UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E DE ESTATÍSTICA
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Um Gerente de Memória Baseado em Paginação para o Intel 486

por

Luciano Piccoli
Rafael Bohrer Ávila

Relatório submetido como requisito para a obtenção do grau de
Bacharel em Ciência da Computação

Prof. Antônio Augusto Medeiros Fröhlich
Orientador

Florianópolis, março de 2002.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Piccoli, Luciano, Ávila, Rafael Bohrer

Um Gerente de Memória Baseado em Paginação para o Intel 486 / Luciano Piccoli e Rafael Bohrer Ávila. — Florianópolis: CGCC da UFSC, 2002.
46 p.: il.

Relatório de Projeto Final — Universidade Federal de Santa Catarina, Curso de Graduação em Ciência da Computação, Florianópolis, 2002. Orientador: Fröhlich, Antônio Augusto Medeiros.

Projeto: Sistemas operacionais, Arquitetura de Computadores.
Gerência de memória, i486, paginação.

AGRADECIMENTOS

Os acadêmicos gostariam de agradecer a seus pais, a seus colegas, ao orientador Antônio Augusto Medeiros Fröhlich e a todas as pessoas que tornaram viável a realização deste projeto.

Faz-se também um agradecimento especial à UFSC e a seus funcionários, os quais colaboraram de forma anônima durante todo o curso.

SUMÁRIO

LISTA DE FIGURAS **4**

RESUMO **5**

ABSTRACT **6**

1 INTRODUÇÃO **7**

2 ESTRATÉGIAS DE GERÊNCIA DE MEMÓRIA **8**

2.1 Segmentação 9

2.2 Paginação 10

2.3 Segmentação paginada 12

3 O PROCESSADOR I486 . . . . . **13**

3.1 Histórico 13

3.2 Características gerais 14

3.2.1 Modelo de memória segmentada 14

3.2.2 Registradores 15

3.2.2.1 Registradores de uso geral 15

3.2.2.2 Registradores de segmentos 16

3.2.2.3 Registradores de sistema 17

3.2.3 Portas de I/O 18

3.3 Modos de operação 18

3.3.1 Modo real 19

3.3.2 Modo virtual 19

3.3.3 Modo protegido 19

3.4 *Multitasking* 20

3.5 Gerência de memória em modo protegido 21

3.5.1 Segmentação 21
3.5.1.1 Modo flat 22
3.5.2 Paginação 23

4 INICIALIZAÇÃO DO SISTEMA 26
4.1 O processo de *boot* 26
4.2 Configuração do sistema - *setup* 27
4.3 Inicialização do *kernel* - *init* 28

5 O GERENTE DE MEMÓRIA 30
5.1 Estrutura do gerente de memória 30
5.1.1 Tabela de páginas do kernel 31
5.1.2 Mapeamento da memória física 33
5.1.3 Segmentos lógicos 34
5.2 Características 35
5.2.1 Proteção da memória 35
5.2.2 Compartilhamento 36
5.2.3 Tratamento de exceções 37
5.2.4 Autoexpansibilidade da pilha 38
5.3 Implementação do gerente de memória 39
5.3.1 Funcionalidade do gerente de memória 39
5.3.2 Exemplo de utilização dos segmentos por processos 42

6 PUBLICAÇÕES 43

7 CONCLUSÃO 44

REFERÊNCIAS BIBLIOGRÁFICAS 45

LISTA DE FIGURAS

Figura 2.1 Estratégia de segmentação. 9

Figura 2.2 Fragmentação externa. 10

Figura 2.3 Estratégia de paginação. 11

Figura 3.1 Localização de um dado usando segmento:*offset*. 15

Figura 3.2 Estrutura de um descritor de segmento. 22

Figura 3.3 Mecanismo de paginação no i486. 23

Figura 3.4 Estrutura do endereço linear. 24

Figura 3.5 Tradução de endereços através da paginação. 25

Figura 5.1 Estrutura da tabela de páginas do kernel. 33

Figura 5.2 Formato de uma entrada da tabela de segmentos. 35

Figura 5.3 Formato de uma entrada em uma tabela de páginas. 35

Figura 5.4 Ilustração do compartilhamento de segmentos entre processos. . 37

RESUMO

Este relatório descreve o desenvolvimento do projeto denominado "Um Gerente de Memória Baseado em Paginação para o Intel 486", realizado durante o ano de 1996. O gerente implementado faz uso de recursos avançados do processador i486, de forma a garantir a integridade e a segurança do sistema. Estes objetivos são plenamente atingidos através do uso do mecanismo de paginação presente no i486. No intuito de facilitar a integração do gerente de memória com processos de nível usuário e as demais partes do núcleo do sistema operacional, define-se uma interface amigável e de simples utilização. Desta forma, o gerente de memória pode ser utilizado tanto na implementação de aplicações dedicadas, quanto como base para o desenvolvimento de um sistema operacional completo.

TITLE: "A MEMORY MANAGER FOR THE INTEL 486 BASED ON PAGING"

ABSTRACT

This report describes the development of "A Memory Manager for the Intel 486 Based on Paging", during the year of 1996. The implemented memory manager makes use of the Intel 486 processor advanced features to ensure integrity and security to the system. These goals are achieved by using the paging mechanism. The user interface defined for the memory manager is friendly an easy to use, allowing its integration to other parts of the kernel. The memory manager can be used to implement dedicated applications or as a base for the development of a new operating system.

1 INTRODUÇÃO

Desde que a IBM escolheu o 8088 como processador principal de sua linha de computadores pessoais, esta família de processadores vem sendo cada vez mais utilizada. Hoje, a maior parte dos computadores pessoais está equipada com estes tipos de processadores. Ao longo dos anos, esta família tem incorporado uma série de novas características, tais como gerência de memória, multitasking e mecanismos de proteção. Entretanto, apesar da potencialidade dos processadores, poucos sistemas operacionais fazem uso dos recursos mais avançados.

O projeto "Um Gerente de Memória Baseado em Paginação para o Intel 486" abrange as áreas de arquitetura de computadores e sistemas operacionais. A gerência de memória é parte fundamental de um sistema operacional, pois é responsável pela disponibilização da memória aos diversos processos em execução, pela garantia da integridade do sistema e por evitar interferências indesejadas entre processos.

O objetivo deste projeto é a implementação de um conjunto de funções e estruturas de dados que se propõe a suportar o gerenciamento da memória de forma robusta e flexível em máquinas baseadas no processador i486 (lê-se "Intel 486"). Para tanto, são utilizados os recursos avançados do processador, de forma que o gerente sirva como base confiável para o desenvolvimento de futuras aplicações.

Na primeira parte deste documento, são expostas as formas de gerenciamento de memória mais utilizadas, mostrando-se suas vantagens e desvantagens. Na seqüência, faz-se um estudo sobre as características importantes do processador i486, destacando os dispositivos de gerência de memória. Após, é exposta a forma implementada para inicializar corretamente o processador. A estrutura e a interface do gerente de memória, bem como um exemplo de sua utilização, são descritas no capítulo subseqüente. Por fim, os autores expõem suas conclusões a respeito do gerente implementado.

2 ESTRATÉGIAS DE GERÊNCIA DE MEMÓRIA

No sentido de otimizar ao máximo a utilização da memória, existem uma série de métodos de gerência com diferentes abordagens. Tais abordagens vão desde métodos simples, onde não existe nenhuma forma de controle, até estratégias mais complexas, proporcionando altos índices de utilização da memória. Além disso, a memória é desperdiçada com o uso de estruturas de controle deve ser levada em consideração.

A utilização de memória é essencial durante toda a vida do sistema operacional. Por exemplo, quando um processo é carregado, seu código executável deve residir em memória. O conjunto de dados de um processo deve estar na memória para que possa ser acessado. E ainda, um processo pode querer alocar mais memória depois de estar rodando. Assim, a gerência eficiente da memória é fundamental para que se possa garantir o desempenho do sistema como um todo.

Em um ambiente sem qualquer tipo de gerência, a utilização da memória é a mais natural possível: à medida em que se necessita de memória, procura-se a primeira área disponível de tamanho suficiente, alocando-a. Vários problemas surgem decorrentes desta abordagem, entre eles a fragmentação da memória e a relocação de código.

