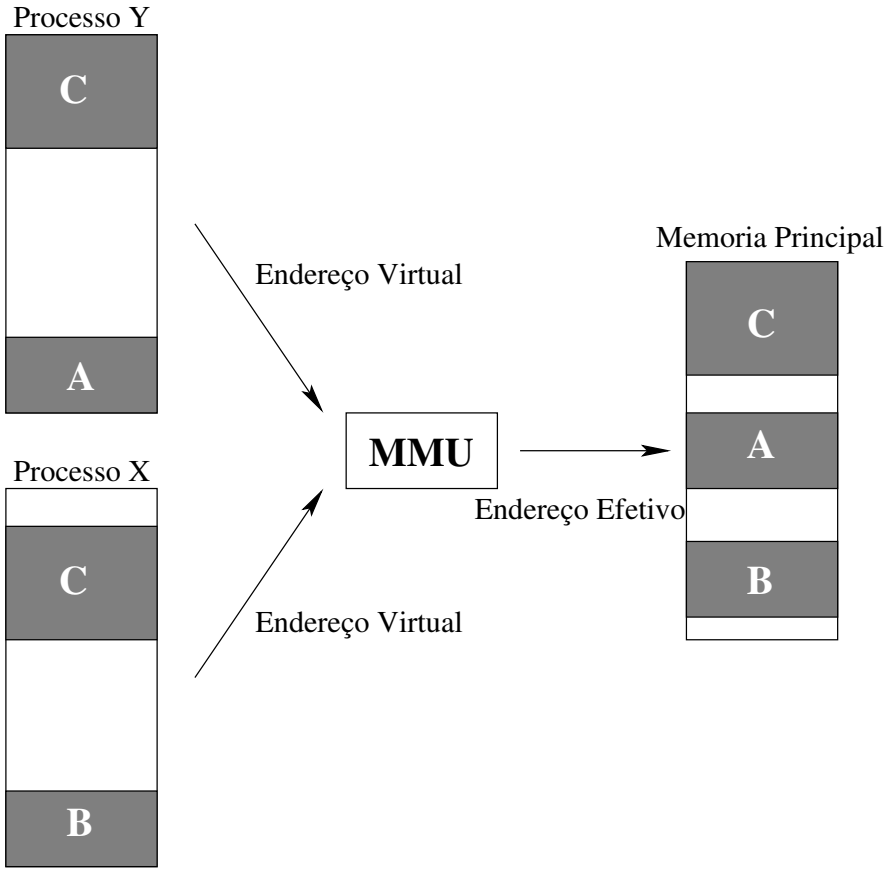

Figura 2.4: Tradução de endereç os realizada pela MMU.

2.3.1 Paginação

Este esquema de gerenciamento de memória consiste em dividir a memória fí sica em pequenos blocos de tamanho fi xo chamados*page frames*, ou apenas *frames*, e o espaç o de endereç amento do programa em páginas (*pages*), que têm o mesmo tamanho dos frames [TK 61, HOW 61]. Quando uma requisição de alocação é feita, é possí vel alocar dados na memória fí sica usando tantos frames quanto forem necessários. Tais frames não precisam ser contí guos na memória fí sica, mas podem ser mapeados contiguamente no espaç o de endereç amento do processo. A visão da memória que o processo tem é a visão lógica. Então, toda vez que uma referência à memória é feita, esta referência é traduzida pela MMU para um endereç o fí sico e a posição de memória desejada é acessada. A estrutura de dados utilizada para controlar o estado da memória fí sica chama-se

tabela de páginas (*page table*). O mecanismo de tradução de endereços utilizado pela paginação é demonstrado na figura 2.5. O endereço virtual (gerado pelo programa) é dividido em duas partes: um número de página virtual p e o deslocamento d dentro desta página. A partir destes dados, é procurado o endereço do frame f que foi mapeado pela página p. Finalmente, um endereço efetivo (na memória física) é gerado através da soma de f e d.

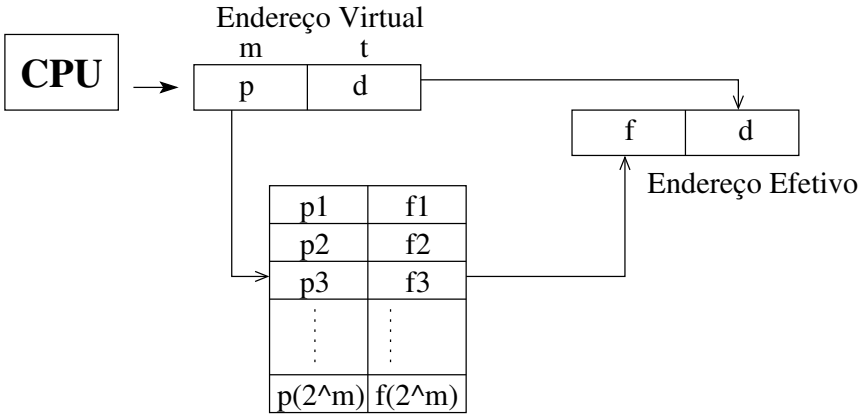

Figura 2.5: Paginação com um nível de tabela de páginas.

Aqui, a perda de memória devido a fragmentação interna geralmente não se torna um grande problema. A perda de memória para fragmentação interna em um sistema paginado pode ser definida em, aproximadamente, a metade do tamanho de um frame por segmento de memória alocado. Considerando que cada segmento de memória pode ter dezenas, centenas, ou até milhares de frames, o desperdício de memória é realmente pequeno, e paginação torna-se uma solução bastante eficiente.

Para tornar menos custosa a pesquisa na tabela de páginas pela página solicitada, existe a possibilidade de fazer uma implementação multi-nível desta estrutura, onde os primeiros níveis de tabelas apontam para outras tabelas e assim sucessivamente até que o último nível aponte para o *frame* desejado. A figura 2.6 demonstra o funcionamento de uma tabela de páginas de dois níveis. Neste caso, o endereço virtual é divido em três partes, um índice para o primeiro nível da tabela de páginas p, outro índice para o segundo nível q, e o deslocamento dentro do frame apontado d. A posição apontada

na tabela de páginas do primeiro ní vel indica em qual tabela de páginas do segundo ní vel deve ser procurado o frame *f* em questão. Após isso, o endereç o do frame requeridoé descoberto, e a montagem do endereç o efetivoé feita do modo habitual (*f* + *d*).

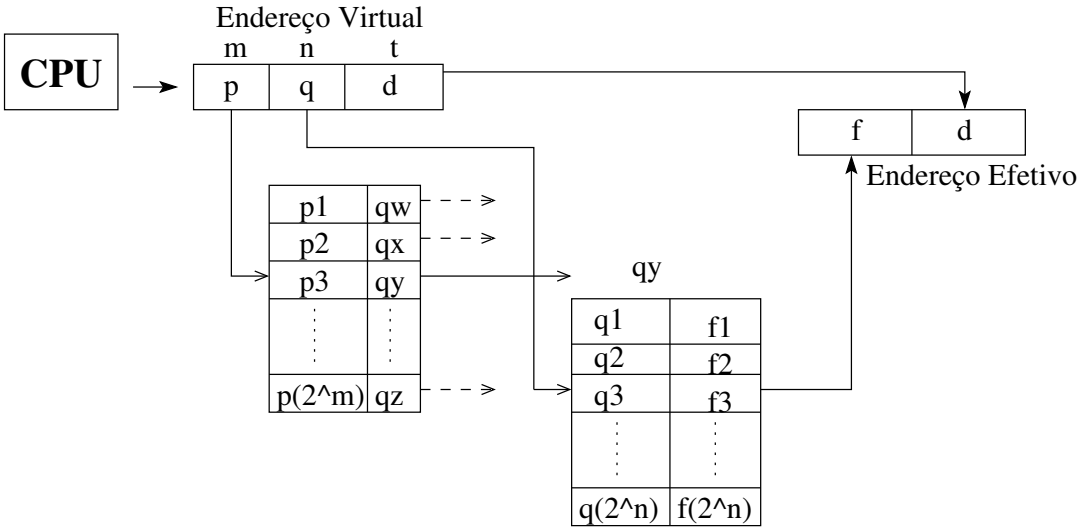

Figura 2.6: Paginação com dois ní veis de tabelas de páginas.

Embora o uso de tabela de página multi-ní vel reduza o tempo necessário para tradução de endereç os, este tempo aindaé muito alto para que se faç a esta pesquisa toda vez que uma referência de memória é realizada. Levando-se em consideração que um mesmo programa geralmente acessa as mesmas páginas, as MMU's modernas disponibilizam uma (ou mais de uma) memória associativa que armazena o resultado das últimas traduções realizadas. Esta memória é denominada *Translation Lookaside Buffer (TLB)*.

Uma das principais vantagens oferecidas pela paginação é o fato de ela proporcionar um ambiente bastante favorável à realização de *SWAP* de memória. *Swap* é um mecanismo utilizado para implementar sistemas de *memória virtual*, ou seja, permitir que o processador tenha a ilusão de possuir mais memória do que realmente tem. Este sistema utiliza um dispositivo secundário de armazenamento, que é maior e mais barato que o utilizado como memória principal (geralmente um disco magnético), para manter *frames* de memória que não estão sendo utilizados. Este controle é feito através de um *bit*

de controle na tabela de páginas, chamado *bit* de presença. Quando um *frame* não está na memória (o valor deste *bit* é zero) e uma referência a ele é realizada, um outro *frame* que está na memória principal é escolhido para ser copiado de volta para o dispositivo secundário, permitindo que o *frame* solicitado seja trazido do para a memória principal.

Outra facilidade que o esquema de paginação pode proporcionar é a proteção das páginas de memória, ou seja, é possível verificar, em tempo de acesso, se a página acessada pode realmente ser lida, escrita ou executada (dependendo da instrução que a referencia). Qualquer violação destas permissões, o fato de um processo tentar acessar uma página não mapeada em seu espaço de endereçamento, ou ainda o acesso a uma página que não possui o seu *bit* de presença ligado pode gerar um *Page Fault*[11].

As rotinas de tratamento de um *Page Fault* devem tratar os erros de modo que ou o processo possa continuar executando após o tratamento (como por exemplo, quando o *Page Fault* é gerado pelo acesso a uma página de memória que está em disco, devido ao mecanismo de *swap* de memória), ou terminando o mesmo (falha de proteção).

2.3.2 Segmentação e Segmentação Paginada

Embora tenham sido propostos mais recentemente que a Paginação, os esquemas de tradução de endereços Segmentação[DEN 65] e Segmentação com Paginação [ORG 72] não são tão largamente utilizados, mesmo existindo em importantes arquiteturas (como os processadores da família x86 da Intel). Neste caso, os processadores Intel podem ser configurados para que funcionem com paginação pura.

O esquema de segmentação divide a memória física em segmentos cujos tamanhos são definidos pelo usuário em tempo de alocação. Para possibilitar a tradução de endereços, são armazenados (em registradores ou em memórias associativas) o endereço inicial e o final de cada segmento. O mecanismo de segmentação é demonstrado na figura 2.7. Quando uma referência à memória é realizada, o endereço gerado pelo processo é somado ao endereço inicial do segmento, o resultado é testado para verificar se não

[11] Um *Page Fault* pode ser gerado tanto por *hardware* quanto por *software*, dependendo da implementação

ultrapassou o limite, caso não ultrapasse, o acesso à memória física é realizado utilizando o endereço resultante da soma, caso contrário, uma falha é gerada.

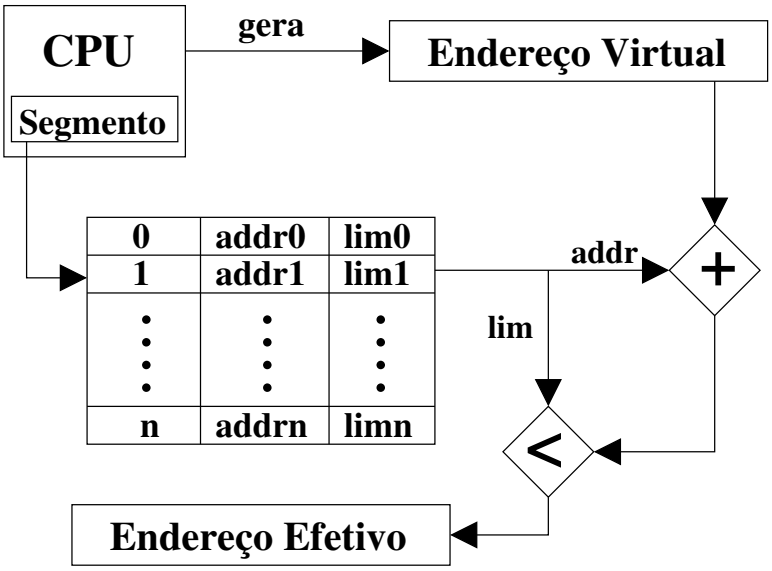

Figura 2.7: Esquema de segmentação de memória.

O esquema de segmentação paginada combina os esquemas de segmentação e paginação. Nestes casos, a utilização de paginação deve-se ao fato de, como citado anteriormente, este esquema facilitar a realização de *Swap* de memória. O funcionamento deste esquema é demostrado na figura 2.8. Quando é gerada uma instrução que acessa a memória, o endereço utilizado por esta instrução passa por duas etapas de tradução. Primeiramente, o endereço lógico é traduzido pelo esquema de segmentação, como pode ser visto em mais detalhes na figura 2.7, gerando um endereço intermediário. Depois disso, este endereço intermediário é traduzido pelo mecanismo de paginação, conforme descrito pela figura 2.5, que gera o endereço físico a ser acessado.

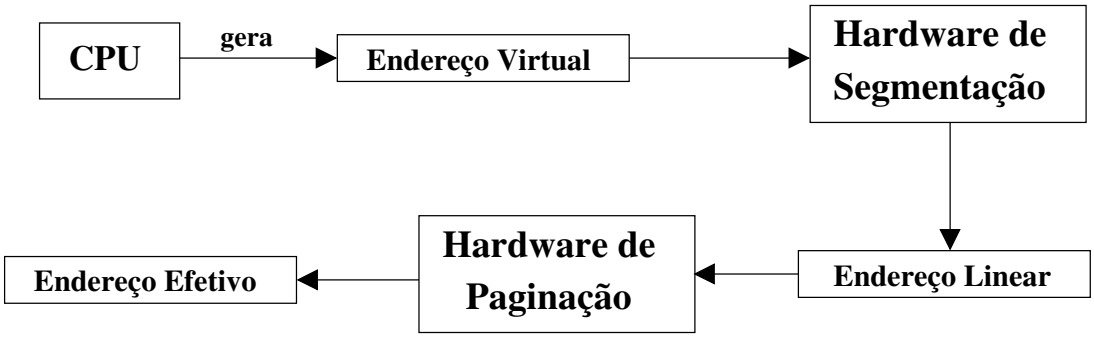

Figura 2.8: Esquema de segmentação paginada de memória.

Capítulo 3

Gerente de Memória do EPOS

Como dito anteriormente, este trabalho abrange dois aspectos de gerência de memória: gerência de memória no nível de sistema e no nível de usuário[1]. Este capítulo tratará do primeiro aspecto.

O principal objetivo deste capítulo é demonstrar como, através do uso da *Application Oriented System Design* (AOSD) [dMF 01], foi modelado e implementado um gerente de memória orientado à aplicação. O grande trunfo deste trabalho (como será apresentado) encontra-se na separação das partes dependentes das independentes de arquitetura, bem como na transparência do esquema de gerência de memória, o que permite que uma aplicação que venha a ser desenvolvida para o EPOS[dMF 02] não tenha a necessidade de saber que tipo de *hardware* ou estratégia de gerência de memória está sendo utilizada.

3.1 Considerações sobre Metodologia e Plataforma Utilizadas

Esta seção fará considerações referentes ao Projeto de Sistemas Orientados à Aplicação (PSOA) [dMF 01], mais especificamente aos Sistemas Operacionais Orientados à Aplicação (SOOA) [dMF 01] e ao *Embedded Parallel Operating System*

[1] Entende-se por usuário um programa "usuário" do Sistema Operacional.

(*EPOS*) [dMF 01, dMF 02].

3.1.1 Sistemas Operacionais Orientados à Aplicação

Sistemas Operacionais Orientados à Aplicação (*Application-Oriented Operating Systems*) é um conceito introduzido por Fröhlich em [dMF 01]. Em tal obra, Fröhlich propõe uma nova metodologia para projeto de sistemas que procura, através da combinação de várias técnicas de engenharia de *software*, atingir um alto grau de configurabilidade que permita a geração de sistemas específicos para determinadas aplicações, garantindo um elevado nível de re-usabilidade ao sistema, portabilidade às aplicações e baixo *overhead*.

Para isto, Fröhlich apresenta, ainda na mesma obra, a metodologia de projeto multi-paradigma *Application-Oriented System Design*. Tal metodologia decompõe o domínio em questão (Sistemas Operacionais, por exemplo) em *famílias de abstrações* independentes de cenário[2] que, através da re-usabilidade, podem gerar várias instâncias do mesmo sistema. A configurabilidade pode ser atingida através do uso de *aspectos de cenário* e *configurable features*. Estas famílias constituem *componentes de software*, e o conjunto destes componentes geram um *framework* que é o sistema em sí. Cada componente deste *framework* é acessado através de uma *Interface Inflada*, que garante a portabilidade dos artefatos de *software* que o utilizam.

