UNIVERSIDADE FEDERAL DE SANTA CATARINA

TRANSPARÊNCIA ARQUITETURAL DE SISTEMAS EMBARCADOS: 8 A 64 BITS

Tarcísio Fischer

Florianópolis - SC
2013 / 2

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

TRANSPARÊNCIA ARQUITETURAL DE SISTEMAS EMBARCADOS: 8 A 64 BITS

Tarcísio Fischer

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de bacharel em Ciência da Computação.

Florianópolis - SC
2013 / 2

Tarcísio Fischer

TRANSPARÊNCIA ARQUITETURAL DE SISTEMAS EMBARCADOS: 8 A 64 BITS

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Antônio Augusto Medeiros Fröhlich
Coorientador: Giovani Gracioli

Banca Examinadora

Mateus Krepsky Ludwich
Rafael Luiz Cancian

SUMÁRIO

LISTA DE FIGURAS

LISTA DE SIGLAS

APIC: *Advanced Process Interrupt Controller*

BIOS: *Basic Input/Output System*

BIST: *Built-In Self-Test*

CPU: *Central Processing Unit*

CR: *Control Register*

CS: *Code Segment*

DS: *Data Segment*

EPOS: *Embedded Parallel Operating System*

ES, FS e GS: Nomes de registradores de seleção de segmento

GDT: *Global Descriptor Table*

GDTR: *Global Descriptor Table Register*

HAL: *Hardware Abstraction Layer*

IA-32: *Intel Architecture 32bits*

IA-32e: *Intel Architecture 32bits Extended*

IDT: *Interrupt Descriptor Table*

IDTR: *Interrupt Descriptor Table Register*

LCD*: Liquid Crystal Display*

LDT: *Local Descriptor Table*

MSR*: Model Specific Registers*

MTRR: *Memory Type Range Registers*

MMU: *Memory Management Unit*

PCI: *Peripheral Component Interconnect*

PAE: *Physical Address Extension*

PML4: *Page Map Level 4*

RAM: *Random Access Memory*

SMM: *System Management Mode*

SO: Sistema Operacional

SS: *Stack Segment*

TSS: *Task-State Segment*

System V ABI: System V Application Binary Interface

VM: Virtual Machine

RESUMO

Transparência arquitetural em software significa isolar implementações dependentes de arquitetura em módulos com uma interface comum, de forma que seja possível usar tais módulos para a construção de estruturas mais abstratas independentemente de qual hardware elas vão executar. EPOS (*Embedded Parallel Operating System)* é um sistema operacional cujo desenvolvimento possui uma grande preocupação com a portabilidade. Esse trabalho almeja, como estudo de caso, portar o EPOS para a arquitetura Intel de 64 bits, tendo como resultado um SO (Sistema Operacional) com transparência arquitetural dos 8 aos 64 bits. Dentre as vantagens de ter tal arquitetura disponível no EPOS estão a possibilidade de endereçamento de mais memória além do próprio fato de ter disponível mais uma arquitetura. A validação do trabalho será feita por meio de testes já existentes no EPOS, sendo que a transparência arquitetural será tida como atingida se os mesmos testes executarem e retornarem respostas equivalentes em todas as arquiteturas.

Palavras-chave: Portabilidade. Sistemas operacionais. Transparência arquitetural.

ABSTRACT

Architectural transparency in software means isolating architecturally dependent modules with one common interface, in such a way that it is possible to use them to create more abstract data structures, independently of which hardware they will execute. EPOS (*Embedded Parallel Operating System*) is an operating system where one of the development targets is portability. As a case study, this work aims port EPOS to the Intel 64 bits archictecture, resulting an OS (Operating System) with archictetural transparency from 8 to 64 bits. Among the advantages of having such an architecture available are the possibility of addressing more physical memory, and the fact of having one more archictecture available for EPOS applications. The project validation will be done by the means of preexistent tests in EPOS. The archictectural transparency is reached when the same tests execute in all archictectures in the same way.

Keywords: Portability. Operating Systems. Architectural Transparency.

1 INTRODUÇÃO

Desenvolver um programa de computador que execute em qualquer processador é algo desejável, mas não é uma tarefa simples. Existem várias formas de se alcançar isso, mas um dos problemas comuns decorrentes dessa necessidade é que algum código precise ser escrito ou reescrito por peculiaridades arquiteturais. Resumidamente, a arquitetura descreve o projeto dos componentes de hardware. Isso significa que a configuração e o uso de diferentes arquiteturas interfere no desenvolvimento do software.