Quando um processo é executado, são feitas referências a endereços de memória, seja no *fetch*[1] de suas instruções ou na leitura de um dado. O código de um processo não pode ser mudado, e portanto os endereços referenciados serão sempre os mesmos. Isto representa um sério problema, pois um processo deveria, pelo exposto, ser sempre carregado no mesmo endereço. Mais ainda, este endereço deveria ser conhecido em tempo de compilação. Obviamente, isto não ocorre assim e, na prática, existem várias maneiras de resolver este problema. Uma das alternativas é a

[1] O *fetch* ocorre quando a próxima instrução a ser executada é buscada na memória

diferenciação de *endereços lógicos* e *endereços físicos*. Este método é mais detalhado no capítulo seguinte.

2.1 Segmentação

Na estratégia de segmentação, a memória é vista como uma coleção de segmentos [sil90], como mostra a figura 2.1. Um segmento é um bloco contínuo de memória que pode ter tamanho variável, de acordo com a necessidade.

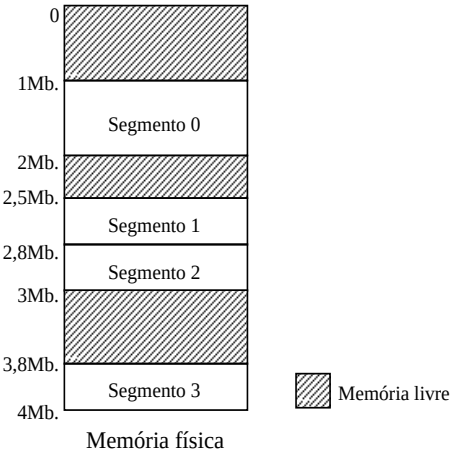


Figura 2.1: Estratégia de segmentação.

Um segmento pode ser definido a partir de qualquer posição de memória. Como os segmentos podem ter tamanho variável, formar-se-ão espaços vazios de tamanho indeterminado entre eles. Dependendo do tamanho, um novo segmento não poderá ser alocado neste espaço.

Uma situação peculiar ocorre quando o mecanismo de segmentação é utilizado. Esta situação, mostrada na figura 2.2, é chamada de *fragmentação externa*. Como se observa, não existe espaço contínuo de tamanho suficiente para conter o segmento S, mas a soma dos espaços existentes é até maior do que o tamanho de S. Estatisticamente, a segmentação desperdiça cerca de 1/3 da memória física total, implicando em uma perda considerável.

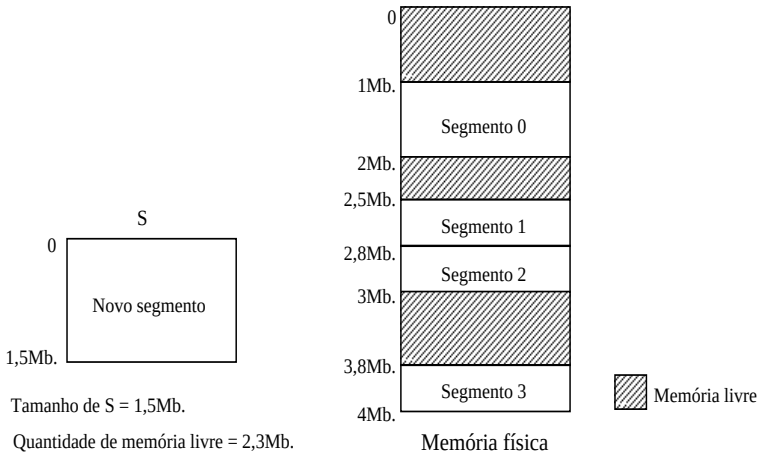


Figura 2.2: Fragmentação externa.

2.2 Paginação

A paginação divide o espaço de endereçamento em blocos de tamanho fixo, chamados páginas. Neste caso, a menor unidade de memória alocável é uma página. O tamanho da página varia de acordo com a arquitetura, podendo, em certos casos, ser definido pelo próprio sistema operacional. Tipicamente, o tamanho da página varia entre 2, 4 ou 8 Kbytes. Em sistemas que fazem uso de memória virtual, o tamanho da página está diretamente relacionado com o tamanho dos blocos de disco.

A vantagem da paginação sobre a segmentação é a inexistência da fragmentação externa. Como todas as páginas têm o mesmo tamanho, não haverá o problema de um bloco de memória (página) não caber num espaço vazio. Porém, com o uso da paginação existe a fragmentação *interna*, que consiste na perda do espaço não aproveitado dentro de uma página. Tipicamente, a quantidade de memória perdida com fragmentação interna é de 1/2 página *por processo*. Em comparação com o mecanismo de segmentação, a utilização da memória é mais eficiente.

Para controlar a tradução de endereços existe, para cada processo, uma tabela relacionando os blocos lógicos (endereços lógicos) com os blocos físicos alocados (endereços físicos). Esta situação pode ser visualizada na figura 2.3, onde existe

um espaço de endereçamento lógico mapeado para endereços físicos através de uma tabela de páginas.

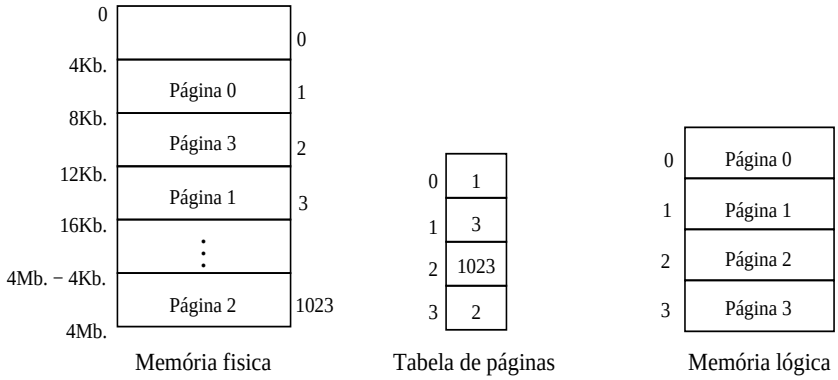


Figura 2.3: Estratégia de paginação.

No entanto, na existência de um vasto espaço de endereçamento lógico, a tabela de páginas associada a cada processo torna-se muito grande, ocupando um espaço de memória considerável. Este problema caracteriza a *superfluidade* [goo90]. Existem uma série de métodos para evitar este problema: aumento do tamanho da página, utilização de dois ou mais níveis de tabelas ou ainda pela utilização da paginação invertida[2].

Uma outra vantagem desta abordagem é a facilidade do compartilhamento de páginas de memória. Para que se efetive um compartilhamento entre processos, basta referenciar a mesma página física atraves das tabelas de páginas dos processos. Como uma página física não está diretamente vinculada a um endereço físico, diferentes processos podem acessar a mesma memória por meio de diferentes endereços lógicos.

[2]Neste mecanismo existe apenas uma tabela de conversão, onde o indexador é o endereço físico e cada entrada possui um endereço lógico e um processo associado. Devido a sua ineficiência, este método é muito pouco utilizado

2.3 Segmentação paginada

As duas estratégias podem ser combinadas, gerando a *segmentação paginada*. A segmentação paginada divide a memória em segmentos, como faz a segmentação. Porém, um segmento é alocado em páginas, não em bytes. Desta forma o problema da fragmentação externa é evitado, existindo apenas a fragmentação interna gerada pela paginação. A segmentação paginada apresenta um excelente aproveitamento de memória é também permite o compartilhamento de memória, a nível de segmentos.

Esta estratégia apresenta algumas desvantagens, tais como a necessidade de mecanismos mais complexos para a tradução de endereços físicos em lógicos, além do aumento do tempo necessário para realizar esta tarefa.

3 O PROCESSADOR I486

3.1 Histórico

O processador Intel 486 é um processador CISC de 32 bits bastante versátil. Atualmente usado nos micros da linha IBM-PC e compatíveis, é o processador que integra grande parte dos computadores pessoais.

O i486 é uma evolução do 8086, processador utilizado no projeto do primeiro PC. Desde então, a Intel passou a produzir novos processadores compatíveis com o 8086/8088. A justificativa para a compatibilidade deve-se ao grande número de aplicativos existentes na época para este processador. O primeiro sucessor do 8086/8088 foi o 80286. A grande novidade deste processador é a existência de dois modos de operação: o modo real e o modo protegido, sendo que no primeiro modo, o 80286 comporta-se exatamente como um 8088, porém com velocidade de execução maior. Além disso, são definidas algumas novas instruções para o modo real não presentes no 8088. Em modo protegido, o 80286 disponibiliza uma série de novos recursos, tornando possível o *multitasking* e a gerência de memória através da segmentação.