3.1.2 *EPOS*

O *EPOS* (*Embedded Parallel Operating System*)[dMF 02] é o sistema operacional desenvolvido inicialmente por Fröhlich para validar os conceitos desenvolvidos em sua tese. O *EPOS* foi concebido como solução para sistemas dedicados, incorporando aplicações na área de sistemas de controle/automação, robôs e sistemas embutidos em geral, além de sistemas para ambientes de computação paralela e distribuída.

[2]Entende-se por cenário um ambiente para o qual o *EPOS* será gerado, ou seja, o conjunto formado pela plataforma e pela aplicação a ser utilizada.

Sua primeira implementação visava utilizá-lo como sistema operacional dedicado para os nós de um *cluster* de computadores pessoais baseados na arquitetura ix86 e interconectados por um sistema de comunicação de alto desempenho (*MYRINET*). Atualmente, os alunos do *LISHA* (Laboratório de Integração *Software*/*Hardware*) da *UFSC* (Universidade Federal de Santa Catarina), coordenados pelo Prof. Dr. Fröhlich, continuam o seu desenvolvimento e realizam suas pesquisas no âmbito deste sistema.

Basicamente, o *EPOS* é composto de três tipos de famílias: *Abstrações*, *Mediadores* e *Aspectos*. As Abstrações são famílias que englobam características, funcionalidades e estruturas independentes de arquitetura (cenário). Elas são amplamente re-usáveis e constituem a maior parte dos componentes do sistema que uma aplicação necessita. Os Mediadores são abstrações dependentes de arquitetura. Eles são responsáveis por implementar as funcionalidades que as abstrações e/ou aplicações necessitam. As famílias de Mediadores precisam obedecer um contrato de interface rígido (Interface Inflada), de modo a garantir a portabilidade do sistema. As famílias de Aspectos são utilizados pelo *EPOS* para oferecer configurabilidade ao sistema.

Além do uso de Aspectos, as famílias podem utilizar *Configurable Features*[3] para permitir sua configuração. O uso de Aspectos e *Configurable Features* permite que o *EPOS* atinja um elevado grau de configurabilidade sem a inserção de muita "sujeira" no código-fonte.

Além dos componentes, o *EPOS* é ainda constituído de um conjunto de ferramentas que permitem que o usuário (programador da aplicação) configure o sistema operacional conforme suas necessidades.

3.2 Famílias de Abstrações

Seguindo os conceitos da metodologia *Projeto de Sistemas Orientados à Aplicação*, o gerente de memória do *EPOS* foi modelado como um conjunto de famílias de abstrações. Isto foi feito após uma apurada análise de domínio, que focou-

[3] A tradução para *Configurable Feature* é *Característica Configurável* mas, como este termo foi introduzido em inglês na obra de Fröhlich, foi decidido por não traduzi-lo no decorrer deste texto.

se em gerência de memória para sistemas embutidos, seguida de uma decomposição de domínio, gerando um conjunto de famílias, aspectos e opções de configuração (*Configurable Features*) pertinentes ao gerente de memória.

O domínio foi abstraído em três famílias, sendo duas *abstrações* independentes de arquitetura (*Address_Space* e *Segment*), e um *mediador* de *hardware* (*MMU*).

Este talvez seja o domínio mais dependente de arquitetura existente, já que a diversidade de mecanismos de gerência de memória é grande. Portanto, foi uma tarefa exemplarmente difícil encontrar denominadores comuns entre os mais variados esquemas e implementações.

Contudo, embora não se tenha chegado a um meio de abstrair totalmente o os esquemas de gerência de memória (*Flat*, *Paginação*, *Segmentação* e *Segmentação Paginada*), foi observado que é possível definir um modo comum de requisição de memória, independente das peculiaridades do sistema. Tal padronização nos procedimentos de requisição de memória levou em conta o fato de que sempre que um processo requer uma quantidade de memória, o mesmo não precisa especificamente solicitar, por exemplo, uma página, basta solicitar um segmento de memória de um determinado tamanho, ficando o procedimento efetivo de alocação a cargo das abstrações que "sabem" que tipo de esquema de gerência está sendo utilizado.

Esta idéia inspirou-se, além das modelagens iniciais do *EPOS*, no trabalho de Piccoli e Ávila [PIC 96], que implementou o sistema de gerência de memória do *ABOELHA*, um *nano-kernel* para sistemas distribuídos, desenvolvido também no LISHA na década de 1990.

Toda vez que um processo requer memória, ele solicita a criação de um *segmento*, o que é realizado por um dos membros da família *Segment*. Sempre que um segmento é instanciado, ele solicita à *MMU* que lhe seja alocada uma certa quantidade de memória. A *MMU* procede a alocação de memória levando em consideração as características da arquitetura sendo utilizada. Para que o processo possa finalmente utilizar o segmento criado, o mesmo necessita anexar este segmento ao seu *espaço de endereçamento*, que é representado pela família *Address_Space*. O membro desta família

tem conhecimento do esquema de gerência de memória sendo utilizado, e solicita à *MMU* que o segmento em questão seja anexado a sí .

O conceito de *memória virtual* não é levado em consideração aqui. Este gerente de memória entende que um processo requisita memória pura e simplesmente. Se esta memória é gerenciada utilizando conceitos relacionados a memória virtual, isto não precisa ser transparente para o usuário (programador de aplicação). Todos estes detalhes podem ser "escondidos" pela implementaç ão especí fi ca para cada arquitetura (*MMU*). O uso ou não de memória virtual pelo sistema operacional pode constituir uma *confi gurable feature* desta *MMU* especí fi ca caso a arquitetura ofereç a suporte para tal.

A fi gura 3.1 apresenta um diagrama com as famí lias que constituem o componente Gerenciador de Memória do *EPOS*. A seguir estas três famí lias serão apresentadas, juntamente com os aspectos e confi gurabilidades defi nidas para elas.

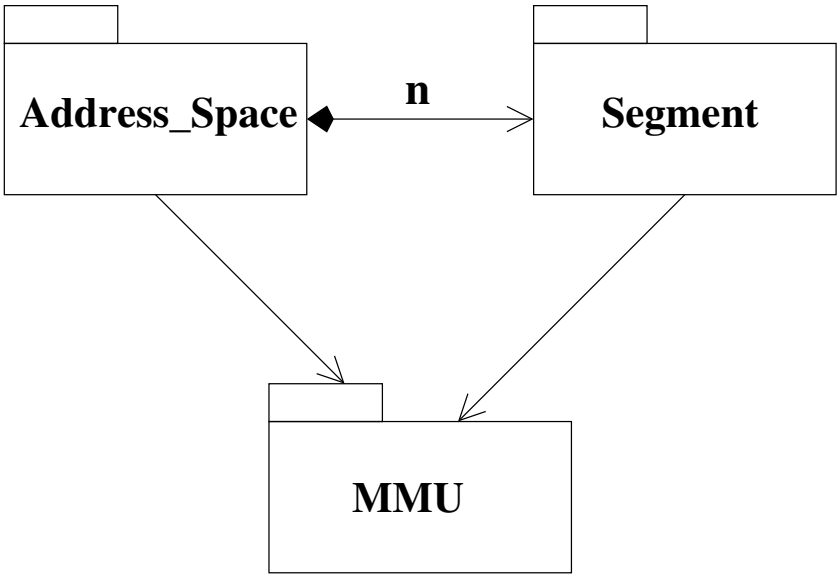

Figura 3.1: Diagrama de famí lias.

3.2.1 Famí lia *Address_Space*

Os membros desta famí lia são responsáveis por defi nir o esquema de mapeamento de endereç os lógicos e fí sicos do sistema operacional. Normalmente, esta

definição leva em consideração o tipo de *hardware* que uma determinada arquitetura apresenta. Contudo, neste gerente de memória, é possível em alguns casos desvincular estes conceitos. Naturalmente não faz sentido utilizar, por exemplo, o membro *Segmented_AS* desta família em uma arquitetura que não disponibiliza *hardware* para segmentação de memória. Contudo, é possível utilizar o membro *Flat_AS* em uma arquitetura cuja MMU suporta paginação e disponibilizar para o programador da aplicação a opção de usar este recurso. Neste exemplo, é possível prover um espaço de endereçamento *flat* com um mecanismo de proteção de memória, bastando para isto garantir que os endereços lógico e físico de cada *frame* sejam coincidam.

Um diagrama contendo os membros desta família pode ser visto na figura 3.2.

3.2.1.1 Membros da família *Address_Space*

Da análise de domínio foram extraídos quatro esquemas de mapeamento de endereços relevantes, que foram agrupados na família *Address_Space*. Estes membros são mutuamente exclusivos, ou seja, somente um pode ser utilizado por instância do sistema operacional. Tais membros são descritos abaixo:

***Flat_AS*:** o membro *Flat_AS* desta família é responsável por garantir a implementação de um esquema de gerência de memória onde todo o espaço de endereçamento do sistema, incluindo memória física e I/O, é entregue a um único processo e não há tradução de endereços. A princípio, principalmente em arquiteturas mais simplificadas, este membro não possui funcionalidade alguma, ou seja, não é necessário que se implemente nada.

Contudo, no *EPOS* foi optado por fornecer um espaço de endereçamento *flat* com a possibilidade de efetuar proteção de regiões de memória se a arquitetura utilizada suportar paginação. Deste modo, este membro implementa algumas funcionalidades que têm por objetivo garantir que todo mapeamento de memória possui endereços físico e lógico iguais. Vale ainda observar que, através do uso dos recursos de meta-programação estática da linguabem de programação *C++*, este mem-

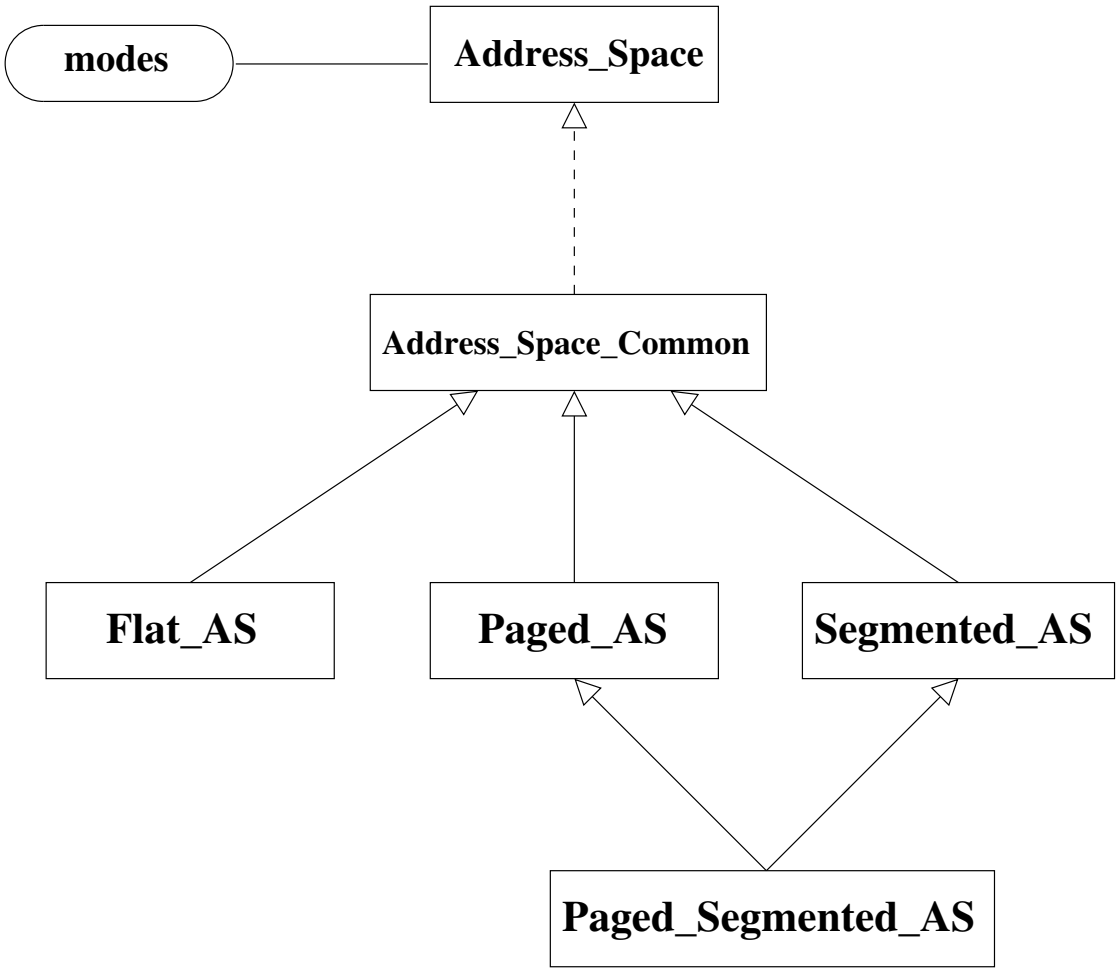

Figura 3.2: Diagrama de classes da famí lia*Address_Space*.

bro não agrega código algum ao sistema quando compilado para uma arquitetura simples.

Se existir suporte a paginação, o *Address_Space* implementa o primeiro ní vel de tabelas de páginas, e a famí lia*Segment* é responsável pelo segundo ní vel desta estrutura. O funcionamento do sistema de paginação quando utilizado no modo *flat* é idêntico ao funcionamento deste mecanismo no modo paginado que será descrito a seguir, com a única resalva de que, aqui, os mapeamentos precisam manter endereç os lógico e fí sico idênticos.

***Paged_AS*:** este membro da famí lia*Address_Space* implementa um espaç o de endereç a-

mento paginado. Neste caso, o *Paged_AS* é responsável por implementar o primeiro nível de tabelas de páginas, ou seja, o *Page Directory* (Diretório de Páginas). O segundo nível da estrutura, as *Page Tables* (Tabelas de Páginas), são implementadas pela família *Segment*.

Este modelo permite que, logicamente, o *EPOS* trate o mecanismo de paginação como um sistema de dois níveis de tabelas de páginas, mas nada impede que, fisicamente, o sistema opere com MMU's que utilizem um número diferente de níveis (um ou mais que dois). Estas peculiaridades de cada arquitetura são resolvidas no mediador *MMU* que, como será descrito adiante, é responsável por abstrair as características de cada processador.

***Segmented_AS* e *Paged_Segmented_AS*:** embora poucas arquiteturas disponibilizem este tipo de *hardware* hoje em dia, os esquemas de *segmentação* e *segmentação paginada* foram modelados no *EPOS*. Isto se deve ao fato destes modelos serem utilizados pela arquitetura ix86, que é bastante difundida.

Contudo, mesmo sendo para esta arquitetura um dos portes deste gerente de memória, estes membros da família *Address_Space* não foram implementados, já que existe a possibilidade de emular paginação pura no processador em questão. Todavia, existe a possibilidade de que, caso interesse a alguma aplicação específica, se faça a implementação de tais famílias de modo prático e sem comprometer a portabilidade do sistema.

3.2.1.2 Interface Inflada

Uma das características da família *Address_Space* é o fato de que, mesmo com todas as diferenças existentes, todos os membros modelados possuem a mesma interface, ou seja, a interface da família é igual à interface de todos os membros da mesma. Este tipo de interface se classifica como *Uniforme*.

A seguir serão apresentados os principais métodos dos membros desta família:

Log_Addr attach(Segment & seg, Log_Addr addr): este método anexa o segmento referenciado por *seg* no endereço indicado por *addr* e retorna o endereço lógico onde o segmento foi mapeado ou outro valor em caso de erro. Este método falha caso o membro da família que está sendo utilizado seja o *Flat_AS* e o parâmetro *addr* não for igual ao endereço físico do segmento *seg*.

void detach(Segment & seg): Este método desanexa o segmento *seg* do *Address_Space*.

Phy_Addr physical(Log_Addr address): Este método retorna o endereço físico referente ao endereço lógico *address*.

3.2.2 Família *Segment*

Esta família implementa o conceito lógico de segmento de memória. Cada segmento representa para o usuário uma porção de de memória física ou uma região de endereçamento onde existe um dispositivo de entrada e saída (I/O). Cada segmento está organizado de acordo com as características de cada um dos membros. Estas peculiaridades de cada tipo de segmento serão descritos nas seções subsequentes.

Cada segmento agrega uma estrutura implementada pelo mediador da *MMU* que se chama *Chunk*. O *Chunk* é responsável por abstrair as peculiaridades de cada arquitetura e por manter, juntamente com a *MMU*, uma interface constante, de modo a não tornar necessário que as *abstrações* do sistema conheçam características específicas de cada implementação. A estrutura do *Chunk* será melhor explicada na seção que trata do mediador da *MMU*.

Os aspectos identificados como importantes para esta família são apresentados juntamente dos membros da mesma na figura 3.3.