O desenvolvimento de uma camada de abstração para toda a parte dependente de arquitetura é uma possível solução para esse problema, pois possibilita que a portabilidade de uma aplicação com uma camada desse tipo se limite ao desenvolvimento dessa camada para a nova arquitetura. Uma forma de obter tal camada é utilizando a chamada HAL (*Hardware Abstraction Layer*). Os sistemas que utilizam HAL usam chamadas de software para atender as requisições. Outra forma é o uso de mediadores de hardware. Eles são entidades que provém em software as funcionalidades de hardware e, em sistemas operacionais orientados a aplicação, diminuem ou removem a necessidade de chamadas de software, possibilitando um bom desempenho sem perder a transparência.

EPOS (*Embedded Parallel Operating System)* é um sistema operacional cujo desenvolvimento tem uma grande preocupação com a portabilidade. Atualmente é possível utilizar o EPOS em diversos processadores de arquiteturas entre 8 até 32 bits, graças a técnicas de programação, por exemplo, meta-programação estática, o uso de mediadores de hardware e famílias de abstrações.

2 OBJETIVOS

O EPOS já tem suporte para várias arquiteturas e plataformas, como alguns modelos de processadores ARM, AVR e IA-32 da Intel. É possível compilar uma versão do EPOS para cada uma das diferentes arquiteturas independentemente.

O objetivo desse trabalho é estender o suporte do EPOS para um processador com arquitetura de 64 bits, nesse caso, a arquitetura Intel64, também chamada de X86 64. Algumas modificações serão necessárias no sistema para suportar a nova arquitetura, por isso, é imprescindível uma preocupação adicional para manter o suporte às arquiteturas anteriores, ou seja, deve continuar sendo possível, ao final do projeto, compilar uma versão do EPOS para as arquiteturas atualmente suportadas.

A nova versão do EPOS será testada por meio de aplicações de exemplo que serão executadas tanto em máquinas virtuais quanto reais, a fim de validar o resultado do trabalho.

3 TÉCNICAS DE PROGRAMAÇÃO

O desenvolvimento de um sistema operacional está longe de ser uma tarefa trivial. Por isso, é útil que se tenha disponível técnicas de programação que auxiliem de forma a melhor organizar e tornar reutilizável partes do código do sistema.

Nas seções posteriores serão mostradas a programação orientada a objetos, que separa o programa em classes e objetos e meta-programação estática, técnica

que permite fazer pré processamento do código fonte.

Em virtude do sistema operacional alvo desse trabalho ter sido escrito na linguagem C++, os exemplos terão uma forte ligação com a mesma. Porém, sempre que possível a UML (*Unified Modeling Language*), será utilizada como forma de apresentação dos exemplos em imagem.

3.1 PROGRAMAÇÃO ORIENTADA A OBJETOS

Há algumas décadas, a programação estruturada era uma das formas mais comuns de implementar um algoritmo (existindo também a programação lógica e a funcional). Logo se percebeu que dividir o software era essencial para a manutenção do mesmo, essa ideia de modularização deu origem à programação modular. Porém, definir um problema do mundo real sem perder a semântica continuava sendo uma tarefa difícil. A programação orientada a objetos surgiu como uma tentativa de minimizar essa perda.

Programação orientada a objetos é um método de implementação no qual programas são organizados como coleções cooperativas de objetos, cada um representando uma instância de uma classe, cujas classes são todas membros de uma hierarquia de classes unidas por relações de herança (BOOCH, 2007).

Cada objeto, ou instância de uma classe, possui atributos (características do objeto) e métodos (competências do objeto). Dessa maneira, programadores podem particionar uma aplicação em pedaços gerenciáveis definindo novos tipos que se aproximam dos conceitos da aplicação. Essa técnica é chamada de abstração de

dados. Quando bem utilizadas, resultam em programas menores, fáceis de entender e de manter (STROUSTRUP, 1997).

Atributos e métodos possuem modificadores de acesso. Eles servem para declarar que um dado atributo ou método pode ou não ser invocado fora do escopo da classe. São três os modificadores: público (podem ser acessados fora do escopo da classe), protegido (podem ser acessados no escopo da classe e subclasses) ou privado (podem ser acessados unicamente dentro do escopo da classe). À prática de declarar privados os atributos de uma classe, sendo eles modificáveis e acessíveis apenas através de métodos, da-se o nome de encapsulamento.

A figura 1 mostra um exemplo de uma classe descrita em UML. A classe representa uma estrutura de dados do tipo pilha. A pilha possui os atributos *content*, representando o conteúdo da pilha, e *size*, representando seu tamanho, ambos privados (Isso pode ser percebido pelo sinal '-' na imagem). Além disso, a pilha possui um construtor e três métodos públicos (Pode ser percebido pelo sinal '+' na imagem): *push(int)*, que insere um novo inteiro no topo da pilha, *pop()*, que remove e retorna o elemento do topo da pilha, e *is_empty()*, que verifica se a pilha está vazia.