Na seqüência, a Intel lançou o 80386, um processador mais poderoso do que o 80286 e totalmente compatível com o 8088. Houve grandes avanços na construção deste processador. Além dos modos real e protegido, foi introduzido o modo virtual, no qual o processador se comporta como no modo real, porém com a possibilidade de utilização de *multitasking*. Outro grande avanço do 80386 é a implementação do mecanismo de paginação. A partir deste mecanismo, torna-se possível a implementação de uma gerência de memória mais eficiente.

Na geração seguinte de processadores surge o i486. Além de todas as funcionalidades do 80386, o i486 possui poucas características adicionais, tais como um coprocessador embutido e algumas instruções novas. O coprocessador permite

que as operações matemáticas envolvendo ponto flutuante possam ser rapidamente realizadas, enquanto as novas instruções facilitam o gerenciamento do sistema.

3.2 Características gerais

Por manter perfeita compatibilidade com seus antecessores, o processador i486 possui diversas características herdadas desde o 8086/8088. Essas características serão descritas nas seções seguintes.

3.2.1 Modelo de memória segmentada

Quando lançou o processador 8086, a Intel introduziu o modelo de memória segmentada que caracteriza todos os processadores da linha. Com a introdução de recursos avançados, nos processadores mais novos, o esquema de segmentação foi adquirindo características diferentes, mas o princípio manteve-se o mesmo.

Em um processador convencional, onde a memória é vista como um vetor contínuo de bytes, um endereço de memória é representado por um único valor, que aponta o dado desejado dentro do vetor. No modelo de memória segmentada, qualquer referência à memória implica na definição de dois fatores: um *segmento* e um *offset*. O segmento identifica uma porção limitada da memória, ou seja, uma faixa de valores dentro do vetor de bytes. O *offset*, então, identifica o dado desejado *dentro do segmento*. A figura 3.1 ilustra esse mecanismo. Note que é possível que um mesmo dado seja referenciado por várias combinações de segmento e *offset*.

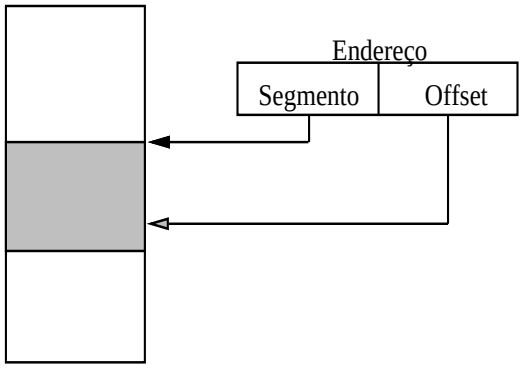


Figura 3.1: Localização de um dado usando segmento:*offset*.

3.2.2 Registradores

Para exercer todas as suas funções, o processador i486 conta com um conjunto relativamente grande de registradores. De acordo com essas funções, os registradores do i486 são separados em diversos grupos.

3.2.2.1 Registradores de uso geral

Os registradores de uso geral são aqueles usados na maioria das instruções do processador, sendo aplicados nas mais diferentes tarefas. A seguir são relacionados os registradores de uso geral, juntamente com suas principais funções.

`EAX :` Usado na maioria das instruções aritméticas e, tradicionalmente, no retorno de chamadas de funções;

`EBX :` Este registrador é usado como base em instruções de endereçamento indexado;

`ECX :` Usado como contador na execução de *loops*;

`EDX :` Usado em algumas funções aritméticas, em conjunto com `EAX`;

`ESI` **e** `EDI` **:** Estes dois registradores são utilizados em operações de *strings*, e também em instruções de endereçamento indexado;

`BP` **:** Usado principalmente como base para a montagem de *stack frames* na entrada de sub-rotinas;

`ESP` **:** *Stack pointer*, usado nas operações de pilha.

Todos os registradores de uso geral são de 32 bits. A fim de manter compatibilidade com o 8086/8088, todos podem ser acessados em sua forma de 16 bits (`AX`, `BX`, `DI`, etc.). Pelo mesmo motivo, os registradores `AX`, `BX`, `CX` e `DX` podem ter suas partes alta e baixa acessadas individualmente, como, por exemplo, `AH` e `AL`.

Apesar de não serem diretamente acessíveis, os registradores `EFLAGS` e `EIP` podem ser citados neste grupo por estarem presentes no contexto de qualquer programa. `EFLAGS` é o registrador de estado do processador, contendo informações sobre estouro de operações aritméticas, condições de igualdade, interrupções, etc. `EIP` aponta, dentro do segmento de código, qual a próxima instrução a ser executada.

3.2.2.2 Registradores de segmentos

O i486 designa alguns registradores especialmente para identificação de segmentos na referências à memória. São eles:

`CS` **:** Identifica o segmento de código de um programa, ou seja, a porção de memória que contém as instruções sendo executadas no momento;

`DS` **:** Identifica o segmento de dados. Todas as instruções que envolvem endereços de dados em memória referenciam-se a este registrador;

`SS :` *Stack segment*, aponta o segmento onde está a pilha de dados de um programa;

`ES, FS e GS :` Apontam segmentos extras de dados, caso seja necessário.

Cada um dos registradores de segmentos possui, além dos 16 bits visíveis ao programador, uma parte "oculta". Essa parte é usada pelos recursos mais avançados de processador (*multitasking*, gerência de memória, etc.) para fins de otimização de desempenho, e é totalmente invisível externamente. Os registradores `FS` e `GS` foram introduzidos somente a partir do 80386.

3.2.2.3 Registradores de sistema

Apesar de não terem uma denominação oficial, esses registradores podem ser assim chamados por terem seu uso restrito unicamente ao sistema operacional. Isto se deve ao fato de serem registradores usados pelos recursos avançados do processador, e seu uso pelas aplicações comprometeria a estabilidade do sistema.

`GDTR e LDTR :` Apontam as tabelas global e local de descritores de segmentos e de *tasks*[1];

`IDTR :` Endereça a tabela de descritores de tratadores de interrupções;

`TR :` Identifica a *task* corrente;

`CR0, CR1, CR2 e CR3 :` Estes são os *Control Registers*, com diversas funções, as quais serão melhor abordadas posteriormente.

[1] O termo *task* representa um processo qualquer do sistema

Além destes, existem ainda os *Trap Registers* (`TR0` a `TR7`) e os *Debug Registers* (`DR0` a `DR7`). Esses registradores têm funções muito restritas, tendo como principal objetivo a depuração de programas, e não têm maior influência no projeto.

3.2.3 Portas de I/O

Os processadores da família 80x86 caracterizam-se por ter seu espaço de I/O diferenciado do espaço de endereçamento de memória. Isto quer dizer que existem instruções específicas para ler e escrever dados em portas de I/O.

Como característica herdada do 8086/8088, o i486 possui um espaço de endereçamento de I/O de 16 bits, ou seja, 65536 portas[2]. A utilização dos computadores IBM-PC determinou a padronização de algumas faixas de portas para determinados dispositivos como, por exemplo, 0x3f8-0x3ff[3], para a porta serial `COM1`.

As instruções para leitura e escrita de dados em portas de I/O são, respectivamente, `in` e `out`. Originalmente, no 8086/8088, essas instruções podiam manipular dados de 8 ou 16 bits. A partir do 80386, elas passaram a suportar também dados de 32 bits (como cada porta é de 8 bits, os bytes de mais alta ordem são escritos nas portas imediatamente subseqüentes).

3.3 Modos de operação

Com o desenvolvimento de recursos cada vez mais avançados em sua linha de processadores, e com a necessidade de mercado de manter compatibilidade com os dispositivos precedentes, a Intel criou os *modos de operação* do 80x86. Essa alternativa possibilitou o desenvolvimento de *software* que utilizasse as novas carac-

[2]Na prática, devido às características de *hardware* dos dispositivos para IBM-PC, esse conjunto foi reduzido para 1024 portas, o que parece ser suficiente até o momento

[3]Esta notação é convenção da linguagem C para representar números em hexadecimal.

terísticas do processador e, ao mesmo tempo, que se pudesse utilizar os programas já desenvolvidos. Os modos de operação e sua aplicabilidade são descritos abaixo.

3.3.1 Modo real

O modo real é o modo de compatibilidade com o 8086/8088. Operando neste modo, o i486 se comporta exatamente como um 8086 muito rápido. Seu conjunto de instruções é reduzido, os dados são de no máximo 16 bits e o *software* desenvolvido para 8086/8088 é perfeitamente executável. Em modo real não existe qualquer recurso avançado de gerência de memória ou de processos, e o espaço de endereçamento é limitado aos 640 Kbytes, dividido em segmentos de 64 Kbytes.