Uma importante função dos segmentos é o compartilhamento de memória entre processos. Isto é implementado através do aspecto *Shared*, que garante a integridade no acesso a estes segmentos, tratando-os como regiões críticas, evitando erros devido à concorrência no acesso. Assim sendo, qualquer processo que conheça o segmento pode compartilhá-lo.

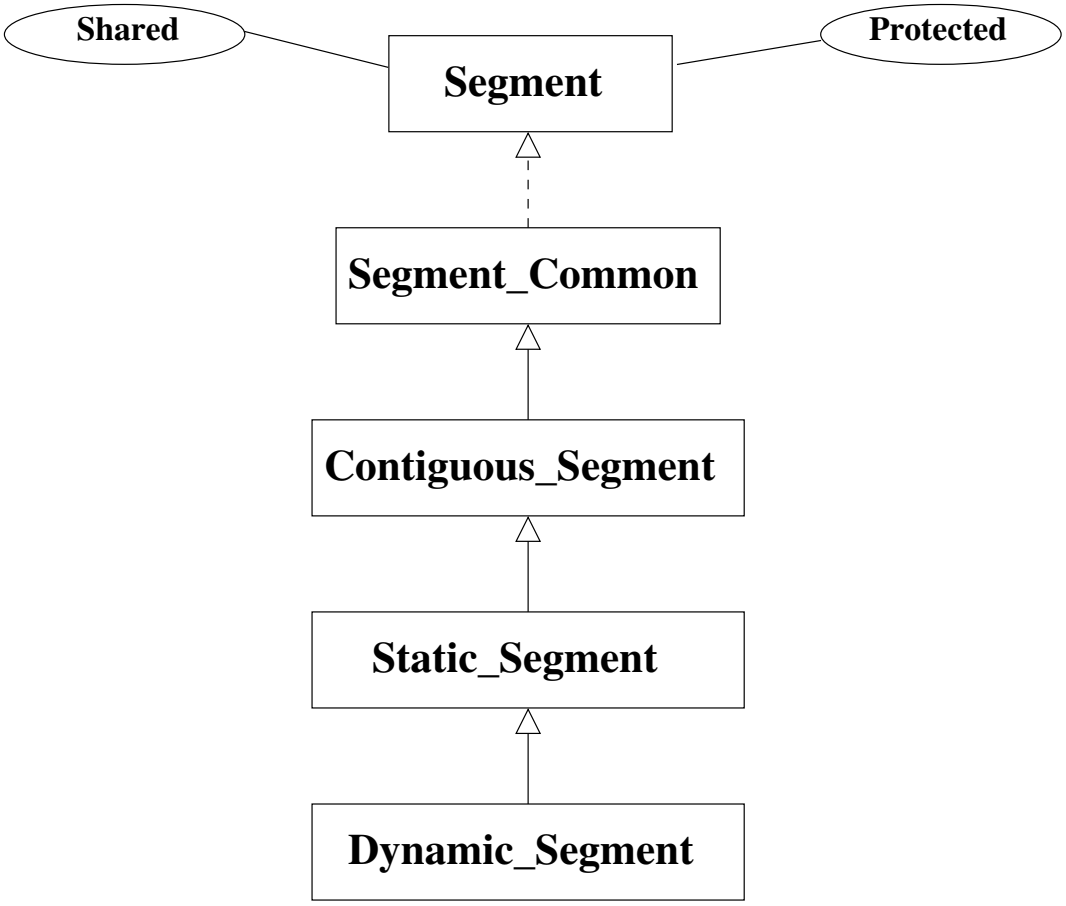

Figura 3.3: Diagrama de classes da família *Segment*.

O aspecto *Protected* é responsável por implementar as características de proteção de cada segmento. Ele controla as permissões de escrita, leitura e execução de cada instância de membros desta família.

3.2.2.1 Membros da família *Segment*

De acordo com os estudos realizados, foram identificados três possíveis membros para esta família. Já que não há conflitos entre esses membros, eles não são mutuamente exclusivos, ou seja, podem ser utilizados conjuntamente em qualquer instância do sistema operacional. A única restrição existente para alguns membros diz respeito à nessecidade de suporte a tradução de endereços, como será descrito abaixo:

***Contiguous_Segment*:** Este é o membro mais simples da família *Segment*, e também o

mais importante. Sua importância se dá ao fato de não agregar tanto código ao sistema (devido à sua simplicidade) e, principalmente, por ser o único que não requer a presença de *hardware* para tradução de endereços para operar, sendo este o caso mais comum em sistemas embutidos.

Um *Contiguous_Segment* ou, em português, segmento contíguo, representa um bloco de memória física onde todas as unidades de armazenamento (*bytes*, palavras ou páginas) são adjascentes. Outra característica deste tipo de segmentoé que, embora possa ter seus *flags* de proteção alterados a qualquer momento, ele não pode ser redimensionado, já que tal funcionalidade acarretaria um *overhead* de processamento inaceitável, além do fato de que apenas em raros casos seria possível encontrar blocos contíguos de memória para satisfazer as requisições de re-alocação sem que isto implicasse na realização de cópia de memória.

Mesmo em sistemas onde há suporte para tradução de endereços, o uso deste membro pode ser mais indicado, já que, na maioria destes sistemas, a localidade espacial dos dados na memória pode impactar de modo significativo na performance [HEN 98].

***Static_Segment*:** Para o uso deste membro é necessário a existência de um esquema de tradução de endereços. Este segmentoé estático, ou seja, não pode ser modificado após criado, nem em seus *flags* de proteção, nem em seu tamanho ou mapeamento. O fato de ele levar em conta a existência de uma MMU implica em maior quantidade de código, o que pode vir a ser inconveniente em alguns casos. Um bom exemplo de utilidade para este tipo de segmento é para segmentos de código, que não são modificados em tempo de execução, são apenas criados e destruídos.

***Dynamic_Segment*:** O *Dynamic_Segment* é o membro mais completo da família. Ele, assim como o *Static_Segment*, requer a presença de uma MMU mas, diferentemente de seu "irmão", permite que sua configuração seja alterada dinamicamente (em tempo de execução). Este membro é a melhor opção para segmentos onde é realizado alocação dinâmica de memória ou para segmentos de pilha (*Stack*), já

que eles tendem a variar de tamanho de forma bastante aleatória, e a opção de re-dimensionamento pode oferecer uma gerência mais racional dos recursos do sistema.

3.2.2.2 Interface Inflada

O projeto desta família seguiu um mecanismo incremental, ou seja, cada membro da família implementa as funcionalidades de outro membro e mais algumas, de modo que todo membro é subclasse de outro. A interface deste tipo de família é chamada *Incremental*, e é composta dos métodos do membro de mais baixo nível na hierarquia de classes da família. Abaixo são descritos os métodos da interface da família *Segment*:

Segment(Size bytes, Flags flags): Este construtor realiza a interação com o mediador *MMU* (que será descrito adiante) para alocar um bloco de memória com o tamanho indicado pelo parâmetro *bytes*, com as configurações indicadas por *flags*. Este deve ser o construtor mais utilizado para alocar memória física, já que, neste tipo de alocação, geralmente não é importante o endereço **f**sico do segmento. Quando se deseja alocar memória física em um endereço específico ou mapear dispositivos de I/O, deve-se utilizar o construtor descrito no ítem seguinte.

Segment(Phy_Addr addr, Size bytes, Flags flags): Este construtor procura alocar um bloco de endereços com o tamanho indicado pelo parâmetro *bytes* que inicie no ponto indicado por *addr*. A utilização deste construtor é mais indicada para mapear dispositivos de entrada e saída.

Size size(): Este método retorna o tamanho do segmento em *bytes*.

MMU::Chunk * chunk(): Este método retorna um ponteiro para o *Chunk* que representa, fisicamente, o segmento de memória.

int resize(int amount): Este método altera a dimensão do segmento aumentando ou diminuindo seu tamanho se o valor do parâmetro *amount* for positivo ou negativo, respectivamente. Ele retorna *1* em caso de sucesso, *0* caso o tipo de segmento não suporte re-dimensionamento ou um valor negativo se ocorrer algum erro.

3.2.3 Famí lia*MMU*

Esta famí lia de mediadores tem por objetivo abstrair as peculiaridades de cada arquitetura. Esta famí lia, no que diz respeitoà interface de seus membros, possui um comportamento dissociado, já que cada membro abstrai as caracterí sticas inerentes de cada *hardware*. No entanto, é extremamente importante que todos os membros desta famí lia obedeç am rigidamente a um contrato de interface, afi m de garantir a portabilidade das *abstrações* e aplicações que fazem uso deste *mediador*. A estrutura desta famí liaé exibida na fi gura 3.4.

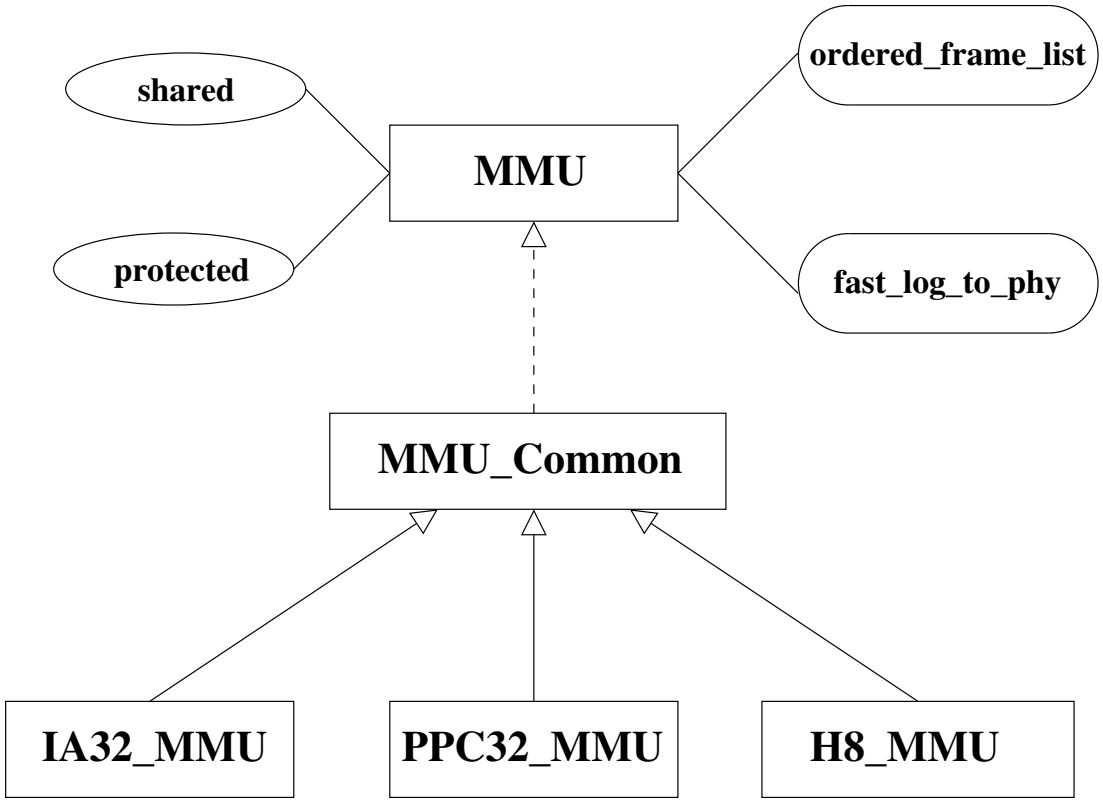

Figura 3.4: Diagrama de classes da famí lia*MMU*.

Nesta seção serão discutidos os três membros implementados deste mediador.

3.2.3.1 Membros da família *MMU*

Uma característica importante desta família é o fato de ela ser totalmente modulável, ou seja, para que outro membro (arquitetura) seja acoplada a esta família, basta que este membro herde a classe *MMU_Common* e obedeça à interface definida.

Para fins de teste, foram implementadas três membros para esta família, que são descritos a seguir:

***IA32_MMU*:** Esta implementação dá suporte ao uso de paginação. Embora a arquitetura *IA32* não possibilite o desligamento do *hardware* de segmentação, é possível configura-lo de modo a emular um sistema de paginação pura. O modo como isto foi feito e como o mecanismo de paginação foi abstraído pode ser melhor entendido na seção que trata da Interface Inflada desta família.

***PPC32_MMU*:** Para esta arquitetura foram feitas duas implementações, sendo uma com e outra sem suporte a paginação. A escolha sobre qual das implementações usar é feita através de uma *Configurable Feature*.

***H8_MMU*:** Este é o membro mais simples da família *MMU*, já que é a única das arquiteturas estudadas que não possui *hardware* de gerência de memória.

3.2.3.2 Interface Inflada

Como dito anteriormente, é extremamente importante garantir que todos os membros da família *MMU* respeitem uma interface comum. Para tanto, tal interface foi modelada de modo *Uniforme*, mesmo sabendo que os membros da família possuem um comportamento dissociado. Esta escolha deve-se principalmente ao fato de o membro *Flat_AS* da família *Address_Space* considerar a possibilidade de uso de paginação, afim de garantir proteção de memória em um espaço de endereçamento *flat*. Assim sendo, é necessário esconder o mecanismo real de gerência de memória de arquiteturas desprovidas de MMU sob um pseudo-mecanismo de paginação.

Para garantir a portabilidade, cada membro da famí lia deve implementar, além das funcionalidades da interface, um conjunto de estruturas de dados. Abaixo estão relacionadas algumas destas estruturas:

***Chunk*:** Esta estrutura é responsável por implementar o canal de comunicação entre as famí lias *Segment* e *MMU*. Cada segmento de memória tem seu equivalente fí sico representado por um *Chunk*. Esta classe é também responsável por abstrair o modo real de alocação de memória. Ele mantém uma interface rí gida eé utilizado apenas pelas *abstrações* de gerência de memória.

***Page_Table*:** Esta estrutura implementa o mecanismo de paginação. Sendo ela também responsável pelas rotinas de mapeamento de memória. Esta classe é utilizada pelo *Chunk* mesmo quando não há paginação, sendo então aqui implementado um possí - vel mecanismo de pseudo-paginação.

***Page_Directory*:** Este membro implementa o primeiro ní vel lógico de tabelas de páginas. Quando não há uso de paginação, sua implementação pode ser vazia. Esta classe é utilizada pelos membros *Flat_AS* e *Paged_AS* da famí lia *Address_Space*.

***Segment_Table*:** Esta estrutura não chegou a ser implementada durante este trabalho, já que não foi utilizada nenhuma arquitetura onde fosse necessário o uso do *hardware* de segmentação. Contudo, todas as implementações desta famí lia possuem implementações vazias desta estrutura, já que ela é utilizada pelos membros *Segmented_AS* e *Paged_Segmented_AS* da famí lia *Address_Space*.

Além destas estruturas, a interface inflada ainda oferece alguns métodos para serem utilizados tanto pelas abstrações do sistema quanto por programas do usuário, embora seja recomendável o uso das abstrações de mais alto ní vel por esteúltimo. Abaixo são descritas estas funções:

Phy_Addr alloc(int n): Este método aloca *n* páginas contí guas de memória e retorna o endereç o **f** sico da primeira destas páginas. Em arquiteturas sem suporte a paginação, uma página é considerada como sendo um *byte*, sendo este o principal conceito do mecanismo de pseudo-paginação.

void free(Phy_Addr addr, int n): Este método re-coloca na lista de frames livres as *n* primeiras páginas (ou *bytes*, no caso de pseudo-paginação) a partir do endereço *addr*.

Phy_Addr physical(Log_Addr addr): Este método retorna o endereço físico equivalente ao endereço lógico *addr*.

void flush_tlb(): Este método invalida todas as entradas do *buffer* de tradução de endereços (*TLB - Translation Lookaside Buffer*).

void flush_tlb(Log_Addr addr): Este método identifica o endereço da página que contém o endereço *addr* e invalida a entrada desta página na TLB.

3.3 Implementação

Este capítulo pretende apresentar detalhes da implementação das famílias modeladas por este trabalho. Também serão apresentadas as principais características das arquiteturas estudadas durante o desenvolvimento, dando especial atenção para a *IA32* (arquitetura de 32 bits da Intel), que foi a utilizada na implementação do protótipo fruto deste trabalho.

3.3.1 Considerações de Implementação

Antes de apresentar a implementação deste trabalho, serão descritas algumas características da linguagem de programação *C++*[STR 97] e do compilador para esta linguagem desenvolvido pela *GNU*, o *GCC* (*GNU* Compiler Collection)[HTT 04a].