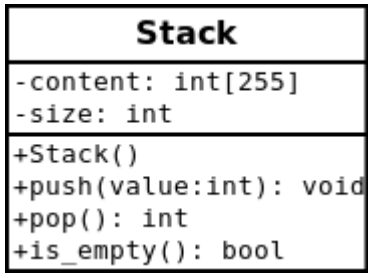


Figura 1: Classe Stack

Classes podem estender outras classes. Quando isso acontece, é dado o nome de herança. Uma classe que estende outra classe é chamada de subclasse ou

classe filha, e possui todos os métodos e atributos da primeira, de acordo com os modificadores de acesso.

É comum em um programa orientado a objetos ter classes com métodos de nome idêntico, mas com implementação diferente. Quando isso acontece, é interessante que se defina uma interface para essas classes. Interfaces são classes sem nenhuma implementação que definem quais serviços as implementações daquela interface devem prover. Interfaces são tão importantes, que é interessante que o programador “programe para uma interface, não para uma implementação” (GAMMA, HELM, et.al; 2000).

Por exemplo, imagine uma interface *Display* responsável por definir os métodos que um *display* deve ter. Mesmo não se tendo conhecimento de detalhes de implementação de um display (pode-se estar usando um *display Liquid Crystal Display* – LCD conectado à porta serial, um monitor de computador ou mesmo uma matriz de *leds* com um pequeno *driver* feito em casa), é possível imaginar que o mesmo deve ter no mínimo um método *clear*, para limpar o *display*, um método *put*, para escrever um caractere na tela e um método *position*, que move o cursor para uma certa posição no *display,* como mostra a figura 2.

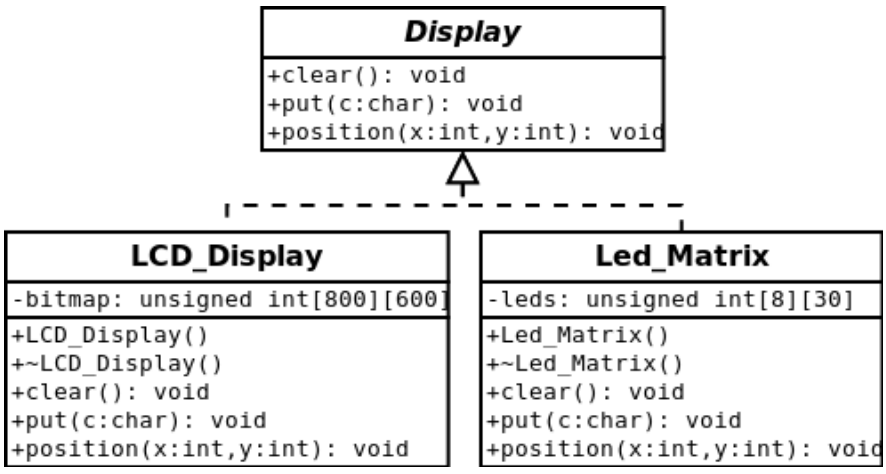


Figura 2: Interface Display e duas classes subclasses

As vezes alguns dos métodos de uma interface possui implementação conhecida. Quando isso acontece, é possível utilizar uma classe abstrata para conter o código conhecido e ainda assim ter métodos sem implementação. Classes abstratas, assim como interfaces, não podem ser instanciadas.

Para exemplificar a utilização de uma classe abstrata, parte-se do exemplo anterior e define-se mais um método para a interface *Display*: o método *put_string(text).* Tal método possui uma possível implementação conhecida, que é iterar sobre os caracteres do parâmetro *text* e chamar o método *put* para cada caractere do mesmo. Dessa forma, transforma-se a interface explicada anteriormente para uma classe abstrata.

A técnica de implementar várias classes concretas com base em uma mesma interface ou uma mesma classe abstrata tem o nome de polimorfismo. O polimorfismo possibilita a declaração de variáveis sem conhecer completamente os detalhes de sua implementação. O conceito de polimorfismo é importante e possibilita uma forma simples para obter transparência arquitetural com uma linguagem orientada a objetos. Por exemplo, é possível ter uma interface para componentes de hardware, como para a CPU (*Central Processing Unit*), de forma que métodos dependentes de arquitetura permaneçam nas implementações dessa interface. Esse é o princípio de mediadores de hardware, que será melhor discutido na seção 5.1.