3.3.2 Modo virtual

O modo virtual é um meio termo entre os modos real e protegido. Nesse modo já existe um certo nível de gerência de processos (ou seja, é possível a execução de vários programas concorrentemente) e de memória. Porém, o conjunto de instruções ainda é restrito ao do 8086/8088, e o modelo de memória ainda segue o esquema de segmentos de 64 Kbytes.

3.3.3 Modo protegido

Disponibilizando todos os recursos do processador, o modo protegido é o modo de operação mais completo do i486, e é o modo a ser utilizado por *software* de alta confiabilidade, como um sistema operacional. No modo protegido pode ser usado todo o espaço de endereçamento de 32 bits (4 Gbytes) do processador, e pode ser implementada a multitarefa de forma que cada processo tenha o seu espaço de endereçamento. Neste modo o i486 oferece ainda 4 níveis de privilégios de execução,

podendo ser implementados esquemas de proteção de recursos, além da restrição a algumas instruções privilegiadas.

O gerente de memória implementado opera completamente em modo protegido. Devido à sua importância no projeto, as características de gerência de memória em modo protegido serão descritas em uma seção à parte.

3.4 *Multitasking*

Em diversos sistemas operacionais da atualidade é comum a implementação de *multitasking*, ou multitarefa. A multitarefa consiste na possibilidade de execução aparentemente simultânea de diversos processos no mesmo processador. O que ocorre, na verdade, em sistemas monoprocessados, é que existem realmente vários processos, mas somente *um* por vez utiliza a CPU, enquanto os demais estão aguardando em filas do sistema. Existem diversos critérios de escolha para se determinar qual processo utilizará a CPU em um determinado momento, sendo mais comum a concessão de uma porção de tempo para cada processo, caracterizando um sistema do tipo *time-sharing*. Ao final de sua porção, o processo é "retirado" da CPU, e um outro entra em seu lugar. Quando um processo passa a ocupar a CPU, diz-se que ele foi *escalonado*.

Para que um processo possa continuar sua execução, após ser retirado e recolocado na CPU, suas informações principais devem ser guardadas. Essas informações formam o que se chama *contexto* de um processo. O i486 define uma estrutura especialmente dedicada a guardar o contexto de um processo. Essa estrutura é chamada de TSS — *Task State Segment* — e pode ser comparada ao PCB do sistema UNIX[bac87]. O processador i486 provê um mecanismo automático para troca de contexto utilizando TSS, o que contribui bastante para a diminuição do *overhead* na realização dessa tarefa.

3.5 Gerência de memória em modo protegido

Como citado anteriormente, quando em modo protegido o processador i486 disponibiliza o conjunto completo de recursos para a gerência de memória do sistema. Esses recursos consistem, basicamente, nos mecanismos de *segmentação* e *paginação*. Estes mecanismos são responsáveis pela tradução dos endereços lógicos, gerados pelos programas, em endereços físicos.

3.5.1 Segmentação

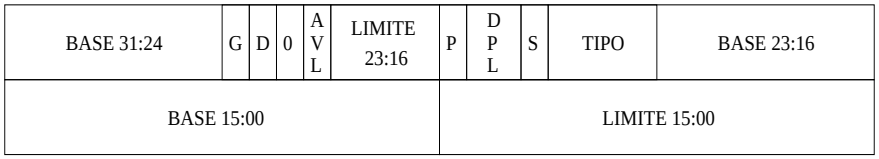
O esquema de segmentos de memória existe mesmo quando o processador está operando em modo protegido. Porém, seu funcionamento é bastante diferente daquele do modo real.

Em modo protegido, um segmento é representado por uma estrutura de dados chamada de *descritor de segmento*. Os descritores de segmentos são armazenados em dois tipos de tabelas: na GDT (*Global Descriptor Table*) e nas LDT (*Local Descriptor Table*). A GDT guarda descritores dos segmentos globais do sistema, como por exemplo segmentos do sistema operacional. As LDT podem ser usadas para guardar descritores de segmentos particulares dos processos.

Com essa nova conotação, os registradores de segmento são usados agora para apontar um descritor de segmento dentro de uma das tabelas. Assim, por exemplo, o código sendo executado em um determinado momento vai estar contido no segmento cujo descritor é apontado por `CS`.

A figura 3.2 mostra o formato de um descritor de segmento. O descritor é dividido em dois tipos de campos, os de controle e os de localização. Os campos de controle contém informações sobre o estado do segmento, tais como o nível de privilégio, o tipo do segmento e a presença ou não na memória. Para definir a localização do segmento, são disponibilizados os campos base e limite. A base é o valor somado ao endereço lógico gerado pelo programa, resultando no endereço

linear. Este, por sua vez, é comparado com o valor do limite. Se for maior, é gerada uma falha de proteção. Esse mecanismo pode ser utilizado para limitar o espaço de endereçamento de um processo e proteger o *software* de sistema.



AVL: Disponivel

BASE: Endereco base do segmento

DPL: Nivel de privilegio do descritor

S: Tipo do descritor (0 = sistema; 1 = aplicacao)

G: Granularidade

LIMITE: Tamanho do segmento

P: Segmento presente

TIPO: Tipo do segmento

D: Tamanho default da operacao (valido em segmentos de codigo)

Figura 3.2: Estrutura de um descritor de segmento.

Cabe aqui lembrar que, para que o i486 operar em modo protegido, é necessária no mínimo a definição de um segmento (para o código) na GDT. Esta tabela pode estar localizada em qualquer lugar da memória física, estando seu endereço físico no registrador `GDTR`.

3.5.1.1 Modo flat

Um efeito muito interessante pode ser conseguido definindo-se todos os segmentos utilizados para terem base igual a zero e limite em 4 Gbytes. Neste caso, chamado *modo flat* de operação, o endereço linear resultante de uma tradução é igual ao endereço lógico. O mecanismo de paginação esclarece melhor a utilidade deste modo. Uma grande vantagem do modo *flat* é a necessidade de definição de alguns poucos segmentos, sendo que todos os processos utilizam os mesmos descritores.

3.5.2 Paginação

Além da segmentação, o i486 oferece um mecanismo de paginação. Este mecanismo é usado em conjunto com a segmentação na tradução de endereços lógicos em endereços físicos. Cabe ressaltar que o uso da paginação é opcional, podendo o sistema operar somente com a segmentação.

Tradicionalmente, para a implementação do mecanismo de paginação, faz-se necessária a presença de tabelas de páginas, contendo a relação entre os endereços lógicos e endereços físicos correspondentes. Ocorre que no i486 existe um grande número de blocos de memória (1048576 blocos de 4 Kbytes, totalizando 4 Gbytes de memória máxima), o que implica em tabelas de páginas muito extensas. Considerando que existe uma tabela de páginas para cada processo, a memória desperdiçada por este controle é muito grande.

Para diminuir a quantidade de memória utilizada no controle dos blocos, o i486 utiliza a tradução de endereços em dois níveis de tabelas. O primeiro nível é chamado de *diretório de páginas*, e contém referências para outras tabelas de páginas contendo, por fim, as referências para os endereços físicos. Esta situação é ser observada na figura 3.3. Os diretórios de páginas e as demais tabelas de páginas ocupam um bloco cada, ou seja 4 Kbytes por tabela.

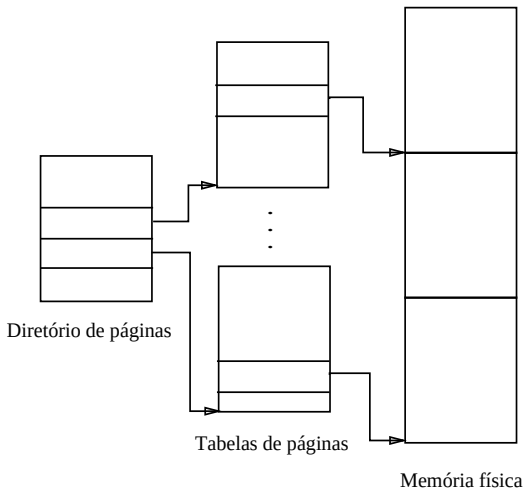


Figura 3.3: Mecanismo de paginação no i486.