3.3.1.1 Metaprogramação Estática

O conceito de *metaprogramação estática* é amplamente utilizado pelo *EPOS* de modo a garantir maior configurabilidade e a geração de um código mais eficiente. A linguagem *C++* oferece este recurso através dos *gabaritos* (*templates* em inglês), que, nesta linguagem, são resolvidos de maneira estática, ou seja, em tempo de

compilação. Assim sendo, o código binário final gerado pelo compilador, não vai conter segmentos de código que não são utilizados, como é mostrado na figura 3.5. Neste caso, se o valor do parâmetro *allocs* for *AOFF_NC*, o compilador gerará código apenas para a função *aoff_nc_malloc()*, deixando de "engordar" o binário gerado com funções desnecessárias.

```
template<Alloc_System allocs = AOFF_NC, bool resizeable = true>
class Memory_Allocator {
  ...
    void * malloc(unsigned int bytes) {
      ...
        switch(allocs) {
            case AOFF_NC:     return aoff_nc_malloc(bytes); break;
            case AOFF_C:      return aoff_c_malloc(bytes); break;
            case BINARY_BUDDY: return binary_buddy_malloc(bytes); break;
        }
    }
};
```

Figura 3.5: Fragmento de código onde ficam claros os benefícios que a metaprogramação estática traz ao sistema.

3.3.1.2 Otimização de Código

As rotinas de otimização de código do compilador (no caso o *GCC*) são responsáveis por, quando possível, tornar mais eficiente o programa escrito pelo programador. No sistema *EPOS* todas as compilações são feitas utilizando o nível 2 de otimização (-O2), que realiza vários tipos de otimização de código, porém sem realizar *loop unrolling* (desenrolar laços), o que geralmente aumenta consideravelmente o tamanho do código binário gerado.

No *EPOS* o *GCC* ainda é responsável por eliminar chamadas desnecessárias de funções, tornando algumas delas *inline*. No caso do exemplo 3.5, o meca-

nismo de metaprogrmação ignora os métodos que nunca são chamados, porém não eliminam o teste (*switch*) nem a nova chamada de função. Contudo, o otimizador de código elimina este teste e torna a função *malloc() inline*, ou seja, sempre que algum programa chamar esta função o código gerado será uma chamada para a função *aoff_nc_malloc()*.

3.3.2 As Abstrações do *EPOS*

Os recursos descritos na seção anterior foram largamente utilizados neste trabalho na implementação de suas abstrações (*Address_Space* e *Segment*). Como dito no capí tulo 3, estas abstrações são responsáveis apenas por oferecer ao usuário do sistema operacional (programador da aplicação) uma interface mais amigável para utilizar os recursos do sistema. Também foi dito que toda a implementação de funções e estruturas de dados utilizados pelo gerente de memória estão no mediador da *MMU*. Assim sendo, é necessário que o uso destas abstrações não impliquem em *overhead* na realização das tarefas do gerente de memória. Este problema é resolvido tornando alguns métodos *inline*, eliminando, assim, chamadas desnecessárias de funções.

3.3.3 O Mediador da *MMU*

Como já abordado, toda a implementação do gerente de memória do sistema é de fato realizada no mediador da *MMU*. Aqui será descrito como foi realizada a implementação deste mediador. Serão descritas também as três arquiteturas estudadas, dando maior atenção à arquitetura *IA32*, que foi a utilizada na implementação do protótipo deste trabalho.

3.3.3.1 H8 - Microcontrolador de 8 bits da Hitachi

Esta arquitetura foi considerada devido a dois fatores: (1) a maioria das aplicações envolvendo sistemas embutidos no mundo utilizam microcontroladores simples, geralmente de 8 ou 16 bits, e (2) o LISHA possui um kit *RCX (Robotic Command eXplorer)* da *Lego*, que é equipado com um modelo de microcontrolador desta famí lia (o H8/300) [HIT].

Esta arquitetura é extremamente simples, e não oferece nenhum tipo de *hardware* para gerência de memória. Assim sendo, a implementação feita para este ambiente é também bastante simples. Outros fatores importantes que foram levados em consideração na implementação do gerente de memória deste sistema é sua pouca quantidade de memória (apenas 28 Kbytes) e seus limites de processamento.

3.3.3.2 PowerPC 405GP - Processador de 32 bits da IBM

A intenção inicial deste trabalho era utilizar a arquitetura PowerPC, porém, após o estudo das caracterí sticas do modelo disponí vel no LISHA, observou-se que este modelo possuia recursos que elevariam a complexidade da implementação. Assim, resolveu-se por implementar o primeiro protótipo para a arquitetura IA32, que já havia sido bem absorvida por pelos membros do laboratório.

O modelo 405GP faz parte da famí lia de processadores desenvolvidos especialmente para sistemas embutidos, sendo assim uma implementação mais simplificada quando comparado a modelos mais complexos como os utilizados nas estações de trabalho Machintosh, por exemplo. O fato é que a simplificação do *hardware*, neste caso, implica no aumento da complexidade do *software*. Uma destas simplificações é a auseência do mecanismo de paginação neste processador [IBM 01].

O 405GP oferece um conjunto de *TLB*'s (*Translation Lookaside Buffers*) que podem ser utilizados para implementar um mecanismo de paginação, mas todo este mecanismo tem que ser conduzido por *software*, inclusive a gerência das *TLB*'s. Outra caracterí stica da MMU deste processadoré o fato de ela permitir que se utilizem até 8 tamanhos diferentes de páginas, variando de 1 Kbyte até 16 Mbytes.

Além do mecanismo de paginação, este processador oferece o *Real-Mode Storage Attribute Control*. Este mecanismo possibilita proteção por área do espaç o de endereç amento. Cada uma destasáreas tem tamanho de 128 Mbytes, e pode conter atributos diferentes de proteção.

Contudo, todos estes mecanismos de proteção podem ser desligados, e a implementação realizada para esta arquitetura funciona do mesmo modo que a imple-

mentada para o H8. Tendo as duas implementações o mesmo código *C++*.

3.3.3.3 IA32 - Arquitetura de 32 bits da Intel

Devido ao fato de esta arquitetura possuir o sistema de paginação mais amigável entre as estudadas, ela foi escolhida como plataforma para o primeiro protótipo do *EPOS* com suporte a paginação.

Além do mecanismo de paginação (que pode ser desligado), existe ainda nesta arquitetura um mecanismo de segmentação, que não pode ser desligado. No entanto, mesmo sendo impossí vel desligá-lo, é possí vel fazer com que este mecanismo opere no modo *Flat*, que é descrito no próprio manual do processador [INT 97b] e foi implementado por Ávila e Piccoli em [PIC 96]. Isto é feito confi gurando todos os segmentos utilizados com base no endereç o 0 (zero) e limite de 4 Gbytes, ou seja, a CPU pode endereç ar todo o espaç o de endereç amento do processador independentemente do segmento que esteja acessando. Assim, todo endereç o linear gerado pelo mecanismo de segmentação é igual ao endereç o lógico gerado pela CPU, sendo a tradução de endereç os função exclusiva do mecanismo de paginação.

O IA32 implementa um mecanismo de paginação em 2 ní veis. Tal mecanismo é representado na fi gura 2.6. Sempre que um endereç o linear é gerado (proveniente do mecanismo de segmentação), este endereç o é automáticamente traduzido pelas tabelas de páginas que devem ser mantidas em memória pelo sistema operacional. O processador possui um registrador (o CR3) que aponta para o *Diretório de Páginas* que está sendo atualmente utilizado.

A implementação do gerente de memória do *EPOS* para este processador resultou em um componente de software que oferece às abstrações de mais alto ní vel do sistema e às aplicações suporte completo ao mecanismo de paginação de memória.

Outra caracterí stica do *software* implementado é a autoexpansibilidade da pilha. Sempre que um *Page_Fault* é gerado, o tratador desta exceção verifi ca o endereç o da falta e compara com o endereç o constante no *Stack-Pointer* (registrador ESP) do processo. Se fi car constatado que o *Page Fault* foi gerado devido ao crescimento da pilha, a

pilha é re-dimensionada e o processo continua sua execução normalmente. Se não for este o caso, levando-se em conta que este gerente de memória não implementou o mecanismo de *Swap*, o processo é abortado.

Capítulo 4

EPOS malloc - Um alocador dinâmico de memória configurável

Um alocador dinâmico de memória é um subsistema responsável por gerenciar requisições de alocação realizadas em tempo de execução, ou seja, alocar memória para a instanciação de objetos e/ou simplesmente alocação de memória para algum atributo, como realizado pelas funções *malloc()* e *free()* e operadores *new* e *delete* na linguagem *C++*.

Este tipo de alocação é realizada em um segmento de memória chamado de *Heap*, que é onde cada processo guarda seus dados alocados dinamicamente. A figura 4.1 demonstra a configuração lógica mais comumente utilizada para o posicionamento dos segmentos de um processo.

No *EPOS*, o alocador dinâmico foi modelado como uma *Utility*, ou seja, um utilitário que realizará as alocações dinâmicas de memória de acordo com as preferências do usuário. Outra vantagem deste tipo de modelagem é o fato de ainda existir a possibilidade de o usuário manipular o seu *heap* do modo que lhe convier, apenas criando um segmento de memória implementando rotinas que gerenciem a alocação e liberação de memória dentro deste segmento. A figura 4.2 apresenta a estrutura deste utilitário.

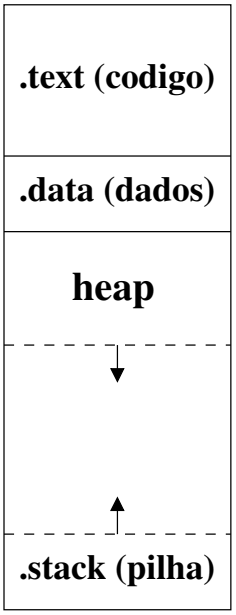

Figura 4.1: Estrutura lógica tradicional dos segmentos de um processo.

4.1 Aspectos e *Configurable Features*

Para garantir a configurabilidade do alocador de memória, foram extraídos da análise de domínio dois aspectos (*debug* e *stats*) e duas *configurable features* (*alloc* e *resizeable*). Estas características são melhor explicadas abaixo:

***alloc*:** Esta *configurable feature* é responsável por definir o mecanismo de alocação que será utilizado pelo alocador.

***resizeable*:** Esta característica define se o alocador dinâmico pode ou não requerer a alteração do tamanho do segmento de memória sendo utilizado. É importante salientar que, mesmo que esta *feature* esteja ativada, existe a possibilidade de o procedimento de alteração do tamanho de um segmento falhar se este segmento não for re-dimensionável (*Contiguous_Segment* ou *Static_Segment*, por exemplo).

***debug*:** Este aspecto tem por objetivo permitir a depuração do alocador, e só é utilizado durante procedimentos de implementação e/ou alteração do mesmo.

***stats*:** Este aspecto é utilizado para registrar estatísticas de análise desempenho tais como

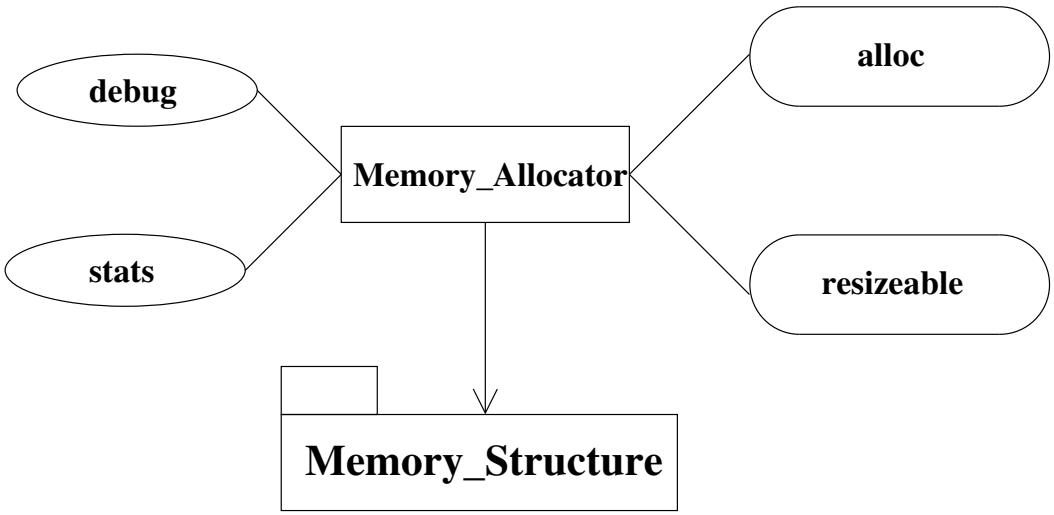

Figura 4.2: Estrutura do *EPOS malloc*.

fragmentação, velocidade e tipos (tamanho) de alocações realizadas. Este aspecto tem um impacto bastante representativo na performance do alocador, e deve ser utilizado com cautela.

4.2 Família *Memory_Structure*

Esta família é responsável por implementar um conjunto de estruturas de dados para gerência de memória tais como listas encadeadas e árvores. É bastante importante que estas implementações evitem gerar qualquer tipo de *overhead*, já que o desempenho do alocador de memória é um dos mais críticos de qualquer aplicação. A figura 4.3 apresenta a estrutura desta família.

Esta família obedece uma interface *uniforme*, já que, embora as estruturas de dados possuam comportamentos diferenciados, para o gerenciador de memória apenas são interessantes um pequeno conjunto de funcionalidades. Os membros desta família não implementam nenhuma rotina de alocação ou liberação de memória, estas rotinas são implementadas pelo *Memory_Allocator*. Uma outra característica desta família é a modularidade, ou seja, uma nova estrutura de dados pode ser adicionada como um membro desta família facilmente, basta obedecer à interface da família. Abaixo as princi-

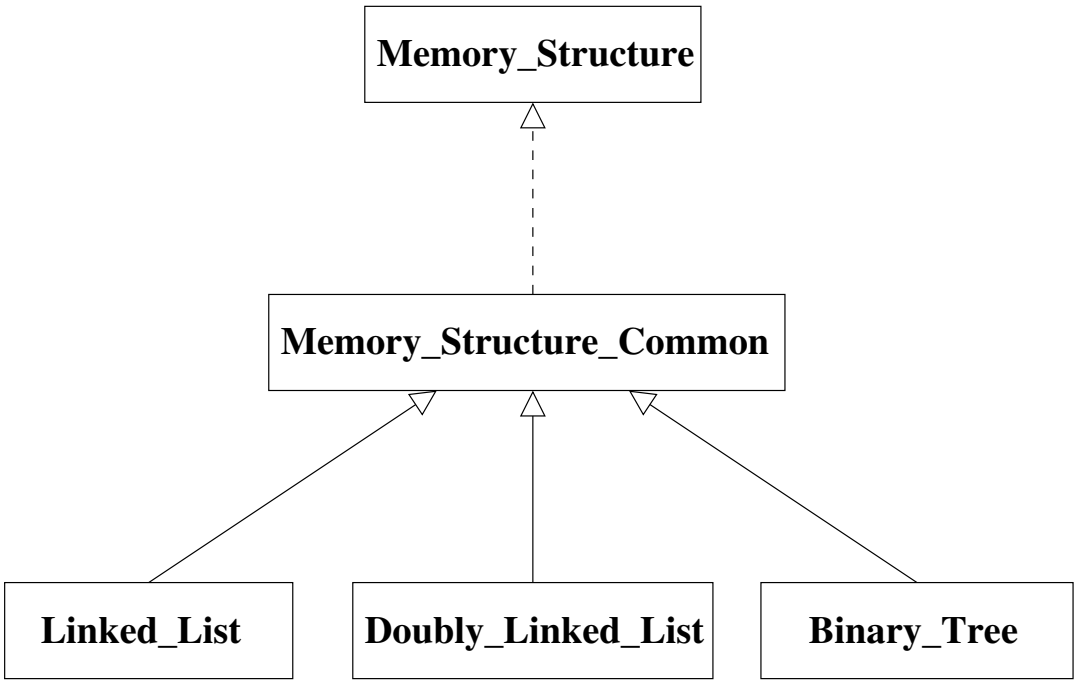

Figura 4.3: Diagrama de classes da famí lia*Memory_Structure*.

pais funcionalidades desta interface são descritas:

int insert(Node & node, Node & prev) Este método insere o elemento referenciado pelo parâmetro *nodo* na estrutura (seja ela qual for) logo após o nodo *prev*, e retorna a posição do nodo inserido.

void remove(Node & node) Este método remove o nodo referenciado por *node* da estrutura, deletando o mesmo.

4.3 *Memory_Allocator*

A famí lia*Memory_Allocator* é responsável por implementar a polí tica de alocação defi nida por*alloc*. Cada implementação utiliza a estrutura de dados (*Memory_Structure*) mais apropriada.

Para fi ns de teste e validação, foram implementados três alocadores, que são apresentados a seguir:

***Address-Ordered First-Fit* sem fusão de blocos contíguos:** esta política de alocação foi implementada utilizando o membro *Linked_List* da família *Memory_Structure* como uma lista de blocos livres de memória. A fim de evitar perda de desempenho na busca por blocos contíguos de memória e diminuir o uso de memória pela estrutura de dados, este alocador não realiza fusão destes blocos, o que pode causar problemas de fragmentação externa.

***Address-Ordered First-Fit* com fusão de blocos contíguos:** diferentemente do alocador anterior, este realiza fusão de blocos contíguos de memória durante a liberação dos mesmos. Para isso, foi utilizado um mecanismo *First-Fit* operando sobre o membro *Doubly_Linked_List* da família de estruturas de controle de memória. Uma lista duplamente encadeada diminui consideravelmente o processo de busca por blocos de memória livres e contíguos, com um aumento mínimo do tamanho da estrutura e de suas rotinas de controle.