3.2 META-PROGRAMAÇÃO ESTÁTICA

Meta-programação significa escrever programas que representam e manipulam outros programas ou eles mesmos. O prefixo “meta” denota a propriedade de “ser sobre”, isto é, meta-programas são programas sobre programas (CZARNECKI; EISENECKER, 2000).

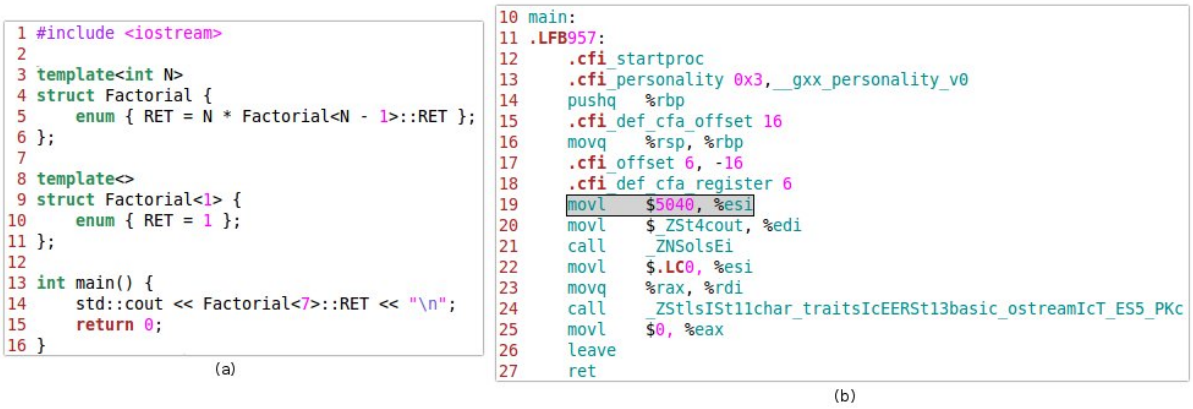
Existem dois tipos de meta-programação: dinâmica e estática. Meta-programação dinâmica diz respeito à auto modificação de programas em tempo de execução. Meta-programação estática diz respeito à auto modificação de programas em tempo de compilação. O uso de meta-programação é dependente de linguagem de programação. Na linguagem C++, por exemplo, é possível utilizar meta-programação estática por meio de *templates*. Meta-programação dinâmica em C++ não será discutida nesse trabalho.

```cpp
#include <iostream>

template<class T, int max_size> class Stack {
private:
    T content[max_size];
    int size;
public:
    Stack() {
        this->size = 0;
    }

    void push(T value) {
        this->content[this->size] = value;
        this->size++;
    }

    T pop() {
        if(this->size == 0) return 0;
        this->size--;
        return this->content[this->size];
    }

    bool is_empty() {
        return this->size == 0;
    }
};

int main() {
    Stack<int, 2> s;
    s.push(1);
    s.push(2);

    while(!s.is_empty()) {
        std::cout << s.pop() << ", ";
    }
    std::cout << "\n";

    Stack<char, 3> t;
    t.push('a');
    t.push('b');
    t.push('c');

    while(!t.is_empty()) {
        std::cout << t.pop() << ", ";
    }
    std::cout << "\n";

    return 0;
}
```

Figura 3: Exemplo da classe Stack utilizando templates.

Em C++, *templates* possibilitam um suporte direto a programação genérica, isso é, utilização de tipos como parâmetros de classes ou funções (STROUSTRUP, 1997). Retomando o exemplo da seção 3.1, uma *Stack* não necessariamente é uma pilha apenas de valores do tipo *int*. É possível criar pilhas de outros tipos, como *char*, *double* ou qualquer outra classe que se possa imaginar. Para evitar de ter que reimplementar a classe para cada tipo de dado, é possível utilizar *templates* para reutilizar uma mesma implementação. A figura 3 mostra uma possível implementação da classe *Stack* utilizando *templates*. A instanciação de um objeto de uma classe que utiliza *template* deve passar o tipo esperado daquela classe como se fosse um parâmetro. Esse dado é utilizado para substituir a classe *template* por uma implementação concreta da classe com o novo tipo.