Os diretórios e tabelas de páginas são páginas de memória, divididas em várias entradas. O tamanho de cada entrada é de 4 bytes, possuindo diferentes interpretações nos diretórios e tabelas de páginas. Nos diretórios, uma entrada contém o endereço físico de uma tabela de páginas, enquanto que nas tabelas de páginas, a entrada possui o endereço físico do bloco de memória. Os endereços referenciados por estas tabelas são, na verdade, compostos de apenas 20 bits. Esta resolução do endereçamento é justificada pelo fato da memória ser dividida em frames de 4 Kbytes, ou seja, existe apenas a necessidade de armazenar o número do bloco endereçado. Os 12 bits restantes na entrada são utilizados para controle, e serão melhor descritos adiante.

Quando a paginação é usada, o mecanismo de tradução de endereços é diferente daquele empregado na segmentação. O i486 não permite que a segmentação seja desligada, e portanto os dois mecanismos são combinados.

O primeiro passo na tradução de um endereço lógico é a obtenção do endereço *linear*. Este endereço é resultante do processo de tradução de endereços via segmentação. Obtido o endereço linear, o processador passa à fase de paginação. Na tradução para o endereço físico, o endereço linear é dividido em três campos, como mostra a figura 3.4.

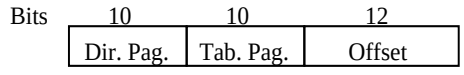


Figura 3.4: Estrutura do endereço linear.

Um registrador especial, chamado `CR3` — *Control Register 3*, também referenciado por PDBR — *Page Directory Base Register*, contém o endereço físico do que se chama diretório de páginas *corrente*. O primeiro campo do endereço linear aponta uma entrada do diretório corrente. Esta entrada contém o endereço físico de uma tabela de páginas. De forma análoga, o segundo campo do endereço linear é usado como índice nessa tabela de páginas, resultando na obtenção do endereço físico da página de memória a ser acessada. Por fim, o terceiro e último campo do endereço

linear representa o deslocamento dentro da página, apontado a posição desejada. A figura 3.5 esquematiza a tradução de um endereço linear em um endereço físico.

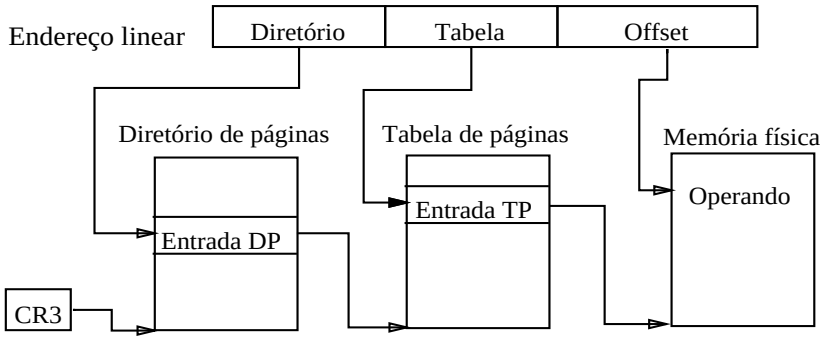


Figura 3.5: Tradução de endereços através da paginação.

Como citado anteriormente, o modo *flat* faz com que os endereços lógicos sejam traduzidos diretamente em endereços lineares. Se o modo *flat* for utilizado juntamente com a paginação, o mecanismo de segmentação se torna transparente, sendo um endereço lógico traduzido unicamente pela paginação.

4 INICIALIZAÇÃO DO SISTEMA

Um dos aspectos mais importantes na implementação do gerente de memória é a inicialização do sistema, desde o momento em que o microcomputador é ligado até o gerente de memória (e outros componentes de um possível sistema operacional) ser carregado e estar apto a prestar seus serviços.

Nos microcomputadores IBM-PC, a inicialização consiste em colocar o processador no modo desejado para trabalho e na configuração de alguns dispositivos externos, tais como controladores de interrupção, temporizadores, etc. Na implementação do gerente de memória, essa tarefa foi dividida em algumas partes, as quais serão vistas a seguir.

4.1 O processo de *boot*

Assim que o computador é ligado, o software residente em ROM (*firmware*), na placa mãe do microcomputador, trata de fazer uma série de verificações no sistema, de forma a assegurar que os principais dispositivos estão funcionando corretamente. Em seguida, o software inicia o processo de *boot*. Este processo consiste em carregar, da primeira unidade de disco, o primeiro bloco de dados, referenciado como *setor de boot*.

O software existente no setor de *boot* é o primeiro passo na inicialização do sistema. Neste ponto, o processador está operando em modo real e alguns dispositivos e estruturas de dados foram inicializados com valores padrão pelo *firmware*. Esta etapa tem duas funções muito importantes. A primeira é, usando a BIOS, carregar demais blocos de dados do disco (os quais conterão a principal parte do sistema operacional, incluindo o gerente de memória) e obter algumas informações sobre o sistema (tamanho da memória física, dispositivos presentes, etc.) [qua89].

Isso deve ser feito ainda nesta etapa, enquanto o processador está em modo real, pois o código da BIOS só pode ser utilizado nesse modo.

A segunda função do *boot* é justamente colocar o processador em modo protegido. Resumidamente, isso é feito ligando-se o bit 0 do registrador `CR0`. Antes, porém, o software de *boot* deve definir, em algum lugar da memória, uma estrutura de dados muito importante para o sistema, a GDT. Conforme visto anteriormente, a GDT é uma tabela que contém descritores de segmentos. Cada descritor identifica um segmento de memória a ser usado no sistema, contendo informações sobre o tamanho do segmento, níveis de privilégio, etc. No gerente de memória implementado, foram definidos 4 segmentos básicos: um segmento de código e um segmento de dados para o *kernel* (com o mais alto nível de privilégio) e outros dois para as aplicações (com nível inferior). Por definição do projeto, estes segmentos são definidos de forma a serem transparentes para todo o sistema, sendo usado o já mencionado modo *flat*.

Assim, inicializada a GDT, e supridos os parâmetros necessários, o processador é colocado em modo protegido e o processo de inicialização segue para a próxima etapa.

4.2 Configuração do sistema - *setup*

Após utilizar os serviços da BIOS e colocar o processador no modo protegido, o software de inicialização entra na fase de *setup*. Esta etapa tem a função de inicializar alguns dispositivos do sistema e preparar o ambiente para o carregamento do sistema operacional (por exemplo, ativar o mecanismo de paginação).

O *setup* inicia com a ativação da linha A20. Essa linha é uma característica dos PCs, e é necessária para a utilização de memória acima de 1 Mbyte. Em seguida, o software define a *Kernel Page Table* (a ser descrita adiante), que será o meio de acesso ao sistema operacional quando da utilização do mecanismo de pagi-

nação. O próximo passo é a definição de tabelas de páginas que farão o mapeamento da memória física. Essas tabelas serão mapeadas no espaço de endereçamento de todos os processos do sistema, como meio de acesso do *kernel* à memória física.

Em seguida, é realizada uma das tarefas mais importantes do *setup*: a ativação do mecanismo de paginação. De forma semelhante à colocação do processador em modo protegido, é necessário o estabelecimento prévio de algumas estruturas de dados. Além da tabela de páginas do *kernel*, deve-se definir, também, uma região da memória para atuar como um espaço de endereçamento temporário durante a inicialização. O endereço dessa área é colocado no registrador `CR3`, e paginação pode, então, ser ativada.

Já utilizando a paginação, o *setup* começa a inicialização dos dispositivos do sistema. Primeiramente, o temporizador do sistema é inicializado de forma a definir uma base de tempo para relógio, escalonamentos, etc. O temporizador utilizado é o presente no chip 8253, componente integrante dos microcomputadores PC. Em seguida, inicializa-se os controladores de interrupção 8259 para trabalhar na faixa estabelecida para o sistema.

Após mais algumas definições, o processo de inicialização passa à última etapa.

4.3 Inicialização do *kernel* - *init*

Como citado na primeira etapa, o *kernel* do sistema operacional (do qual faz parte o gerente de memória) é carregado do disco para a memória utilizando-se os serviços da BIOS. Para que possa entrar em funcionamento, o *kernel* precisa ter algumas de suas estruturas inicializadas. Essa etapa é chamada de *init*.

A inicialização do gerente de memória consiste basicamente na preparação da estrutura de alocação de páginas. Baseado no tamanho da memória física, o *init* passa por todas as páginas de memória formando uma lista encadeada. Essa

lista, chamada de `Free_Frame_List`, é uma das estruturas de dados do gerente de memória, e é vital para o controle de alocação de memória no sistema.

O restante do *init* trabalha com os demais componentes do *kernel*, fazendo inicializações semelhantes àquelas feitas para o gerente de memória. Em sua última parte, o init dá a "partida" no sistema.