***Binary Buddy - Buddy System* Binário:** este é o mecanismo mais comum de implementação de *Buddy Systems*. Ele foi implementado utilizando o membro *Binary_Tree* da família *Memory_Structure*, com algoritmos de busca em profundidade. *Buddy Systems* podem ocasionar fragmentação interna, o que pode se tornar um problema para algumas aplicações.

4.4 Interface Inflada

A interface inflada desta família provê apenas dois métodos e dois operadores, que são descritos a seguir:

void * malloc(int n): Este método aloca a quantidade de *bytes* apontada por *n* na *heap* do processo e retorna um ponteiro contendo o endereço da alocação.

void free(void * obj): Este método libera o bloco de memória apontado pelo ponteiro *obj*, inserindo este bloco na estrutura de blocos livres.

void * operator new(unsigned int n): Este operador chama o método *malloc()*, repassando *n* como parâmetro.

void operator delete(void * obj): Este operador chama o método *free()*, repassando *obj* como parâmetro.

Capítulo 5

Conclusão

Embora complexo, o domínio de gerência de memória foi amplamente absorvido, possibilitando a modelagem e implementação de um gerente de memória que satisfez as espectativas iniciais do projeto. O fato de todos os componentes do gerenciador manterem uma interface bastante completa facilita enormemente os trabalhos relacionados à portabilidade do sistema.

Este trabalho traz para o *EPOS* recursos que não são encontrados em sistemas operacionais projetados para atuarem no mesmo tipo de plataforma, que são *paginação* e a conciliação de um espaço de endereçamento *flat* com uma MMU (se existir), afim de provêr facilidades de proteção de memória.

Sob o ponto de vista da aplicação, o *EPOS malloc* deixa a critério do usuário a configuração do alocador dinâmico de memória, o que pode tornar as aplicações mais eficientes, já que a alocação de memória é o "calcanhar de aquiles" de qualquer aplicação quando se está levando em conta o desempenho.

5.1 Trabalhos Futuros

Como trabalhos futuros, várias extensões do projeto podem ser realizadas. Dentre elas destacam-se:

Suporte a SMP (*Symmetric Multiprocessors*): para garantir o funcionamento deste ge-

rente de memória em sistemas multiprocessados é necessário adicionar um controle de coerência de dados nas *caches*, tendo em vista que sistemas multiprocessados compartilham a memória principal, mas, geralmente, possuem sistemas próprios de memórias associativas. Isto poderia ser modelado como uma *confi gurable feature* do mediador da *MMU* em questão.

Suporte a DSM (*Distributed Shared Memory*): hoje em dia, cada vez mais o setor de super-computadores vem sendo tomado pelos *clusters*, e o uso de memória distribuí da traz grandes benefí cios ao processamento nestes ambientes.

Portes: Principalmente com o objetivo de conduzir experimentos no que diz respeito à portabilidade do sistema. Portes previstos para serem disponibilizados são o de uma MMU com suporte a paginação para o processador PowerPC 405GP e uma MMU para os controladores AVR A8 recentemente adquiridos pelo LISHA.

Referências Bibliográfi cas

[AND 92a] ANDERSON, T. The case for application-specifi c operating systems. In: PROCEEDINGS OF THE THIRD WORKSHOP ON WORKSTATION OPERATING SYSTEMS, 1992. **Proceedings...** Key Biscayne, U.S.A.: [s.n.], 1992. p.92–94.

[AND 92b] ANDERSON, T. The Case for Application-Specifi c Operating Systems. In: PROCEEDINGS OF THE THIRD WORKSHOP ON WORKSTATION OPERATING SYSTEMS, 1992. **Proceedings...** Key Biscayne, U.S.A.: [s.n.], 1992. p.92–94.

[BAR 99] BARR, M. **Programming Embedded Systems in C and C++**. O'Reilly, Janeiro, 1999.

[BAR 00] BAR, M. **Linux Internals**. Osborne McGraw-Hill, 2000.

[BEU 99] BEUCHE, D. et al. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In: PROCEEDINGS OF THE 2ND IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 1999. **Proceedings...** St Malo, France: [s.n.], 1999.

[BOL 97] BOLOSKY, W. J. et al. Operating System Directions for the Next Millennium. In: PROCEEDINGS OF THE SIXTH WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS, 1997. **Proceedings...** Cape Cod, U.S.A.: [s.n.], 1997. p.106–110.

[DAL 68] DALEY, R. C.; DENNIS, J. B. Virtual memory, processes, and sharing in multics. **Communications of the ACM**, [S.l.], v.11, n.5, p.306–313, 1968.

[DEN 65] DENNIS, J. B. Segmentation and the design of multiprogrammed computer systems. **Journal of the ACM**, [S.l.], v.12, n.4, p.589–602, oct, 1965.

[dMF 01] DE MEDEIROS FRöHLICH, A. A. **Application-Oriented Operating Systems**. Number17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik GmbH, 2001.

[dMF 02] DE MEDEIROS FRöHLICH, A. A. Epos - the reference manual. UFSC, 2002. Relatório técnico.

[HEN 98] HENNESSY, J. L.; PATTERSON, D. A. **Computer Organization and Design: The Hardware/Software Interface**. 2. ed. Morgan Kaufmann Publishers, 1998.

[HIT] HITACHI. **H8 300L Series Programming Manual**. Hitachi.

[HOW 61] HOWARTH, D. J.; KILBURN, T. The manchester university atlas oper-ating system, part ii: User s description. **Computer Journal**, [S.l.], v.4, n.2, p.226–229, oct, 1961.

[HTT 04a] HTTP://GCC.GNU.ORG/. **GNU Compiler Collection**. Site na Internet.

[HTT 04b] HTTP://WWW.GNU.ORG. **GNH Is Not Unix**. Site na Internet.

[HTT 04c] HTTP://WWW.LISHA.UFSC.BR. **Laboratório de Integração Software/Hardware**. Site na Internet.

[IBM 01] IBM Corporation. **PowerPC 405GP Embedded Processor User's Manual**, 2001.

[INT 97a] INTEL. **Intel Architecture Software Developer Manual**. Intel, 1997.

[INT 97b] INTEL. **Intel Architecture Software Developers Manual**. Intel, 1997.

[INT 97c] INTEL. **Intel Architecture Software Developers Manual**. Intel, 1997.

[INT 98a] INTEL. **Intel Architecture optimization Manual**. Intel, 1998.

[Int 98b] Intel Co. **Intel 440BX AGPset: 82443BX Host Bridge/Controller**, 1998.

[JN 02] JUAN NAVARRO, SITARAM IYER, P. D.; COX, A. Practical, transparent operating system support for superpages. In: PROCEEDINGS OF THE 5TH SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 2002. **Proceedings...** Boston, U.S.A: ACM Press, 2002. Operating Systems Review, p.89–104.

[KOL 92] KOLDINGER, E. J.; CHASE, J. S.; EGGERS, S. J. Architectural Support for Single Address Space Operating Systems. **Operating Systems Review**, [S.l.], v.26, p.175–186, Outubro, 1992.

[LAR 01] LARMAN, C. **Applying UML and Patterns**. second. ed. Unknown, 2001.

[ORG 72] ORGANICK, E. **The Multics System: an Examination of its Structure**. MIT Press, 1972.

[PIC 96] PICOLLI, L.; ÁVILA, R. B. Um gerente de memória baseado em paginação para o intel 486. CGCC da UFSC, 1996. Relatório técnico.

[RUB 97] RUBINI, A. **Linux Device Drivers**. Sebastopol, U.K.: O'Reilly, 1997.

[SIL 98] SILBERSCHATZ, A.; GALVIN, P.; PETERSON, J. **Operating Systems Concepts**. fi fth. ed. John Wiley and Sons, 1998.

[SMA 98] SMARAGDAKIS, Y.; BATORY, D. Implementing Reusable Object-Oriented Components. In: PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, 1998. **Proceedings...** Victoria, Canada: [s.n.], 1998.

[SP 94] SCHRöDER-PREIKSCHAT, W. **The Logical Design of Parallel Operating Systems**. Prentice-Hall, 1994.

[STR 97] STROUSTRUP, B. **The C++ Programming Language**. 3. ed. Addison-Wesley, Junho, 1997.

[TAN 92] TANENBAUM, A. S. **Modern Operating Systems**. Prentice-Hall, 1992.

[TEN 00] TENNENHOUSE, D. Proactive computing. **Communications of the ACM**, [S.l.], v.43, n.5, p.43–50, May, 2000.

[THO 01] THOMSEN, C. et al. Chaos — an rcx os where chaos is only in the name. Aalborg University, Maio, 2001. Relatório TécnicoE2-113-f1a.

[TK 61] TOM KILBURN, DAVID J. HOWARTH, R. P.; SUMNER, F. H. The manchester university atlas operating system, part i: Internal organization. **Computer Journal**, [S.l.], v.4, n.2, p.222–225, oct, 1961.

Apêndice A

Código-Fonte

```
// address_space.h
// Author: Arliones

#ifndef __address_space_h
#define __address_space_h

#include <system/config.h>
#include <mmu.h>
#include <segment.h>

__BEGIN_SYS
__BEGIN_INT

class Address_Space_Common
{
protected:
    Address_Space_Common() {}

public:
    typedef MMU::Log_Addr Log_Addr;
    typedef MMU::Phy_Addr Phy_Addr;
};

class Address_Space: public Address_Space_Common
{
public:
    Address_Space() __DEF;
    Address_Space(const Id & id) __DEF;
    Address_Space(const Address_Space & obj) __DEF;
    ~Address_Space() __DEF;
```

```
    const Id & id();
    bool valid();

    Log_Addr attach(Segment & seg, Log_Addr addr);
    int detach(Segment & seg);

    Phy_Addr physical(Log_Addr address);
};

class Flat_AS: public Address_Space
{
public:
    Flat_AS() __DEF;
    Flat_AS(const Id & id) __DEF;
    Flat_AS(const Flat_AS & obj) __DEF;
    ~Flat_AS() __DEF;

    const Id & id();
    bool valid();

    Log_Addr attach(Segment & seg, Log_Addr addr);
    int detach(Segment & seg);
};

class Paged_AS: public Address_Space
{
public:
    Paged_AS() __DEF;
    Paged_AS(const Id & id) __DEF;
```

```
    Paged_AS(const Paged_AS & obj) __DEF;
    ~Paged_AS() __DEF;

    const Id & id();
    bool valid();

    Log_Addr attach(Segment & seg, Log_Addr addr);
    int detach(Segment & seg);

    Phy_Addr physical(Log_Addr address);
};

class Segmented_AS: public Address_Space
{
public:
    Segmented_AS() __DEF;
    Segmented_AS(const Id & id) __DEF;
    Segmented_AS(Segmented_AS & obj) __DEF;
    ~Segmented_AS() __DEF;

    const Id & id();
    bool valid();

    Log_Addr attach(Segment & seg, Log_Addr addr);
    int detach(Segment & seg);

    Phy_Addr physical(Log_Addr address);
};

class Paged_Segmented_AS: public Paged_AS, public Segmented_AS
```

```
{
public:
    Paged_Segmented_AS() __DEF;
    Paged_Segmented_AS(const Id & id) __DEF;
    Paged_Segmented_AS(const Paged_Segmented_AS & obj) __DEF;
    ~Paged_Segmented_AS() __DEF;

    const Id & id();
    bool valid();

    Log_Addr attach(Segment & seg, Log_Addr addr);
    int detach(Segment & seg);

    Phy_Addr physical(Log_Addr address);
};

__END_INT
__END_SYS


#ifdef __FLAT_AS_H
#include __FLAT_AS_H
#endif

#ifdef __PAGED_AS_H
#include __PAGED_AS_H
#endif

#ifdef __SEGMENTED_AS_H
#include __SEGMENTED_AS_H
```

```
#endif

#ifdef __PAGED_SEGMENTED_AS_H
#include __PAGED_SEGMENTED_AS_H
#endif

#endif
```

```
// segment.h
// Author: Arliones

#ifndef __segment_h
#define __segment_h

#include <system/config.h>
#include <mmu.h>

__BEGIN_SYS
__BEGIN_INT

class Segment_Common
{
protected:
    Segment_Common() {}

public:
    typedef MMU::Log_Addr Log_Addr;
    typedef MMU::Phy_Addr Phy_Addr;
    typedef MMU::Flags Flags;
};

class Segment: public Segment_Common
{
public:
    Segment() __DEF;
    Segment(Size bytes, Flags flags = MMU::APP) __DEF;
    Segment(Phy_Addr addr, Size bytes, Flags flags = MMU::APP) __DEF;
    Segment(const Id & id) __DEF;
```

```
    Segment(const Segment & obj) __DEF;
    ~Segment() __DEF;

    const Id & id();
    bool valid();

    Size size() const;
    Phy_Addr phy_address() const;

    int resize(Size amount);
};

class Contiguous_Segment: public Segment_Common
{
public:
    Contiguous_Segment() __DEF;
    Contiguous_Segment(Size bytes, Flags flags = MMU::APP) __DEF;
    Contiguous_Segment(Phy_Addr addr, Size bytes, Flags flags = MMU::API
__DEF;
    Contiguous_Segment(const Id & id) __DEF;
    Contiguous_Segment(const Contiguous_Segment & obj) __DEF;
    ~Contiguous_Segment() __DEF;

    const Id & id();
    bool valid();

    Size size();
    Phy_Addr phy_address() const;
};
```

```
class Static_Segment: public Contiguous_Segment
{
public:
    Static_Segment() __DEF;
    Static_Segment(Size bytes, Flags flags = MMU::APP) __DEF;
    Static_Segment(Phy_Addr addr, Size bytes, Flags flags = MMU::APP) _
    Static_Segment(const Id & id) __DEF;
    Static_Segment(const Static_Segment & obj) __DEF;
    ~Static_Segment() __DEF;

    const Id & id();
    bool valid();

    Size size();
    Phy_Addr phy_address() const;
};

class Dynamic_Segment: public Static_Segment
{
public:
    Dynamic_Segment() __DEF;
    Dynamic_Segment(Size bytes, Flags flags = MMU::APP) __DEF;
    Dynamic_Segment(Phy_Addr addr, Size bytes, Flags flags = MMU::APP)
    Dynamic_Segment(const Id & id) __DEF;
    Dynamic_Segment(const Dynamic_Segment & obj) __DEF;
    ~Dynamic_Segment() __DEF;

    const Id & id();
    bool valid();
```

```
    Size size();
    Phy_Addr phy_address() const;

    void resize(Size amount);
};

__END_INT
__END_SYS


#ifdef __CONTIGUOUS_SEGMENT_H
#include __CONTIGUOUS_SEGMENT_H
#endif

#ifdef __STATIC_SEGMENT_H
#include __STATIC_SEGMENT_H
#endif

#ifdef __DYNAMIC_SEGMENT_H
#include __DYNAMIC_SEGMENT_H
#endif

#endif
```

```
// mmu.h
// Author: Arliones

#ifndef __mmu_h
#define __mmu_h

#include <system/config.h>

__BEGIN_SYS
__BEGIN_INT

class MMU_Common
{
protected:
    MMU_Common() {}

public:
    typedef CPU::Log_Addr Log_Addr;
    typedef CPU::Phy_Addr Phy_Addr;

    typedef unsigned int Flags;
    enum {
        RD = 0x01,
WR = 0x02,
EX = 0x04,
NC = 0x08,
CT = 0x10,
IO = 0x20,
USR = 0x40,
SUP = 0x80,
```

```
APP = (RD | WR | EX | USR),
SYS = (RD | WR | EX | SUP)
    };

    class Page_Table;
    class Page_Directory;
    class Segment_Table;
    class Chunk;
};

class MMU: public MMU_Common
{
public:
    MMU() __DEF;
    ~MMU() __DEF;

    void flush_tlb();
    void flush_tlb(Log_Addr addr);

    static Phy_Addr alloc(int bytes);
    static void free(Phy_Addr addr, int bytes);

    static Phy_Addr physical(Log_Addr addr);
};

class IA32_MMU: public MMU_Common
{
public:
    IA32_MMU() __DEF;
    ~IA32_MMU() __DEF;
```

```
    void flush_tlb();
    void flush_tlb(Log_Addr addr);

    static Phy_Addr alloc(int bytes);
    static void free(Phy_Addr addr, int bytes);

    static Phy_Addr physical(Log_Addr addr);
};

class PPC32_MMU: public MMU_Common
{
public:
    PPC32_MMU() __DEF;
    ~PPC32_MMU() __DEF;

    void flush_tlb();
    void flush_tlb(Log_Addr addr);

    static Phy_Addr alloc(int bytes);
    static void free(Phy_Addr addr, int bytes);

    static Phy_Addr physical(Log_Addr addr);
};

class H8_MMU: public MMU_Common
{
public:
    H8_MMU() __DEF;
    ~H8_MMU() __DEF;
```

```
    void flush_tlb();
    void flush_tlb(Log_Addr addr);

    static Phy_Addr alloc(int bytes);
    static void free(Phy_Addr addr, int bytes);

    static Phy_Addr physical(Log_Addr addr);

    static void switch_pd(Page_Directory * pd);
    static void switch_st(Segment_Table * st);
};

__END_INT
__END_SYS

#ifdef __IA32_MMU_H
#include __IA32_MMU_H
#endif

#ifdef __PPC32_MMU_H
#include __PPC32_MMU_H
#endif

#ifdef __H8_MMU_H
#include __H8_MMU_H
#endif

#endif
```

Apêndice B

Artigo

Famílias de Abstrações de Gerência de Memória para o EPOS

Arliones Stevert Hoeller Junior e Antônio Augusto de Medeiros Fröhlich
UFSC/CTC/LISHA
Caixa-Postal 476
88049-900 Florian´opolis - SC, Brasil
{arliones,guto}@lisha.ufsc.br
http://www.lisha.ufsc.br/∼{arliones,guto}

11 de fevereiro de 2004

Resumo

Ambientes de Programação Paralela e *Sistemas Embutidos* têm estado cada vez mais presentes no cotidiano das pessoas. Hoje, a grande maioria dos dispositivos eletrônicos existentes utilizam algum tipo de processador ou microcontrolador em sua implementação. Tendo como objetivo explorar os aspectos comuns destes sistemas, surgiu o *EPOS (Embedded Parallel Operating System)*. Este trabalho apresentar´a como foram projetadas e implementadas as fam´ılias de abstrações relacionadas à gerência de mem´oria do *EPOS*.