Templates têm mais utilidade do que simplesmente programação genérica. Como explicado por Czarnecki (2000), com ajuda de algumas outras características da linguagem C++, *templates* formam uma linguagem Turing-completa, ou seja, tão poderosa quanto a própria linguagem C++. Porém, a meta-programação por *template* utiliza o paradigma funcional de programação. A figura 4(a) mostra um exemplo de implementação de um programa que utiliza essa técnica de meta-programação estática para calcular o fatorial de um número. Funções desse tipo são chamadas meta-funções, e são geralmente criadas utilizando recursão.



```
1 #include <iostream>
2
3 template<int N>
4 struct Factorial {
5     enum { RET = N * Factorial<N - 1>::RET };
6 };
7
8 template<>
9 struct Factorial<1> {
10     enum { RET = 1 };
11 };
12
13 int main() {
14     std::cout << Factorial<7>::RET << "\n";
15     return 0;
16 }
```

(a)

```
10 main:
11 .LFB957:
12     .cfi_startproc
13     .cfi_personality 0x3,__gxx_personality_v0
14     pushq   %rbp
15     .cfi_def_cfa_offset 16
16     movq    %rsp, %rbp
17     .cfi_offset 6, -16
18     .cfi_def_cfa_register 6
19     movl    $5040, %esi
20     movl    $_ZSt4cout, %edi
21     call    _ZNSolsEi
22     movl    $.LC0, %esi
23     movq    %rax, %rdi
24     call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
25     movl    $0, %eax
26     leave
27     ret
```

(b)

Figura 4: (a) Exemplo de implementação da função fatorial usando meta-programação estática. (b) Código gerado pelo compilador GNU g++.

O código acima funciona da seguinte forma: Quando o programa faz uma chamada à meta-função *Factorial<N>* e tenta acessar o valor de *RET*, o compilador precisa calcular o valor de *RET* do *enum* de *N * Factorial<N - 1>*. Isso é feito recursivamente até que *N = 1*, quando o compilador usa a especialização para *Factorial<1>*, onde *RET = 1*. Note que sem essa especialização, o compilador não conseguiria gerar código algum, pois ficaria preso na recursão. O resultado da compilação desse código vai ser exatamente o valor do fatorial. Sendo assim, nenhuma chamada de método será necessária e o compilador colocará o valor diretamente no código, como é mostrado no trecho de código gerado, na figura 4(b).

Por vezes, são necessárias informações adicionais na construção de meta-funções. Essas metainformações podem ser representadas utilizando características de tipos, também chamadas de *traits*. Existem três maneiras de representar essas informações: *Member traits* (Figura 5), que é a forma mais simples, onde a metainformação é definida como um membro do tipo, *traits classes* (Figura 6), de forma que ao invés de apenas uma metainformação se tem um pacote amarrado com várias delas, e *traits templates* (Figura 7), que possibilita a especialização de

um *template* para os diferentes tipos e possui uma especialização para cada *traits* associado a um tipo (CZARNECKI; EISENECKER, 2000).

```
template<class ElementType_, int size_>
class Vector {
public:
    typedef ElementType_ ElementType;
    enum { size = size_ };
    // ...
private:
    ElementType elems_[size];
    // ...
};
```

Figura 5: Exemplo de member traits (baseado no código de CZARNECKI; EISENECKER, 2000)

```
enum MatrixShapes {square, diagonal, lower_triang, upper_triang};

class Diagonal10x10MatrixOfDoubles {
public:
    // traits class
    struct Config {
        typedef double ElementType;
        enum {
            rows = 10,
            cols = 10,
            shape = diagonal
        };
    };

    // Operations and data members
    double at(int i, int j);
    // ...
};
```

Figura 6: Exemplo de traits classes (baseado no código de CZARNECKI; EISENECKER, 2000)

```
template<class T>
class numeric_limits {
public:
    // ...
    static const bool is_specialized = false;
    static const bool is_signed = false;
    static const bool is_integer = false;
    // ...
    static const int digits = 0;
    static const int digits10 = 0;
    // ...
    static T min() throw();
    // ...
};

template<>
class numeric_limits<float> {
public:
    // ...
    static const bool is_specialized = true;
    // ...
    static const int digits = 24;
    static const int digits10 = 6;
    // ...
    static float min() throw() { return 1.17549435E-38F; }
};
```

Figura 7: Exemplo de traits templates (baseado no código de CZARNECKI; EISENECKER, 2000)

4 SISTEMAS OPERACIONAIS

Existem basicamente dois tipos de software: software aplicativo, que são utilizados no dia a dia das pessoas para, por exemplo, navegar na web, escutar música e acessar o sistema bancário, e software básico, o software que os aplicativos usam para não ter que trabalhar diretamente com o hardware. Um dos mais importantes softwares básicos é o sistema operacional. Sistema operacional é um software cujo trabalho é controlar os recursos de um computador e disponibilizar uma interface simples para os programadores de aplicativos. Outros exemplos de softwares básicos são os compiladores e interpretadores de linha de comando (TANENBAUM, 2006). A figura 8 ilustra essas diferentes camadas de hardware,

software básico e software aplicativo.