A última tarefa do init é criar e disparar o primeiro processo do sistema, chamado *loader*, já utilizando as chamadas do *kernel*. O *loader*, por sua vez, é responsável pela criação e disparo de todos os outros processos do sistema.

5 O GERENTE DE MEMÓRIA

5.1 Estrutura do gerente de memória

A estratégia de gerência de memória adotada no projeto é a segmentação paginada, pois combina características de eficiência e flexibilidade, além de ser o esquema atualmente mais utilizado em sistemas operacionais. Contudo, como visto anteriormente, o i486 não proporciona o mecanismo de paginação pura, existindo a necessidade do uso combinado com a segmentação[1].

De forma a tornar o mecanismo de segmentação do i486 o mais transparente possível, o gerente de memória utiliza o modo *flat*. Como visto anteriormente, no modo *flat* os segmentos de memória são todos sobrepostos, ocupando todo o espaço de endereçamento lógico de 0 a 4 Gbytes. O gerente de memória define quatro segmentos básicos: dois para código e dados do sistema e dois para código e dados das aplicações. Os segmentos de sistema possuem nível de proteção 0, enquanto que os segmentos de aplicação possuem nível de proteção 3. Desta forma, o gerente, e o sistema como um todo, fica protegido de quaisquer ações destrutivas dos programas aplicativos. Estes segmentos são definidos na GDT ainda em tempo de inicialização do sistema, na fase de *boot*.

Apesar do i486 oferecer uma série de recursos que facilitam o *multitasking* e a gerência de memória, existem alguns problemas sérios que necessitam ser sanados. Tais problemas e a forma encontrada pelo gerente para resolvê-los são descritos nas seções subseqüentes.

[1] A segmentação paginada utilizada no gerente de memória é um conceito lógico, e não deve ser confundida com a segmentação do i486

5.1.1 Tabela de páginas do kernel

Quando um processo está em execução, a cada instante é gerado um novo endereço lógico, seja para o *fetch* da próxima instrução ou para a busca de um dado. Para acessar efetivamente a memória, o endereço lógico é traduzido para um endereço físico. No i486, este processo passa por duas etapas: tradução pela segmentação e tradução pela paginação. Como o mecanismo de segmentação não está sendo efetivamente utilizado, o endereço lógico gera um endereço linear com o mesmo valor, o qual é traduzido de acordo com o espaço de endereçamento atual (diretório de páginas), apontado pelo registrador `CR3`.

Contudo, os processos muitas vezes necessitam realizar chamadas ao sistema operacional (por exemplo, para alocar uma nova área de memória). Logo, para que o código do sistema operacional possa ser executado, é necessário que ele esteja *mapeado* em algum ponto conhecido do espaço de endereçamento corrente.

Existe ainda um outro problema, mais grave do que o anterior. Durante a troca de contexto de processos, os valores atuais dos registradores da CPU são salvos em uma TSS e carregados de outra TSS. O registrador `CR3` faz parte do contexto do processo, logo é salvo e em seguida carregado com um novo valor. Porém, imediatamente após a troca deste registrador, a tradução de endereços passa a ser de acordo com o novo espaço de endereçamento. Isto significa que todas as TSS devem estar mapeadas no mesmo endereço lógico, caso contrário, o processador carregará valores inválidos nos registradores, resultando em falha geral do sistema.

A forma mais simples de solucionar estes problemas consiste no mapeamento das áreas de código, dados e pilha do sistema operacional em um endereço lógico comum a todos os espaços de endereçamento. Desta forma, as chamadas de sistema estão acessíveis a todos os processos, no mesmo endereço lógico.

O gerente de memória define este ponto comum a todos os processos como sendo a *tabela de páginas do kernel*. Esta tabela de páginas é organizada de

forma a mapear as áreas de código, dados e pilha do sistema (daí o nome tabela de páginas do *kernel*), além de outras estruturas de controle para o gerenciamento. Este mapeamento é descrito a seguir.

IDT : A primeira entrada da tabela de páginas do kernel é uma referência à IDT — *Interrupt Descriptor Table*. A IDT é uma tabela semelhante à GDT, que contém descritores das rotinas de tratamento de interrupções. Qualquer tipo de interrupção ocorrida no sistema gera uma chamada a uma rotina apontada pela IDT.

O mapeamento da IDT é justificado pela necessidade de manipulação das exceções relativas à gerência de memória e para possibilitar o mapeamento dos tratadores das demais interrupções.

GDT : A GDT, como visto, é a tabela de descritores de segmentos, e o gerente a utiliza constantemente. A necessidade deste mapeamento deve-se ao fato de que a GDT mapeia, além dos segmentos de memória, todas as TSS do sistema, que precisam ser manipuladas no momento da criação de um novo processo.

Tabela de páginas do kernel : Como dito anteriormente, a tabela de páginas do kernel contém dados de controle para a gerência de memória. Desta forma é imprescindível que ela possa ser constantemente consultada e alterada. Para que isto seja possível, a própria tabela é mapeada em uma de suas entradas.

Tabelas de TSS e espaços de endereçamento : Uma TSS é composta de várias informações relativas ao estado de um processo. De forma a permitir que estes dados possam ser manipulados, desde a criação do processo, cada TSS é mapeada individualmente na tabela de páginas do kernel. O número de TSS é arbitrariamente definido pelo sistema, e no planejamento inicial do gerente foi definido em 32.

De forma semelhante, o espaço de endereçamento de um processo, representado por um diretório de páginas, também é constantemente alterado, exigindo o seu mapeamento.

Área do sistema : Pelos motivos citados anteriormente, código, dados e pilha do sistema operacional precisam ser mapeados na tabela de páginas do kernel para que possam ser utilizados na prestação de serviços aos processos.

A tabela de páginas do kernel e suas entradas são visualizadas na figura 5.1.

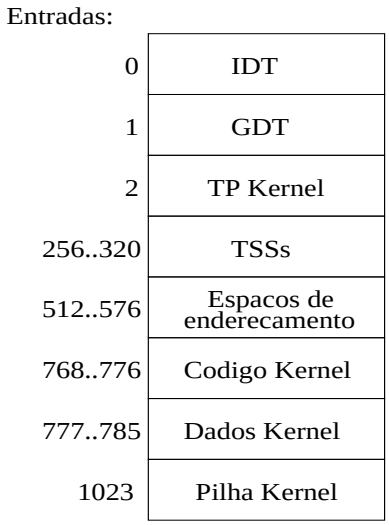


Figura 5.1: Estrutura da tabela de páginas do kernel.

5.1.2 Mapeamento da memória física

O gerente de memória é uma entidade indispensável dentro de um sistema operacional, pois é responsável pelo gerenciamento de toda a memória do sistema. Porém, para realizar as sua funções, o gerente tem a necessidade de acessar livremente qualquer ponto da memória física. Por exemplo, quando um processo requer a alocação de uma área de memória, o gerente tem o dever de alocar e mapear novas páginas no seu espaço de endereçamento. Por questão de segurança, esta nova

área de memória é completamente preenchida com o valor `0x00` antes de ser repassada ao processo. Caso contrário, seria possível a um processo acessar informações ainda presentes em uma página liberada por outro.

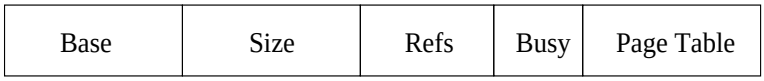
Para permitir que o gerente acesse livremente qualquer ponto da memória, optou-se por mapear, em uma área comum a todos os espaços de endereçamento, toda a memória física. O endereço lógico escolhido para conter este mapeamento é 2 Gbytes. Desta forma, o máximo de memória física gerenciável é de 2 Gbytes, um montante considerável para os padrões atuais.

5.1.3 Segmentos lógicos

Para facilitar a tarefa do gerenciamento, são definidos os objetos de memória denominados segmentos lógicos. Um segmento lógico nada mais é do que uma tabela de páginas contendo o endereço de alguns frames de memória alocados. Os segmentos podem ter no máximo o tamanho de 4 Mbytes, devido a limitação de espaço da tabela de páginas (1024 entradas X tamanho do bloco = 4 Mbytes).

O controle interno dos segmentos lógicos alocados é realizado através de uma tabela de segmentos. Cada entrada desta tabela contém informações relacionadas ao segmento, conforme mostrado na figura 5.2. Cada segmento possui um identificador, e o processo que o conhecer é capaz de utilizá-lo. No caso do gerente de memória, o identificador de segmento consiste de um número inteiro, que e o próprio índice na tabela de segmentos.