Palavras-chave: Gerência de Mem´oria, Sistemas Operacionais, Software B´asico, Sistemas Embutidos, Paginação, Engenharia de Software

Abstract

Embedded Systems and *Parallel Programming Environments* have been each time more present on people's daily life. Nowadays, most of the off-the-shelf electronic devices use some kind of processor or microcontroller in their implementation. The *EPOS (Embedded Parallel Operating System)* was conceived focusing to explore the commonalities on such systems. The goal of this work is to present how the families involved on the *EPOS's* Memory Management were designed and implemented.

Keywords: Memory Management, Basic Software, Operating Systems, Embedded Systems, Paging, Software Engeneering

1 Introdução

Em [TEN 00], Tennenhouse relata que, no ano de 2000, somente 2% dos processadores produzidos no mundo teriam como destino plataformas interativas (como computadores pessoais), ou seja, 98% destes processadores foram utilizados em sistemas dedicados, incluindo robôs, ve´ıculos e, principalmente, sistemas embutidos. Al´em disso, a grande maioria dos recursos destinados ao fomento de pesquisas nesta ´area têm como destino projetos que envolvem aqueles 2% de processadores.

Embora existam hoje muitos sistemas operacionais que se projetam no mercado como soluções para plataformas dedicadas, tais sistemas não provêm a mesma performance que pode ser oferecida por um sistema operacional desenvolvido especificamente para elas. Juntando a isso o fato de que, principalmente quando se trabalha com sistemas embutidos, h´a uma s´erie de limitações no *hardware* utilizado (para diminuir o custo ou o espaço ocupado), precisamos sempre utilizar um sistema operacional que não prejudique o desempenho do sistema executando operações desnecess´arias.

No entanto, como dito por [dMF 01] citando [AND 92a, SP 94], "*os adjetivos* gen´erico *e* ´otimo *não podem ser atribuídos ao mesmo sistema operacional*"[1]. Assim sendo, para consiguir em um determinado sistema dedicado um conjunto de *software* b´asico (sistema operacional, *device drivers*, e compiladores) que satisfaça requisitos de desempenho e que utilize todos os recursos dispon´ıveis no *hardware*, ´e necess´ario que se desenvolva um sistema operacional diferente para cada nova plataforma, o que ´e, sem d´uvida alguma, muito contraproducente.

Para tratar tais problemas, muitos pesquisadores da ´area de *software* b´asico têm dedicado seus esforços à criação e adaptação de t´ecnicas de engenharia de *software* que possam prover ao projeto de sistemas operacionais vantagens como reusabilidade, facilidade de manutenção e desenvolvimento mais eficiente. Vantagens estas que j´a têm ajudado projetistas e desenvolvedores de *software* aplicativo h´a muitos anos.

[1] Traduç˜ao pr´opria de: *the adjectives generic and optimal cannot be assigned to the same operating system*

Neste contexto, surgiu o *EPOS* (*Embedded Parallel Operating System*). Este sistema foi concebido com a finalidade de comprovar os conceitos apresentados por Fröhlich em [dMF 01]. Este trabalho apresentar´a como foram projetados e implementados os componentes de software envolvidos no gerenciamento de mem´oria do *EPOS*.

2 Conceitos

O termo *gerenciador de memória* pode ser empregado sob dois pontos de vista: o ponto de vista do sistema operacional e o da aplicação.

Sob o ponto de vista do sistema operacional, o gerenciador de mem´oria ´e um conjunto de funcionalidades que deve permitir às camadas de mais alto n´ıvel o acesso à mem´oria f´ısica e aos dispositivos de entrada e sa´ıda, provendo serviços de proteção e compartilhamento. J´a o ponto de vista da aplicação ´e diferente. Para a aplicação ´e necess´ario que os detalhes da implementação do sistema operacional e do *hardware* fiquem transparentes, ou seja, que seja definida uma interface para o gerenciador de mem´oria do sistema que atenda às suas necessidades e que, ao mesmo tempo, seja independente das camadas mais baixas do mesmo gerenciador.

A seguir serão demostrados alguns conceitos b´asicos de gerenciamento de mem´oria e, ap´os, serão descritos alguns esquemas de tradução de endereços. Todos estes conceitos apresentados a seguir foram extra´ıdos dos principais livros de sistemas operacionais [SIL 98, TAN 92] e em alguns artigos cient´ıficos que são citados no decorrer do texto.

2.1 Fragmentação

Fragmentação ´e o problema que ocorre quando h´a mem´oria não utilizada que não pode ser alocada por algum motivo. Existem dois tipos de fragmentação:

Fragmentação Externa: ocorre quando h´a mem´oria livre para atender a uma requisição, por´em, devido ao fato de esta mem´oria não estar contiguamente posicionada, esta requisição não pode ser satisfeita. Este ´e o pior tipo de fragmentação, pois ´e imposs´ıvel prever o tamanho da perda de mem´oria que pode ser ocasionada. A figura 1(a) demonstra a ocorrência de fragmentação externa.

Fragmentação Interna: ocorre quando h´a mem´oria alocada não utilizada. Este tipo de fragmentação est´a presente apenas em sistemas que fazem uso de partições de tamanho fixo para alocar mem´oria. Esta situação est´a representada na figura 1(b).

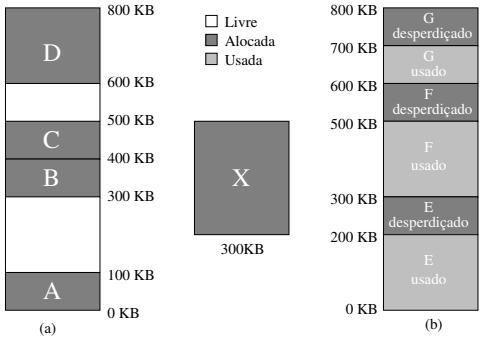

Figura 1: A fragmentação impede que o processo X seja alocado devido a fragmentação externa (a) e fragmentação interna (b).

2.2 Algoritmos de Alocação

O *algoritmo de alocação*, ou *mecanismo de alocação*, ´e respons´avel por implementar a pol´ıtica utilizada para escolher o bloco de mem´oria livre a ser alocado. A escolha de um algoritmo apropriado pode refletir de modo importante no desempenho e na quantidade de mem´oria perdida para fragmentação, principalmente em esquemas de gerência de mem´oria que não possuam suporte de *hardware*.

A literatura cl´assica de sistemas operacionais, outros artigos, relat´orios e documentações de implementações existentes apresentam in´umeros algoritmos. A seguir serão - apresentadas as principais classes de algoritmos de alocação, citando alguns representantes destas classes:

Bitmaped-Fit: esta classe utiliza *bitmaps* para controlar o uso da mem´oria. Cada *bit* no *bitmap* corresponde a um bloco de mem´oria.

Sequential-Fit: classe de algoritmos de alocação que utilizam uma lista encadeada (simples ou duplas) para armazenar os blocos de mem´oria livre. Exemplos de algoritmos desta classe são o *First-Fit* e o *Next-Fit*. A figura 2 apresenta a alocação de um bloco de mem´oria em uma lista de mem´oria j´a em uso atrav´es o algoritmo *First-Fit* (de (a) para (b)), e a liberação de outro bloco de mem´oria com a realização de fusão do mesmo bloco com um bloco cont´ıguo (de (b) para (c)).

Segregated-Free-List: aqui, a lista de blocos livres ´e dividida em v´arias outras listas, de acordo com o tamanho dos blocos. Estes algoritmos implementam pol´ıticas como o *Good-Fit* e o *Best-Fit*. Exemplos de algoritmos desta classe incluem o *Simple Segregated Storage*, *Segregated-Fit* e *Buddy-Systems*. A figura 3 apresenta uma s´erie de alocações e liberações de mem´oria utilizando um *Buddy System*.

Indexed-Fit: esta classe de algoritmos envolve aqueles onde a busca por blocos livres de mem´oria ´e realizada em uma estrutura de dados indexada (como ´arvores ou tabelas de *hash*). Para usar estes mecanismos, ´e necess´ario analisar a eficiência de alguns m´etodos. Manter ´arvores balanceadas e/ou ordenadas pode implicar no uso m´etodos custosos, o que precisa ser sempre considerado ao utilizar este tipo de mecanismo.

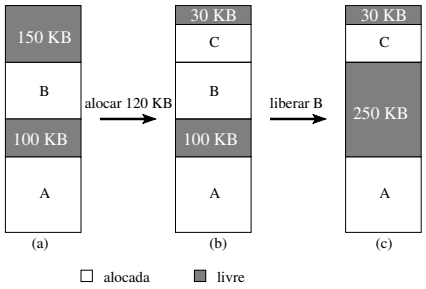

Figura 2: Processo de alocação utilizando listas encadeadas com o algoritmo First-Fit (a)(b) e de liberação de mem´oria com fusão de segmentos de mem´oria livres (b)(c).

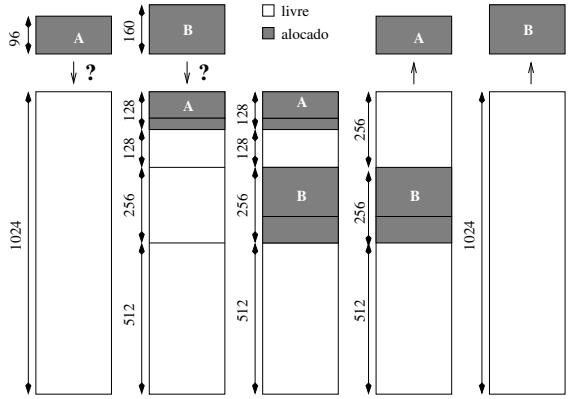

Figura 3: Processos de alocação e liberação de mem´oria no sistema Buddy System.

2.3 Tradução de Endereços

Esquemas de *tradução de endereços* são largamente empregados hoje em dia para facilitar a implementação de sistemas multi-tarefa. Quando estes esquemas são utilizados, o processo tem a ilusão de estar sozinho na *CPU*, tendo todo o espaço de endereçamento do processador para s´ı.

Boa parte das arquiteturas atualmente comercializadas oferecem algum tipo de suporte em *hardware* para tradução de endereços. Este *hardware* recebe o nome de MMU (*Memory Management Unit*). A MMU permite a separação entre espaço de endereçamento real (endereços f´ısicos) e do processador (endereços l´ogicos). A tradução de endereços ´e demonstrada na figura 4.

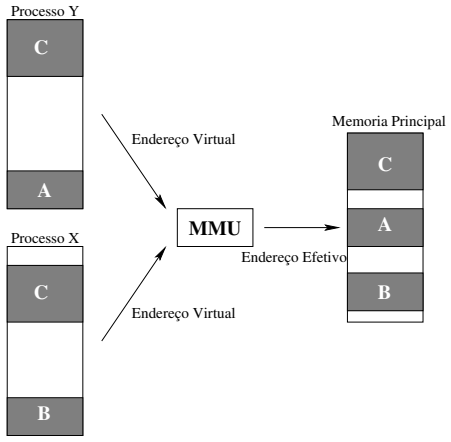

Figura 4: Tradução de endereços realizada pela MMU.

Existem ao menos três mecanismos utilizados para traduzir endereços: paginação, segmentação e segmentação paginada, este ´ultimo resultante da combinação dos dois primeiros.

A paginação[TK 61, HOW 61] consiste na divisão do espaço de endereçamento f´ısico em pequenos blocos de tamanho fixo chamados *frames*, e do espaço de endereçamento l´ogico em p´aginas (*pages*), que têm o mesmo tamanho dos *frames*. Quando ´e feita uma requisição de alocação, pode-se alocar dados na mem´oria f´ısica usando tantos frames quantos forem necess´arios. Tais frames não precisam ser cont´ıguos na mem´oria f´ısica, mas podem ser mapeados contiguamente no espaço de endereçamento l´ogico do processo. A visão da mem´oria que o processo tem ´e a visão l´ogica. Então, toda vez que uma referência à mem´oria ´e feita, esta referência ´e traduzida pela MMU para um endereço f´ısico e a posição de mem´oria desejada ´e acessada. A estrutura de dados utilizada para controlar o estado da mem´oria f´ısica chama-se tabela de p´aginas (*page table*). O mecanismo de tradução de endereços utilizado pela paginação ´e demonstrado na figura 5.

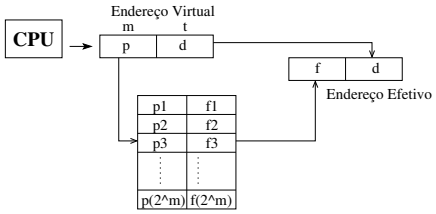

Figura 5: Paginação com um n´ıvel de tabela de p´aginas.

Tamb´em ´e poss´ıvel combinar v´arios n´ıveis de tabelas de p´aginas. A figura 6 apresenta o mecanismo de tradução

de endereços utilizando dois n´ıveis de tabelas de p´aginas. Aqui, cada entrada da tabela de p´aginas do primeiro n´ıvel aponta para uma tabela de p´aginas secund´aria que, finalmente, aponta para o *frame* solicitado.

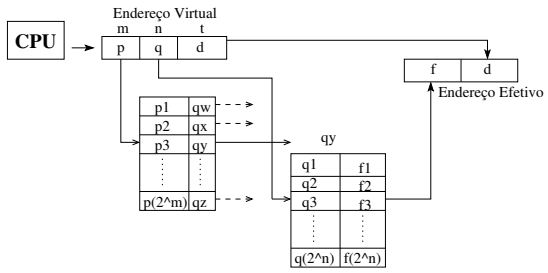

Figura 6: Paginação com dois n´ıveis de tabelas de p´aginas.

O conceito de Segmentação foi primeiramente apresentado em [DEN 65], e o de Segmentação com Paginação em [ORG 72]. Estes mecanismos, embora mais recentes, não são tão largamente utilizados, mesmo existindo em importantes arquiteturas (como os processadores da fam´ılia x86 da Intel). Neste caso, os processadores Intel podem ser configurados para que funcionem com paginação pura.

O esquema de segmentação divide a mem´oria f´ısica em segmentos cujos tamanhos são definidos pelo usu´ario em tempo de alocação. Para possibilitar a tradução de endereços, são armazenados (em registradores ou em mem´orias associativas) o endereço inicial e o final de cada segmento. O mecanismo de segmentação ´e demonstrado na figura 7, e o de Segmentação Paginada na figura 8

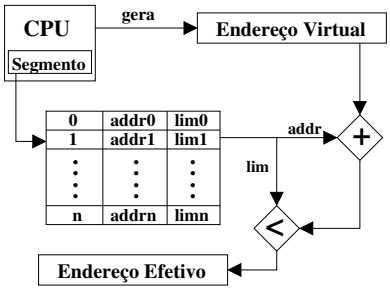

Figura 7: Esquema de segmentação de mem´oria.

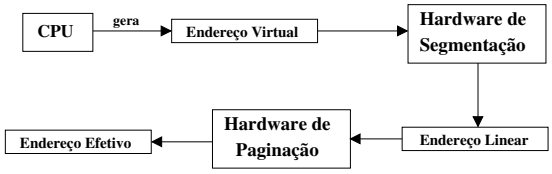

Figura 8: Esquema de segmentação paginada de mem´oria.

3 Sistemas Operacionais Orientados à Aplicação

Sistemas Operacionais Orientados à Aplicação (*Application-Oriented Operating Systems*) ´e um conceito introduzido por Fröhlich em [dMF 01]. Nesta obra ´e proposta uma nova metodologia para projeto de sistemas que procura, atrav´es da combinação de v´arias t´ecnicas de engenharia de *software*, atingir um alto grau de configurabilidade que permita a geração de sistemas espec´ıficos para determinadas aplicações, garantindo um elevado n´ıvel de re-usabilidade ao sistema, portabilidade às aplicações e baixo *overhead*.