Essa interface de software que o sistema operacional disponibiliza impede que programadores de aplicativos tenham acesso direto ao hardware, evitando que implementações diferentes para um mesmo dispositivo existam, o que poderia causar problemas além de dificultar o desenvolvimento por parte do desenvolvedor de software aplicativo, que teria que entender detalhes específicos de cada dispositivo.

Figura 8: Um sistema de computador consiste em hardware, em programas de sistema e em programas aplicativos. (TANENBAUM, 2006)

Para auxiliar na separação entre software aplicativo e software básico, muitos processadores possuem modos de operação, de forma que softwares básicos podem executar em modo *kernel*, ou modo supervisor, enquanto softwares aplicativos executam em modo usuário. É importante observar que nem todo processador possui modos de operação. Em processadores mais antigos e alguns processadores mais simples com foco em sistemas embarcados, tanto software aplicativo quanto software básico são obrigados a executar com o mesmo nível de privilégio. O uso desses modos de operação impede que qualquer software tenha

acesso aos dispositivos diretamente, sendo obrigados a passar pelo sistema operacional.

4.1 PROCESSOS E THREADS

A segunda principal função que se espera de um sistema operacional é o gerenciamento de recursos, principalmente a gestão de processos e de memória. Processos são programas em execução (SILBERSCHATZ, GALVIN, GAGNE, 2009). O sistema operacional deve gerenciar os processos de forma que dê a impressão que todos os processos estão executando em paralelo, mesmo que só exista uma CPU disponível. O que realmente acontece é que o sistema operacional aloca tempos de uso de CPU para cada processo, seguindo algum critério de escalonamento.

Cada processo possui um código associado (código de máquina a ser executado na máquina), um ponteiro para a instrução atual em execução, uma pilha para valores temporários, uma seção de dados contendo variáveis globais e possivelmente uma segunda estrutura de dados chamada *heap*, onde é possível alocar memória dinamicamente no decorrer da execução do programa (SILBERSCHATZ, GALVIN, GAGNE, 2009).

Todas essas estruturas de dados que compõem um processo são importantes para o sistema operacional conseguir gerenciar os diversos processos em execução. Quando o SO (Sistema Operacional) decide que um novo processo deve continuar sua execução, as estruturas de dados do processo atual são salvas (inclusive o

ponteiro para a instrução em execução) de forma que posteriormente o mesmo processo possa continuar onde havia parado. A figura 9 mostra a estrutura de um processo carregado em memória.

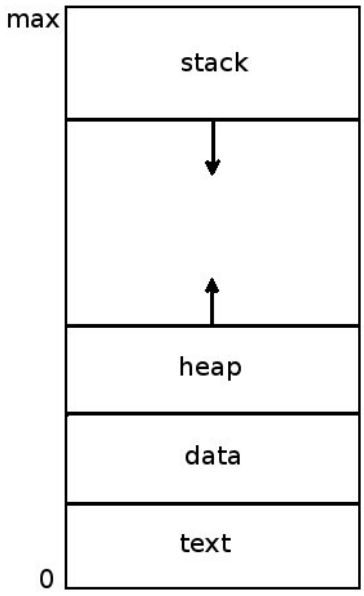


Figura 9: Um processo carregado em memória (SILBERSCHATZ, GALVIN, GAGNE, 2009).

No escopo de um processo, é comum existir a necessidade de executar alguma atividade enquanto se espera pelo resultado de outra. Por exemplo, em um sistema de banco de dados, uma consulta pode estar aguardando dados do disco quando uma segunda consulta chega. Ao invés de aguardar o término da primeira, seria eficiente iniciar a execução da segunda consulta paralelamente. Esse efeito de pseudo paralelismo pode ser atingido utilizando múltiplas linhas de execução, ou *threads*, dentro de um mesmo processo.

Threads compartilham o espaço de endereçamento em um processo, ou seja, os segmentos de código e dados são os mesmos. Porém, cada *thread* possui um ponteiro para instrução diferente, além de possuir uma pilha de chamada de função própria. A figura 10 demonstra a diferença entre três processos rodando cada um em

uma *thread*, e um processo rodando três *threads*.