Os processos que souberem o identificador de um segmento poderão mapeá-lo no seu espaço de endereçamento, caracterizando o compartilhamento de memória.



Base: base interna do segmento (0 a 4 Mbytes)
Size: tamanho do segmento (máximo 4 Mbytes)
Refs: contador de referências ao segmento
Busy: indica se o segmento está sendo utilizado
Page Table: ponteiro para a tabela de páginas do segmento

Figura 5.2: Formato de uma entrada da tabela de segmentos.

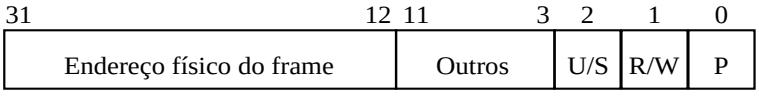
5.2 Características

5.2.1 Proteção da memória

A proteção da memória é necessária para garantir a integridade do sistema. Para atingir este objetivo, o gerente de memória implementado faz uso dos mecanismos de proteção presentes no i486.

A nível de segmentação do processador i486, são definidos quatro tipos de segmentos, sendo dois para os processos em nível usuário e outros dois para nível de sistema, conforme já exposto anteriormente.

Em relação à paginação, a proteção de memória é realizada a nível de página, sendo que cada entrada de diretório e tabela de páginas contém um certo número de bits de controle com este propósito. Estes bits são descritos na figura 5.3.



P − Bit de presença da página
U/S − Permissão Usuario/Supervisor
R/W − Leitura/Escrita

Figura 5.3: Formato de uma entrada em uma tabela de páginas.

As páginas contendo informações do sistema possuem o bit U/S zerado. Se uma aplicação de usuário tenta acessar esta página de memória, é gerada uma exceção reportando o problema.

Para as aplicações, as páginas que contêm áreas de código têm o bit R/W zerado, impossibilitando a alteração do código durante a sua execução.

5.2.2 Compartilhamento

Em sistemas multitarefa é praticamente indispensável a necessidade de comunicação entre processos. Dentre as abordagens existentes, uma delas é através do compartilhamento de memória. Compartilhando uma certa quantidade de dados, dois ou mais processos podem estabelecer um protocolo de comunicação e trocar informações como lhes for necessário.

Tendo em vista essa necessidade, o gerente de memória implementado permite que dois ou mais processos possam compartilhar um mesmo segmento lógico. Ao ser criado, todo segmento de memória recebe um identificador único, que é passado ao processo que o criou. Através deste identificador o processo pode, mais tarde, mapear ou desmapear este segmento de seu espaço de endereçamento, destruí-lo quando não for mais necessário, mudar seu tamanho ou ainda *repassar* o identificador a outros processos. Recebendo um identificador de segmento, um processo pode mapeá-lo em seu espaço de endereçamento e passar a compartilhá-lo, conforme ilustra a figura 5.4.

Como os segmentos lógicos são independentes de localização, um mesmo segmento não precisa ser necessariamente acessado no mesmo endereço lógico, podendo os processos mapeá-lo em qualquer área livre de seu espaço de endereçamento.

Em relação ao aspecto de segurança, assume-se que um processo é responsável pelos segmentos que disponibiliza para compartilhamento. Como o gerente de memória permite que se definam níveis de acesso no mapeamento de um segmento,

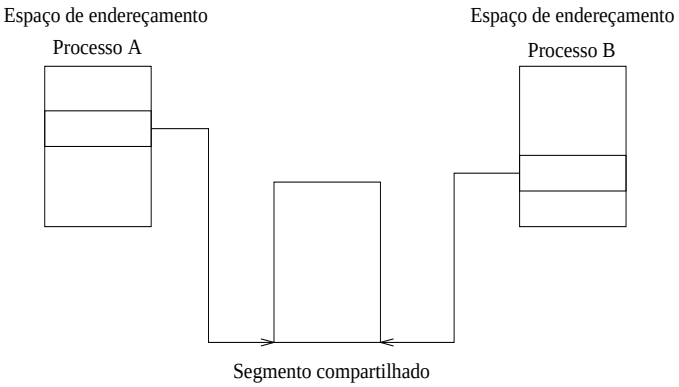


Figura 5.4: Ilustração do compartilhamento de segmentos entre processos.

é possível a implementação de um controle mais elaborado de compartilhamento como, por exemplo, o esquema de *capabilities* sugerido por Tanenbaum[tan92].

5.2.3 Tratamento de exceções

Ao acessar indevidamente um endereço de memória, um processo deve ser imediatamente terminado. Isto é possível pelo tratamento de duas exceções que podem ser geradas ao ocorrer tal problema.

A primeira delas é a exceção `0x0D` — *General Protection Failure*. Esta exceção é gerada sempre que ocorrem violações de proteção. Tais falhas tem uma série de origens, porém, grande parte estão relacionadas a segmentação do i486. Ou seja, quando um processo gerar uma *General Protection Failure*, ele deve ser terminado. Por exemplo, esta exceção pode ser gerada quando um processo, executando em nível 3 de privilégio, acessa os segmentos reservados para o uso do sistema, com nível 0 de privilégio.

A outra exceção relacionada a faltas de memória é a `0x0E` — *Page Fault*. Esta falha tende a ocorrer mais comumente, pois está relacionada com o mecanismo de paginação, e não com os segmentos do i486 que são pouco utilizados. O *Page Fault* ocorre quando uma página, ao ser referenciada, está com o seu bit de presença zerado, ou quando o nível de privilégio da página é maior que o do usuário.

O tratamento desta falta está muitas vezes relacionado com memória virtual, sendo que ao ocorrer um *Page Fault*, significa que a referida página está em disco e necessita ser trazida para a memória principal. Porém, como o gerente de memória não prevê a implementação de memória virtual, esta exceção será tratada somente em um caso, conforme será exposto na seção seguinte; nos demais casos, o processo causador da falta é terminado.

5.2.4 Autoexpansibilidade da pilha

Durante a execução, cada processo possui uma pilha associada. Assim como todos os outros dados relativos a um processo, inclusive o código executável, a pilha é implementada em um segmento de memória. Porém, distintamente dos outros dados, a pilha tem uma dinâmica muito grande, e seu tamanho varia com muita freqüência. Isto ocorre porque a pilha é utilizada na chamada de sub-rotinas, na passagem de parâmetros, entre outros, que são bastante utilizados. Para que esta característica seja atingida, o segmento de pilha deve ter a capacidade de se expandir à medida em que os dados são armazenados. Isso deve ser feito de forma transparente ao processo.

Este é um dos aspectos mais problemáticos no projeto, pois o processador i486 não oferece qualquer mecanismo de apoio à implementação desta característica, como por exemplo avisar o sistema quando uma falta de memória e causada por operações de pilha. A solução encontrada para gerenciar a expansibilidade do segmento de pilha é a utilização da exceção de *Page Fault*, através da análise do endereço da falta.

Ao criar-se um novo processo é definido um segmento de pilha. O tamanho inicial deste segmento é de 4 Kbytes. A medida que a pilha é utilizada, o `SP` — *Stack Pointer* — do processo aproxima-se de uma área não mapeada. No momento em que o `SP` ultrapassa o tamanho inicial do segmento, é gerado um *Page Fault*.

No tratador da exceção é analisado o `SP` do processo, podendo-se concluir se existe a necessidade de expansão da pilha ou se realmente ocorreu uma falta de outra natureza (acidental ou propositalmente). No primeiro caso, o segmento de pilha é aumentado em 4 Kbytes, permitindo que o processo continue a executar normalmente. Por outro lado, confirmada a falha do processo, este é imediatamente abortado.

5.3 Implementação do gerente de memória

Para permitir a facilidade de integração do gerente de memória a um sistema operacional completo, foi definida uma interface simples e de fácil uso, tanto para processos usuários como para outros módulos do sistema que necessitem manipular a memória do sistema.

A padronização utilizada para definir os nomes das chamadas as gerente foi baseada na orientação a objetos. Todas as funções implementadas tem o seguinte formato: objeto_ação. Por exemplo, na criação de um segmento de memória, deve-se chamar a função `segment_create`.

No contexto de gerência de memória são identificados os seguintes objetos: frame — *página de memória*, page_table — *tabela de páginas*, segment — *segmento de memória* — e address_space — *espaço de endereçamento*. A seguir são mostradas as rotinas implementadas pelo gerente de memória, sendo que muitas delas são apenas de uso interno.