Para isto, ´e apresentada ainda na mesma obra a metodologia de projeto multi-paradigma *Application-Oriented System Design*. Tal metodologia decompõe o dom´ınio em questão (Sistemas Operacionais, por exemplo) em *famílias de abstrações* independentes de cen´ario[2] que, atrav´es da reusabilidade, podem gerar v´arias instâncias do mesmo sistema. A configurabilidade pode ser atingida atrav´es do uso de *aspectos de cenário* e *configurable features*. Estas fam´ılias constituem *componentes de software*, e o conjunto destes componentes geram um *framework* que ´e o sistema em s´ı. Cada componente deste *framework* ´e acessado atrav´es de uma *Interface Inflada*, que garante a portabilidade dos artefatos de *software* que o utilizam.

4 *EPOS*

O *EPOS* (*Embedded Parallel Operating System*)[dMF 02] ´e o sistema operacional desenvolvido inicialmente por para validar os conceitos desenvolvidos na tese de Fröhlich . Este sistema foi concebido como solução para sistemas dedicados, incorporando aplicações na ´area de sistemas de controle e automação, robôs e sistemas embutidos em geral, al´em de sistemas para ambientes de computação paralela e distribu´ıda.

Sua primeira implementação visava utiliz´a-lo como sistema operacional dedicado para os n´os de um *cluster* de computadores pessoais baseados na arquitetura ix86 e interconectados por um sistema de comunicação de alto desempenho (*MYRINET*). Atualmente, os alunos do *LISHA* (Laborat´orio de Integração *Software/Hardware*) da *UFSC* (Universidade Federal de Santa Catarina), coordenados pelo Prof. Dr. Fröhlich, continuam o seu desenvolvimento e realizam suas pesquisas no âmbito deste sistema.

A composição do *EPOS* est´a baseada em três tipos de fam´ılias: *Abstrações*, *Mediadores* e *Aspectos*. As Abstrações são fam´ılias que englobam caracter´ısticas, funcionalidades e estruturas independentes de arquitetura (cen´ario).

[2]Entende-se por cen´ario um ambiente para o qual o sistema ser´a gerado, ou seja, o conjunto formado pela plataforma e pela aplicaç ˜ao a ser utilizada.

Elas são amplamente re-us´aveis e constituem a maior parte dos componentes do sistema que uma aplicação necessita. Os Mediadores são abstrações dependentes de arquitetura. Eles são respons´aveis por implementar as funcionalidades que as abstrações e/ou aplicações necessitam. As fam´ılias de Mediadores precisam obedecer um contrato de interface r´ıgido (Interface Inflada), de modo a garantir a portabilidade do sistema. As fam´ılias de Aspectos são utilizados pelo *EPOS* para oferecer configurabilidade ao sistema.

Al´em do uso de Aspectos, as fam´ılias podem utilizar *Configurable Features*[3] para permitir sua configuração. O uso de Aspectos e *Configurable Features* permite que ao *EPOS* atinjir um elevado grau de configurabilidade sem a inserção de muita "sujeira" no c´odigo-fonte.

Al´em dos componentes, o *EPOS* ´e ainda constitu´ıdo de um conjunto de ferramentas que permitem que o usu´ario (programador da aplicação) configure o sistema operacional conforme suas necessidades.

5 Famílias de Abstrações

Seguindo os conceitos da metodologia *Projeto de Sistemas Orientados à Aplicação*, o gerente de mem´oria do *EPOS* foi modelado como um conjunto de fam´ılias de abstrações. Isto foi feito ap´os uma apurada an´alise de dom´ınio, que focou-se em gerência de mem´oria para sistemas embutidos, seguida de uma decomposição de dom´ınio, gerando um conjunto de fam´ılias, aspectos e opções de configuração (*Configurable Features*) pertinentes ao gerente de mem´oria.

O dom´ınio foi abstra´ıdo em três fam´ılias, sendo duas *abstrações* independentes de arquitetura (*Address_Space* e *Segment*), e um *mediador* de *hardware* (*MMU*).

Este talvez seja o dom´ınio mais dependente de arquitetura existente dada a diversidade de mecanismos de gerência de mem´oria existentes. Portanto, foi uma tarefa exemplarmente dif´ıcil encontrar denominadores comuns entre os mais variados esquemas e implementações.

Contudo, embora não se tenha chegado a um meio de abstrair totalmente o os esquemas de gerência de mem´oria (*Flat*, *Paginação*, *Segmentação* e *Segmentação Paginada*), foi observado que ´e poss´ıvel definir um modo comum de requisição de mem´oria, independente das peculiaridades do sistema. Tal padronização nos procedimentos de requisição de mem´oria levou em conta o fato de que sempre que um processo requer uma quantidade de mem´oria, o mesmo não precisa especificamente solicitar, por exemplo, uma p´agina, basta solicitar um segmento de mem´oria de um determinado tamanho, ficando o procedimento efetivo de alocação a cargo das abstrações que "sabem" que tipo de esquema de gerência est´a sendo utilizado.

[3] A traduç ˜ao para *Confi gurable Feature* ´e *Caracter´ı stica Confi gur´avel* mas, como este termo foi introduzido em inglˆes na obra de Fr¨ohlich, foi decidido por n˜ao traduzi-lo no decorrer deste texto.

Esta id´eia inspirou-se, al´em das modelagens iniciais do *EPOS*, no trabalho de Piccoli e Ávila [PIC 96], que implementou o sistema de gerência de mem´oria do *ABOELHA*, um *nano-kernel* para sistemas distribu´ıdos, desenvolvido no LISHA na d´ecada de 1990.

Toda vez que um processo requer mem´oria, ele solicita a criação de um *segmento*, o que ´e realizado por um dos membros da fam´ılia *Segment*. Sempre que um segmento ´e instanciado, ele solicita à *MMU* que lhe seja alocada uma certa quantidade de mem´oria. A *MMU* procede a alocação de mem´oria levando em consideração as caracter´ısticas da arquitetura sendo utilizada. Para que o processo possa finalmente utilizar o segmento criado, o mesmo necessita anexar este segmento ao seu *espaço de endereçamento*, que ´e representado pela fam´ılia *Address_Space*. O membro desta fam´ılia tem conhecimento do esquema utilizado de gerência de mem´oria, e solicita à *MMU* que o segmento em questão seja anexado a s´ı.

O conceito de *memória virtual* pode ser totalmente ignorado aqui. Este gerente de mem´oria entende que um processo requisita mem´oria pura e simplesmente. Se esta mem´oria ´e gerenciada utilizando conceitos relacionados a mem´oria virtual, isto não precisa ser transparente para o usu´ario (programador de aplicação). Todos estes detalhes podem ser "escondidos" pela implementação espec´ıfica para cada arquitetura (*MMU*). O uso ou não de mem´oria virtual pelo sistema operacional pode constituir uma *configurable feature* desta *MMU* espec´ıfica caso a arquitetura ofereça suporte para tal.

A figura 9 apresenta um diagrama com as fam´ılias que constituem o componente Gerenciador de Mem´oria do sistema. A seguir estas três fam´ılias serão apresentadas, juntamente com os aspectos e *configurable features* definidas para elas.

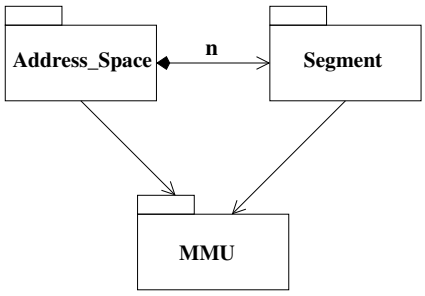

Figura 9: Diagrama de fam´ılias.

5.1 Família *Address_Space*

Os membros desta fam´ılia são respons´aveis por definir o esquema de mapeamento de endereços l´ogicos e f´ısicos do sistema operacional. Normalmente, esta definição leva em

consideração o tipo de *hardware* que uma determinada arquitetura apresenta. Contudo, neste gerente de mem´oria, ´e poss´ıvel em alguns casos desvincular estes conceitos. Naturalmente não faz sentido utilizar, por exemplo, o membro *Segmented_AS* desta fam´ılia em uma arquitetura que não disponibiliza *hardware* para segmentação de mem´oria. Contudo, ´e poss´ıvel utilizar o membro *Flat AS* em uma arquitetura cuja MMU suporta paginação e disponibilizar para o programador da aplicação a opção de usar este recurso. Neste caso, ´e poss´ıvel prover um modo de endereçamento *flat* com um mecanismo de proteção de mem´oria, bastando para isto garantir que os endereços l´ogico e f´ısico de cada *frame* coincidam.

Um diagrama contendo os membros desta fam´ılia pode ser visto na figura 10.

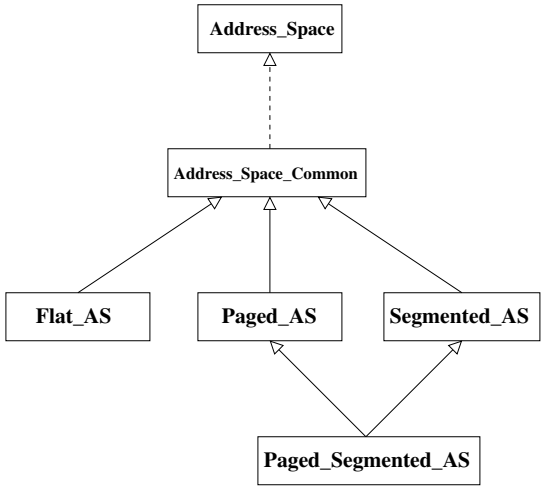

Figura 10: Diagrama de classes da fam´ılia *Address_Space*.

Da an´alise de dom´ınio foram extra´ıdos quatro esquemas de mapeamento de endereços relevantes, que foram agrupados na fam´ılia *Address_Space*. Estes membros são mutuamente exclusivos, ou seja, somente um pode ser utilizado por instância do sistema operacional. Tais membros são descritos nas pr´oximas seções.

5.1.1 *Flat_AS*

O membro *Flat_AS* desta fam´ılia ´e respons´avel por garantir a implementação de um esquema de gerência de mem´oria onde todo o espaço de endereçamento do sistema, incluindo mem´oria f´ısica e I/O, ´e entregue a um ´unico processo e não h´a tradução de endereços. A princ´ıpio, principalmente em arquiteturas mais simplificadas, este membro não possui funcionalidade alguma, ou seja, não ´e necess´ario que se implemente nada.

Contudo, no *EPOS* foi optado por fornecer um espaço de endereçamento *flat* com a possibilidade de efetuar proteção de regiões de mem´oria se a arquitetura utilizada suportar paginação. Deste modo, este membro implementa algumas funcionalidades que têm por objetivo garantir que todo mapeamento de mem´oria possui endereços f´ısico e l´ogico iguais. Vale ainda observar que, atrav´es do uso dos recursos de meta-programação est´atica da linguabem de programação *C++*, este membro não agrega c´odigo algum ao sistema quando compilado para uma arquitetura simples.

Se existir suporte a paginação, o *Address_Space* implementa o primeiro n´ıvel de tabelas de p´aginas, e a fam´ılia *Segment* ´e respons´avel pelo segundo n´ıvel desta estrutura. O funcionamento do sistema de paginação quando utilizado no modo *flat* ´e idêntico ao funcionamento deste mecanismo no modo paginado que ser´a descrito a seguir, com a ´unica resalva de que, aqui, os mapeamentos precisam manter endereços l´ogico e f´ısico idênticos.

5.1.2 *Paged_AS*

Este membro da fam´ılia *Address_Space* implementa um espaço de endereçamento paginado. Neste caso, o *Paged_AS* ´e respons´avel por implementar o primeiro n´ıvel de tabelas de p´aginas, ou seja, o *Page Directory* (Diret´orio de P´aginas). O segundo n´ıvel da estrutura, as *Page Tables* (Tabelas de P´aginas), são implementadas pela fam´ılia *Segment*.

Este modelo permite que, logicamente, o *EPOS* trate o mecanismo de paginação como um sistema de dois n´ıveis de tabelas de p´aginas, mas nada impede que, fisicamente, o sistema opere com MMU's que utilizem um n´umero diferente de n´ıveis (um ou mais que dois). Estas peculiaridades de cada arquitetura são resolvidas no mediador *MMU* que, como ser´a descrito adiante, ´e respons´avel por abstrair as caracter´ısticas de cada processador.

5.1.3 *Segmented_AS* e *Paged_Segmented_AS*

Embora poucas arquiteturas disponibilizem *hardware* para segmentação hoje em dia, os esquemas de *segmentação* e *segmentação paginada* foram modelados no *EPOS*. Isto se deve ao fato destes modelos serem utilizados pela arquitetura ix86, que ´e bastante difundida.

Contudo, mesmo sendo para esta arquitetura um dos portes deste gerente de mem´oria, estes membros da fam´ılia *Address_Space* não foram implementados, j´a que existe a possibilidade de emular paginação pura no processador em questão. Todavia, existe a possibilidade de que, caso interesse a alguma aplicação espec´ıfica, se faça a implementação de tais fam´ılias de modo pr´atico e sem comprometer a portabilidade do sistema.

5.1.4 Interface Inflada

Log_Addr attach(Segment & seg, Log_Addr addr): este m´etodo anexa o segmento *seg* no endereço indicado por

addr e retorna o endereço l´ogico onde o segmento foi mapeado ou outro valor em caso de erro. Este m´etodo falha caso o membro da fam´ılia que est´a sendo utilizado seja o *Flat_AS* e o parâmetro *addr* não for igual ao endereço f´ısico do segmento *seg*.

void detach(Segment & seg): Este m´etodo desanexa o segmento *seg* do *Address_Space*.

Phy_Addr physical(Log_Addr address): Este m´etodo - retorna o endereço f´ısico referente ao endereço l´ogico *address*.

5.2 Família *Segment*

Esta fam´ılia implementa o conceito l´ogico de segmento de mem´oria. Cada segmento representa para o usu´ario uma porção de de mem´oria f´ısica ou uma região de endereçamento onde existe um dispositivo de entrada e sa´ıda (I/O). Cada segmento est´a organizado de acordo com as caracter´ısticas de cada um dos membros. Estas peculiaridades de cada tipo de segmento serão descritos nas seções subsequentes.

Cada segmento agrega uma estrutura implementada pelo mediador *MMU* que se chama *Chunk*. O *Chunk* ´e respons´avel por abstrair as peculiaridades de cada arquitetura e por manter, juntamente com a *MMU*, uma interface constante, de modo a não tornar necess´ario que as *abstrações* do sistema conheçam caracter´ısticas espec´ıficas de cada implementação. A estrutura do *Chunk* ser´a melhor explicada na seção que trata do mediador da *MMU*. Os aspectos identificados como importantes para esta fam´ılia são apresentados juntamente dos membros da mesma na figura 11.

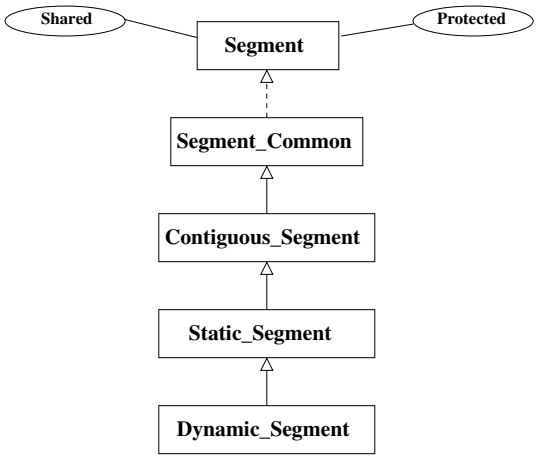

Figura 11: Diagrama de classes da fam´ılia *Segment*.

Uma importante função dos segmentos ´e o compartilhamento de mem´oria entre processos. Isto ´e implementado atrav´es do aspecto *Shared*, que garante a integridade no acesso a estes segmentos, tratando-os como regiões cr´ıticas, evitando erros devido à concorrência no acesso. Assim sendo, qualquer processo que conheça o segmento pode compartilh´a-lo. O aspecto *Protected* ´e respons´avel por implementar as caracter´ısticas de proteção de cada segmento. Ele controla as permissões de escrita, leitura e execução de cada instância de membros desta fam´ılia.

De acordo com os estudos realizados, foram identificados três poss´ıveis membros para esta fam´ılia. J´a que não h´a conflitos entre esses membros, eles não são mutuamente exclusivos, ou seja, podem ser utilizados conjuntamente em qualquer instância do sistema operacional. A ´unica restrição existente para alguns membros diz respeito à nessecidade de suporte a tradução de endereços, como ser´a descrito abaixo:

5.2.1 *Contiguous_Segment*

Este ´e o membro mais simples da fam´ılia *Segment*, e tamb´em o mais importante. Sua importância se d´a ao fato de não agregar tanto c´odigo ao sistema (devido à sua simplicidade) e, principalmente, por ser o ´unico que não requer a presença de *hardware* de tradução de endereços para operar, sendo este o caso mais comum em sistemas embutidos.