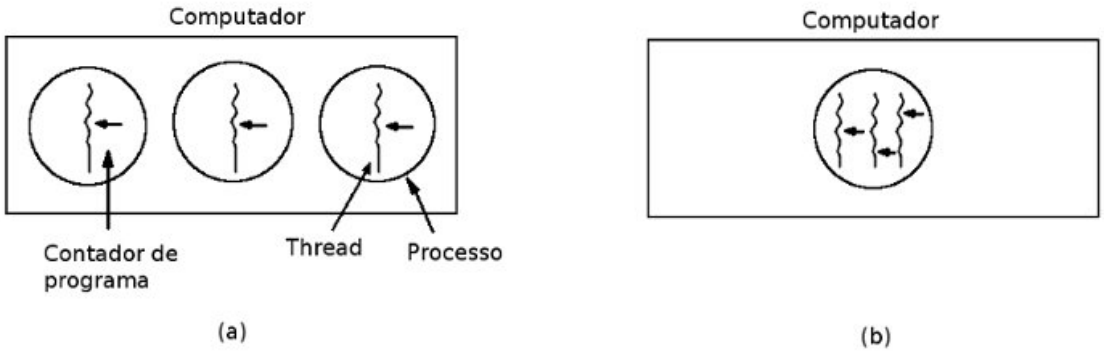


Figura 10: (a) Três processos, cada um com uma *thread*. (b) Um processo com três *threads*. (TANENBAUM, WOODHULL, 1999)

4.2 GERENCIAMENTO DE MEMÓRIA

Memória é um recurso possivelmente escasso, principalmente em sistemas embarcados, que precisa ser gerenciado visando principalmente o tempo, a alocação e desalocação para os diversos processos e a proteção. Proteção no sentido que deseja-se que um espaço de memória destinado à um processo não possa ser modificado ou lido por outro processo.

Existem muitas formas de organização de memória. Geralmente tem-se uma hierarquia de memória em vários níveis, indo do mais próximo do processador, como os registradores, que têm acesso rápido mas são voláteis (ou seja, não persistem quando o sistema é desligado) e caros, até níveis mais distantes como discos rígidos que, apesar de serem não voláteis e baratos, são extremamente lentos. Existem ainda outros tipos intermediários de memória, como a memória cache, que é

bastante rápida, volátil e cara (alguns processadores possuem vários níveis de cache), e a memória RAM (*Random Access Memory* - Memória de acesso randômico), que têm velocidade e preço médios e são voláteis.

Sistemas podem possuir alguns tipos de memória e não possuir outros, um dos trabalhos do SO é gerenciar todos os níveis de memória disponíveis disponibilizando memória o mais rápido possível à aplicação, e mantendo a consistência dos dados. Entre as técnicas mais clássicas de alocação de memória pode-se citar o gerenciamento por mapa de bits, que mantém uma estrutura de dados onde cada bit representa um bloco de memória. A granularidade dos blocos pode ser escolhida, uma granularidade maior significa que um bit representa um bloco maior, ou seja, a alocação de um bloco consome mais memória. Uma granularidade menor significa um mapa de bits maior (TANENBAUM, 2006).

Outra forma de gerenciar o espaço livre de memória é por meio de uma lista encadeada. Cada elemento da lista representa um bloco de tamanho arbitrário com memória alocada ou livre. Cada vez que um elemento é liberado, a estrutura deve ser atualizada de forma que a memória fique livre para ser utilizada novamente (TANENBAUM, 2006).

Memória virtual é uma forma de gerenciamento de memória que possibilita os processos utilizarem mais memória do que a disponível. A ideia é que um processo que necessita de um número grande de memória para executar, digamos 32MB, possa ser executado em uma máquina com apenas 4MB disponível na memória principal, escolhendo esses 4MB que estão atualmente sendo utilizados e colocando o restante em alguma memória de acesso lento, geralmente o disco.

A paginação divide a memória virtual em pedaços mapeados em memória

física. Quando a memória virtual é maior que a memória física, trechos de memória serão mapeados para o mesmo endereço, de forma que um dos dois trechos deve ser colocado em disco enquanto o outro estiver sendo mapeado. Quando um processo faz a requisição de um endereço lógico, esse endereço é traduzido para outro endereço físico de forma transparente. A figura 11 ilustra a tradução de um endereço lógico em físico utilizando paginação de um nível (quando existe apenas uma tabela de página). Paginação será abordada mais profundamente na seção 6.4.