5.3.1 Funcionalidade do gerente de memória

Seguindo o padrão definido acima, nesta seção são mostradas as funções disponibilizadas pelo gerente de memória. Algumas rotinas são apenas de uso in-

terno, enquanto outras são utilizadas por processos usuários ou por outros módulos do sistema operacional.

Primeiramente são descritas as funções utilizadas internamente. A um nível mais básico, é necessária a presença de rotinas que permitam a alocação e a desalocação de páginas de memória. Para tanto, são implementadas as funções:

`frame_allocate :` Aloca um frame de memória, retirando o primeiro frame da lista de frames livres e retornando o seu endereço físico.

`frame_dealocate :` Desaloca um frame de memória, recolocando-o na fila de frames livres.

Além da manipulação de frames, necessita-se de funções que preencham tabelas de páginas através da alocação de frames de memória. Tais funções são:

`page_table_map :` Aloca frames de memória de forma a preencher algumas entradas de uma tabela de páginas, de acordo com um tamanho especificado.

`page_table_unmap :` Desaloca frames de memória de uma tabela de páginas.

Conforme definido anteriormente, cada processo no sistema possui seu próprio espaço de endereçamento. As funções de manipulação de espaços de endereçamento permitem sua criação e destruição. De acordo com as definições prévias do gerente de memória, é necessário o mapeamento de certas áreas de memória comuns a todos os processos. Quando um espaço de endereçamento é criado, este mapeamento já é feito automaticamente. Cabe lembrar que essas áreas comuns são de manipulação exclusiva do gerente de memória, não sendo acessíveis às aplicações. Estas funções são:

`address_space_create :` Aloca um frame de memória que contém o diretório de páginas, preenchendo algumas entradas predefinidas.

`address_space_destroy :` Desaloca o frame contendo o diretório de páginas.

O segmento lógico é o principal objeto do gerente de memória, visto que os processos usuários visualizam o espaço de endereçamento como um conjunto de segmentos. Para a manipulação dos segmentos são definidas cinco funções:

`segment_create :` Cria um segmento lógico, alocando uma entrada na tabela de segmentos, criando uma tabela de páginas e mapeando os frames necessários para preencher o segmento.

`segment_destroy :` Remove um segmento da tabela de segmentos, desalocando todos os frames utilizados pelo segmento.

`segment_attach :` Coloca um segmento no diretório de páginas do processo, na posição especificada. Incrementa o contador de referências ao segmento na tabela de segmentos.

`segment_detach :` Remove um segmento do diretório de páginas do processo, decrementando o contador de referências da tabela de segmentos. Se o contador de referências é zerado, a rotina `segment_destroy` é chamada.

`segment_resize :` Aumenta ou diminui o tamanho de um segmento, sempre respeitando o limite máximo de 4 Mbytes por segmento.

5.3.2 Exemplo de utilização dos segmentos por processos

A partir das funções definidas acima, um processo usuário é capaz de lidar facilmente com os segmentos de memória. Para tanto, basta realizar algumas chamadas ao gerente de memória. Para criar e utilizar um novo segmento, um processo deve realizar os seguintes passos:

- Criar um segmento através da rotina *segment_ create*, passando o seu tamanho inicial. Após a criação, é retornado o identificador do novo segmento para o processo
- Para efetivamente utilizar o novo segmento, o processo deve mapeá-lo em seu espaço de endereçamento através da rotina `segment_attach`
- Quando necessário, este segmento pode ter o tamanho modificado através da função `segment_resize`

Após estes passos iniciais, é desejável que outros processos acessem o novo segmento. O compartilhamento pode ser realizado através do conhecimento do identificador do segmento por parte dos outros processos. Assim, para que um segundo processo possa acessar o segmento criado basta mapeá-lo em seu espaço de endereçamento através da rotina `segment_attach`. Cabe salientar que o mesmo segmento pode estar sendo utilizado por diferentes processos em diferentes endereços lógicos (ver figura 5.4).

Por fim, o processo deve desalocar o segmento utilizado. Para tanto, basta chamar a função `segment_detach`. Note-se que não se faz necessária a chamada da rotina `segment_destroy`, pois esta rotina é chamada automaticamente quando o contador de referências é zerado.

6 PUBLICAÇÕES

Durante o desenvolvimento do gerente de memória, foi possível sua publicação e apresentação nos seguintes eventos:

- VI Seminário Catarinense de Iniciação Científica. Universidade Federal de Santa Catarina. Florianópolis, 12 a 13 de setembro de 1996;
- IV Seminário de Iniciação Científica. Universidade Regional do Noroeste do Estado do Rio Grande do Sul. Ijuí, 7 a 11 de outubro de 1996.

Apresentam-se em anexo cópias dos trabalhos publicados.

Além destas publicações, o gerente de memória implementado teve sua primeira participação em eventos internacionais englobado em um projeto maior, chamado "ABOELHA - um nano-*kernel* para um sistema distribuído". O ABOELHA foi publicado no *International Conference on Information Systems Analysis and Synthesis*, ocorrido em Orlando, em julho de 1996, sob o título "*A Concurrent Programming Environment for the i486*".

7 CONCLUSÃO

Este relatório apresentou o desenvolvimento do projeto "Um Gerente de Memória Baseado em Paginação para o Intel 486", durante o ano de 1996. Durante este período foram realizadas duas etapas distintas.

Na primeira etapa do projeto elaborou-se um estudo detalhado do processador i486 e de seus recursos de gerência de memória, além da realização de testes relativos a fase inicialização do processador. Ainda, fez-se um planejamento prévio da estrutura de gerenciamento e da forma como os recursos do i486 deveriam utilizados.

Durante a segunda fase do projeto, definiu-se a interface do gerente com os processos usuários e demais partes do sistema, bem como o tratamento de estouro de pilha. Feita a definição, partiu-se para a implementação do gerente em linguagem C, sendo posteriormente realizados exaustivos testes para verificar a sua funcionalidade.

Após concluído, o gerente de memória pôde ser utilizado conjuntamente com os trabalhos "Um Gerente de Memória para o Nó//" [sav96] e "Um Gerente de Processos para o Nó//" [oli96] para a construção do nano-kernel, denominado ABOELHA. Este kernel em desenvolvimento, tem por objetivo a construção de um ambiente de programação paralela para ser utilizado em atividades de ensino, além de ser possível seu uso na área de automação industrial e na construção de um sistema operacional distríbuido.

Como continuação deste trabalho, os autores sugerem algumas mudanças na estratégia de gerência, como por exemplo a substituição do mapeamento da memória física por um mecanismo alternativo — utilização de entradas livres na tabela de páginas do kernel para a manipulação dos frames físicos. Além disso, sugere-se a utilização de um mecanismo alternativo para o controle da memória ao invés dos segmentos lógicos implementados, os quais limitam-se em 4 Mbytes.

REFERÊNCIAS BIBLIOGRÁFICAS

[avi96] ÁVILA, Rafael B. et alli. *Um Gerente de Memória Baseado em Paginação para o Intel 486*. Santa Rosa, 1996

[bac87] BACH, M. *The Design of the Unix Operating System*. Englewood Cliffs: Prentice-Hall, 1987.

[goo90] GOOR, A. J. van de. *Computer Architecture and Design*. USA: Addison-Wesley, 1990.

[int87] INTEL Co. *80386 System Software Writer's Guide*. Santa Clara: Intel Corporation, 1987.

[int90] INTEL Co. *i486 Programmer's Reference Manual*. Santa Clara: Intel Corporation, 1990.

[lef89] LEFFLER, S. *The Design and Implementation of the 4.3 BSD Unix Operating System*. Reading: Addison-Wesley, 1989.

[oli96] OLIVEIRA, João P. S. de, OLIVEIRA, Jorge R. S. de. *Um Gerente de Processos para o Nó//*. Florianópolis, 1996 (Trabalho de Conclusão de Curso).

[pic96] PICCOLI, Luciano et alli. *Um Gerente de Memória Baseado em Paginação para o Intel 486*. Florianópolis, 1996.

[qua89] QUADROS, Daniel G. A. *PC Assembler*: Usando o BIOS. Rio de Janeiro: Campus, 1989.

[sav96] SAVIETTO, Hélder. *Um Gerente de Memória para o Nó//*. Florianópolis, 1996 (Trabalho de Conclusão de Curso).

[sil90] SILBERSCHATZ, Abraham et alli. *Operating System Concepts*. 3. ed. Massachusetts: Addison-Wesley, 1990.

[tan92] TANENBAUM, Andrew S. *Modern Operating Systems*. Englewood Cliffs: Prentice-Hall, 1992.