Um *Contiguous_Segment* ou, em português, segmento cont´ıguo, representa um bloco de mem´oria f´ısica onde todas as unidades de armazenamento (*bytes*, palavras ou p´aginas) são adjascentes. Outra caracter´ıstica deste tipo de segmento ´e que, embora possa ter seus *flags* de proteção alterados a qualquer momento, ele não pode ser re-dimensionado, j´a que tal funcionalidade acarretaria um *overhead* de processamento inaceit´avel, al´em do fato de que apenas em raros casos seria poss´ıvel encontrar blocos cont´ıguos de mem´oria para satisfazer as requisições de re-alocação sem que isto implicasse na realização de c´opia de mem´oria.

Mesmo em sistemas onde h´a suporte para tradução de endereços, o uso deste membro pode ser mais indicado j´a que, na maioria destes sistemas, a localidade espacial dos dados na mem´oria pode impactar de modo significativo na performance [HEN 98].

5.2.2 *Static_Segment*

Para o uso deste membro ´e necess´ario a existência de um esquema de tradução de endereços. Este segmento ´e est´atico, ou seja, não pode ser modificado ap´os criado, nem em seus *flags* de proteção, nem em seu tamanho ou mapeamento. O fato de ele levar em conta a existência de uma MMU implica em maior quantidade de c´odigo, o que pode vir a ser inconveniente em alguns casos. Um bom exemplo de utilidade para este tipo de segmento ´e para segmentos de c´odigo, que não são modificados em tempo de execução, são apenas criados e destru´ıdos.

5.2.3 *Dynamic_Segment*

O *Dynamic_Segment* ´e o membro mais completo da fam´ılia. Ele, assim como o *Static_Segment*, requer a presença de uma MMU. Contudo, diferentemente de seu "irmão", permite que sua configuração seja alterada dinamicamente (em tempo de execução). Este membro ´e a melhor opção para segmentos onde ´e realizado alocação dinâmica de mem´oria ou para segmentos de pilha (*Stack*), j´a que eles tendem a variar de tamanho de forma bastante aleat´oria, e a opção de re-dimensionamento pode oferecer uma gerência mais racional dos recursos do sistema.

5.2.4 Interface Inflada

O projeto desta fam´ılia seguiu um mecanismo incremental, ou seja, cada membro da fam´ılia implementa as funcionalidades de outro membro e mais algumas, de modo que todo membro ´e subclasse de outro. A interface deste tipo de fam´ılia ´e chamada *Incremental*, e ´e composta dos m´etodos do membro de mais baixo n´ıvel na hierarquia de classes da fam´ılia. Abaixo são descritos os m´etodos da interface da fam´ılia *Segment*.

Segment(Size bytes, Flags flags): Este construtor realiza a interação com o mediador *MMU* (que ser´a descrito adiante) para alocar um bloco de mem´oria com o tamanho indicado pelo parâmetro *bytes*, com as configurações indicadas por *flags*. Este deve ser o construtor mais utilizado para alocar mem´oria f´ısica, j´a que, neste tipo de alocação, geralmente não ´e importante o endereço f´ısico do segmento. Quando se deseja alocar mem´oria f´ısica em um endereço espec´ıfico ou mapear dispositivos de I/O, deve-se utilizar o construtor descrito no ´ıtem seguinte.

Segment(Phy_Addr addr, Size bytes, Flags flags): Este construtor procura alocar um bloco de endereços com o tamanho indicado pelo parâmetro *bytes* que inicie no ponto indicado por *addr*. A utilização deste construtor ´e mais indicada para mapear dispositivos de entrada e sa´ıda.

Size size(): Este m´etodo retorna o tamanho do segmento em *bytes*.

MMU::Chunk * chunk(): Este m´etodo retorna um ponteiro para o *Chunk* que representa, fisicamente, o segmento de mem´oria.

int resize(int amount): Este m´etodo altera a dimensão do segmento aumentando ou diminuindo seu tamanho se o valor do parâmetro *amount* for positivo ou negativo, respectivamente. Ele retorna *1* em caso de sucesso, *0* caso o tipo de segmento não suporte re-dimensionamento ou um valor negativo se ocorrer algum erro.

5.3 Família *MMU*

Esta fam´ılia de mediadores tem por objetivo abstrair as peculiaridades de cada arquitetura. Esta fam´ılia, no que diz respeito à interface de seus membros, possui um comportamento dissociado, j´a que cada membro abstrai as caracter´ısticas inerentes de cada *hardware*. No entanto, ´e extremamente importante que todos os membros desta fam´ılia obedeçam rigidamente a um contrato de interface, afim de garantir a portabilidade das *abstrações* e aplicações que fazem uso deste *mediador*. A estrutura desta fam´ılia ´e exibida na figura 12.

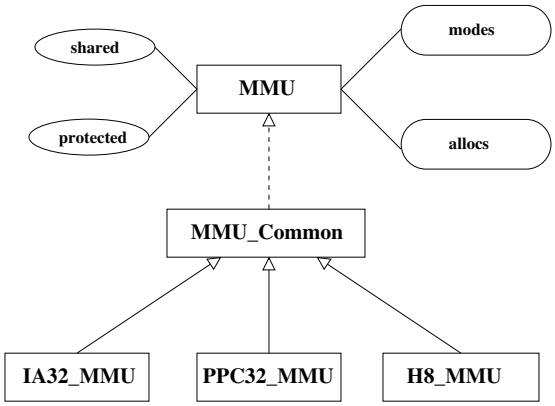

Figura 12: Diagrama de classes da fam´ılia *MMU*.

Uma caracter´ıstica importante desta fam´ılia ´e o fato de ela ser totalmente modul´avel, ou seja, para que outro membro (arquitetura) seja acoplada a esta fam´ılia, basta que este membro herde a classe *MMU_Common* e obedeça à interface definida. Nas seções seugintes serão discutidos os três membros implementados deste mediador.

5.3.1 *IA32_MMU*

Esta implementação d´a suporte ao uso de paginação. Embora a arquitetura *IA32* não possibilite o desligamento do *hardware* de segmentação, ´e poss´ıvel configura-lo de modo a emular um sistema de paginação pura. O modo como isto foi feito e como o mecanismo de paginação foi abstra´ıdo pode ser melhor entendido na seção que trata da Interface Inflada desta fam´ılia.

5.3.2 *PPC32_MMU*

Para esta arquitetura foram idealizadas duas implementações, sendo uma com e outra sem suporte a paginação. A escolha sobre qual das implementações usar ´e feita atrav´es de uma *Configurable Feature*.

5.3.3 *H8_MMU*

Este ´e o membro mais simples da fam´ılia *MMU*, j´a que ´e a ´unica das arquiteturas estudadas que não possui *hardware* de gerência de mem´oria.

5.3.4 Interface Inflada

Para garantir a portabilidade, cada membro da fam´ılia deve implementar, al´em das funcionalidades da interface, uma s´erie de estruturas de dados. Abaixo estão relacionadas algumas destas estruturas:

***Chunk*:** Esta estrutura ´e respons´avel por implementar o canal de comunicação entre as fam´ılias *Segment* e *MMU*. Cada segmento de mem´oria tem seu equivalente f´ısico representado por um *Chunk*. Esta classe ´e tamb´em respons´avel por abstrair o modo real de alocação de mem´oria. Ele mant´em uma interface r´ıgida e ´e utilizado apenas pelas *abstrações* de gerência de mem´oria.

***Page_Table*:** Esta estrutura implementa o mecanismo de paginação. Sendo ela tamb´em respons´avel pelas rotinas de mapeamento de mem´oria. Esta classe ´e utilizada pelo *Chunk* mesmo quando não h´a paginação, sendo então aqui implementado um poss´ıvel mecanismo de pseudo-paginação.

***Page_Directory*:** Este membro implementa o primeiro n´ıvel l´ogico de tabelas de p´aginas. Quando não h´a uso de paginação, sua implementação pode ser vazia. Esta classe ´e utilizada pelos membros *Flat_AS* e *Paged_AS* da fam´ılia *Address_Space*.

***Segment_Table*:** Esta estrutura não chegou a ser implementada durante este trabalho, j´a que não foi utilizada nenhuma arquitetura onde fosse necess´ario o uso do *hardware* de segmentação. Contudo, todas as implementações desta fam´ılia possuem implementações vazias desta estrutura, j´a que ela ´e utilizada pelos membros *Segmented_AS* e *Paged_Segmented_AS* da fam´ılia *Address_Space*.

Al´em destas estruturas, a interface inflada ainda oferece alguns m´etodos para serem utilizados tanto pelas abstrações do sistema quanto por programas do usu´ario, embora seja recomend´avel o uso das abstrações de mais alto n´ıvel por este ´ultimo. Abaixo são descritas estas funções:

Phy_Addr alloc(int n): Este m´etodo aloca *n* p´aginas cont´ıguas de mem´oria e retorna o endereço f´ısico da primeira destas p´aginas. Em arquiteturas sem suporte a paginação, uma p´agina ´e considerada como sendo um *byte*, sendo este o principal conceito do mecanismo de pseudo-paginação.

void free(Phy_Addr addr, int n): Este m´etodo re-coloca na lista de frames livres as *n* primeiras p´aginas (ou *bytes*, no caso de pseudo-paginação) a partir do endereço *addr*.

Phy_Addr physical(Log_Addr addr): Este m´etodo retorna o endereço f´ısico equivalente ao endereço l´ogico *addr*.

void flush_tlb(): Este m´etodo invalida todas as entradas do *buffer* de tradução de endereços (*TLB - Translation Lookaside Buffer*).

void flush_tlb(Log_Addr addr): Este m´etodo identifica o endereço da p´agina que cont´em o endereço *addr* e invalida a entrada desta p´agina na TLB.

6 Conclusão

Embora complexo, o dom´ınio de gerência de mem´oria foi amplamente absorvido, possibilitando a modelagem e implementação de um gerente de mem´oria que satisfez as espectativas iniciais do projeto. O fato de todos os componentes do gerenciador manterem uma interface bastante completa facilita enormemente os trabalhos relacionados à portabilidade do sistema.

Este trabalho traz para o *EPOS* recursos que não são encontrados em sistemas operacionais projetados para atuarem no mesmo tipo de plataforma, que são *paginação* e a conciliação de um espaço de endereçamento *flat* com uma MMU (se existir), afim de provêr facilidades de proteção de mem´oria.

Contudo, o maior trunfo deste trabalho ´e tornar o *EPOS* port´avel tanto a arquiteturas de 8 bits quanto para arquiteturas complexas de 32 ou 64 bits, sem subutilizar os recursos disponibilizados em processadores mais avançados, o que ´e imposs´ıvel em outros sistemas operacionais.

6.1 Trabalhos Futuros

Como trabalhos futuros, v´arias extensões do projeto podem ser realizadas. Dentre elas destacam-se:

Portes: Principalmente com o objetivo de conduzir experimentos no que diz respeito à portabilidade do sistema. Portes previstos para serem disponibilizados são o de uma MMU com suporte a paginação para o PowerPC 405GP e uma MMU para os controladores AVR A8 recentemente adquiridos pelo LISHA.

Suporte a SMP: para garantir o funcionamento deste gerente de mem´oria em sistemas multiprocessados ´e necess´ario adicionar um controle de coerência de dados nas

caches, tendo em vista que sistemas multiprocessados compartilham a mem´oria principal, mas, geralmente, possuem sistemas pr´oprios de mem´orias associativas. Isto poderia ser modelado como uma *configurable feature* do mediador da *MMU* em questão.

Suporte a DSM: hoje em dia, cada vez mais o setor de super-computadores vem sendo tomado pelos *clusters*, e o uso de mem´oria distribu´ıda traz grandes benef´ıcios ao processamento nestes ambientes.

SMP + MYRINET + MPI: ´e previsto ainda um porte completo do *EPOS* para o *SNOW Cluster* do LISHA.

Referências

[AND 92a] ANDERSON, T. The case for application-specifi c operating systems. In: PROCEEDINGS OF THE THIRD WORKSHOP ON WORKSTATION OPERATING SYSTEMS, 1992. **Proceedings...** Key Biscayne, U.S.A.: [s.n.], 1992. p.92–94.

[AND 92b] ANDERSON, T. The Case for Application-Specifi c Operating Systems. In: PROCEEDINGS OF THE THIRD WORKSHOP ON WORKSTATION OPERATING SYSTEMS, 1992. **Proceedings...** Key Biscayne, U.S.A.: [s.n.], 1992. p.92–94.

[BAR 99] BARR, M. **Programming Embedded Systems in C and C++**. O'Reilly, Janeiro, 1999.

[BAR 00] BAR, M. **Linux Internals**. Osborne McGraw-Hill, 2000.

[BEU 99] BEUCHE, D. et al. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In: PROCEEDINGS OF THE 2ND IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 1999. **Proceedings...** St Malo, France: [s.n.], 1999.

[BOL 97] BOLOSKY, W. J. et al. Operating System Directions for the Next Millennium. In: PROCEEDINGS OF THE SIXTH WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS, 1997. **Proceedings...** Cape Cod, U.S.A.: [s.n.], 1997. p.106–110.

[DAL 68] DALEY, R. C.; DENNIS, J. B. Virtual memory, processes, and sharing in multics. **Communications of the ACM**, [S.l.], v.11, n.5, p.306–313, 1968.

[DEN 65] DENNIS, J. B. Segmentation and the design of multiprogrammed computer systems. **Journal of the ACM**, [S.l.], v.12, n.4, p.589–602, oct, 1965.

[dMF 01] DE MEDEIROS FR¨oHLICH, A. A. **Application-Oriented Operating Systems**. Number17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik GmbH, 2001.

[dMF 02] DE MEDEIROS FR¨oHLICH, A. A. Epos - the reference manual. UFSC, 2002. Relat´orio t´ecnico.

[HEN 98] HENNESSY, J. L.; PATTERSON, D. A. **Computer Organization and Design: The Hardware/Software Interface**. 2. ed. Morgan Kaufmann Publishers, 1998.

[HOW 61] HOWARTH, D. J.; KILBURN, T. The manchester university atlas oper-ating system, part ii: User s description. **Computer Journal**, [S.l.], v.4, n.2, p.226–229, oct, 1961.

[IBM 01] IBM Corporation. **PowerPC 405GP Embedded Processor User's Manual**, 2001.

[INT 97a] INTEL. **Intel Architecture Software Developer Manual**. Intel, 1997.

[INT 97b] INTEL. **Intel Architecture Software Developers Manual**. Intel, 1997.

[INT 97c] INTEL. **Intel Architecture Software Developers Manual**. Intel, 1997.

[INT 98a] INTEL. **Intel Architecture optimization Manual**. Intel, 1998.

[Int 98b] Intel Co. **Intel 440BX AGPset: 82443BX Host Bridge/Controller**, 1998.

[JN 02] JUAN NAVARRO, SITARAM IYER, P. D.; COX, A. Practical, transparent operating system support for superpages. In: PROCEEDINGS OF THE 5TH SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 2002. **Proceedings...** Boston, U.S.A: ACM Press, 2002. Operating Systems Review, p.89–104.

[KOL 92] KOLDINGER, E. J.; CHASE, J. S.; EGGERS, S. J. Architectural Support for Single Address Space Operating Systems. **Operating Systems Review**, [S.l.], v.26, p.175–186, Outubro, 1992.

[LAR 01] LARMAN, C. **Applying UML and Patterns**. second. ed. Unknown, 2001.

[ORG 72] ORGANICK, E. **The Multics System: an Examination of its Structure**. MIT Press, 1972.

[PIC 96] PICOLLI, L.; ÁVILA, R. B. Um gerente de mem´oria baseado em paginaç ˜ao para o intel 486. CGCC da UFSC, 1996. Relat´orio t´ecnico.

[RUB 97] RUBINI, A. **Linux Device Drivers**. Sebastopol, U.K.: O'Reilly, 1997.

[SIL 98] SILBERSCHATZ, A.; GALVIN, P.; PETERSON, J. **Operating Systems Concepts**. fi fth. ed. John Wiley and Sons, 1998.

[SMA 98] SMARAGDAKIS, Y.; BATORY, D. Implementing Reusable Object-Oriented Components. In: PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, 1998. **Proceedings...** Victoria, Canada: [s.n.], 1998.

[SP 94] SCHR¨oDER-PREIKSCHAT, W. **The Logical Design of Parallel Operating Systems**. Prentice-Hall, 1994.

[TAN 92] TANENBAUM, A. S. **Modern Operating Systems**. Prentice-Hall, 1992.

[TEN 00] TENNENHOUSE, D. Proactive computing. **Communications of the ACM**, [S.l.], v.43, n.5, p.43–50, May, 2000.

[THO 01] THOMSEN, C. et al. Chaos — an rcx os where chaos is only in the name. Aalborg University, Maio, 2001. Relat´orio T´ecnicoE2-113-f1a.

[TK 61] TOM KILBURN, DAVID J. HOWARTH, R. P.; SUMNER, F. H. The manchester university atlas operating system, part i: Internal organization. **Computer Journal**, [S.l.], v.4, n.2, p.222–225, oct, 1961.