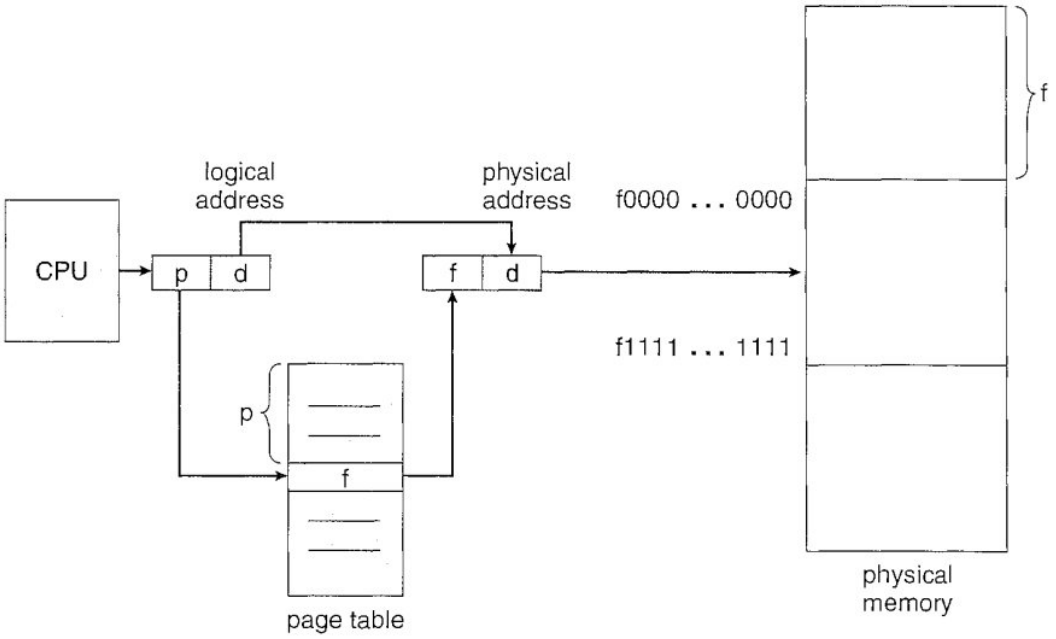


Figura 11: Exemplo de paginação de 1 nível. (SILBERSCHATZ, GALVIN, GAGNE, 2009)

É importante notar que nem sempre essas tecnologias estão disponíveis nos processadores. Por exemplo, processadores mais antigos não possuem nenhum suporte a paginação. No caso dos processadores da arquitetura IA-32 e Intel64, como será mostrado na seção 6.4, é possível configurar a hierarquia de paginação de várias formas, inclusive modificando o tamanho das páginas.

4.3 SISTEMAS OPERACIONAIS ORIENTADOS À APLICAÇÃO

Diversas aplicações que necessitam de alto desempenho acabam esbarrando no sistema operacional, em virtude da necessidade de otimizações próximas do hardware. Outro problema bastante evidente em sistemas embarcados, são os recursos geralmente escassos. Porém, mesmo em sistemas de propósito geral existem aplicativos com esse problema.

Sistemas gerenciadores de bancos de dados são um exemplo. Esses sistemas acabam tendo que reimplementar partes do SO, como gerenciamento de *threads*, gerenciamento de memória, I/O (*Input/Output*), entre outras, na tentativa de ganhar desempenho. Uma forma de resolver o problema é simplesmente ignorar o sistema operacional e trabalhar diretamente com o hardware (Dado que se tenha as devidas permissões). Outra proposta é utilizar um sistema operacional orientado a aplicação (ANDERSON, 1992).

Sistemas operacionais orientados a aplicação são sistemas operacionais que se preocupam primeiramente com as necessidades da aplicação. Os seguintes corolários são conhecidos para esses tipos de sistemas (FRÖHLICH, 2001):

- Um sistema operacional orientado a aplicação sempre está associado à uma aplicação, restringindo o escopo a casos em que se conhece a aplicação antes do desenvolvimento do sistema operacional, em sistemas computacionais dedicados.
- Deve sempre implementar as funcionalidades necessárias e suficientes para suportar a aplicação. A restrição extra referente a não implementação de mais funcionalidades do que as esperadas pela aplicação é importante pois afeta a

qualidade do sistema (desempenho, utilização de recursos e configurabilidade, por exemplo).

- As funcionalidades do sistema operacional devem ser entregues à aplicação correspondente quando elas forem solicitadas.

Dessa forma, o SO terá como finalidade suportar os requisitos necessários para a execução da aplicação. Assim, é possível ter um ganho de desempenho, por exemplo, com o fato de que pode não ser necessário gerar código para algumas partes do SO que nunca são utilizadas pela aplicação, diminuindo o tamanho final do programa.

EPOS (*Embedded Parallel Operating System*) é um sistema operacional orientado a aplicação projetado com o escopo voltado para a funcionalidade, customização e eficiência.

5 EPOS - EMBEDDED PARALLEL OPERATING SYSTEM

EPOS é um sistema operacional orientado a aplicação atualmente desenvolvido no laboratório do LISHA (Laboratório de Integração de Software e Hardware) na UFSC (Universidade Federal de Santa Catarina). Ele possui implementação para várias arquiteturas, incluindo AVR8, ARM e MIPS. Todas as arquiteturas utilizam as mesmas abstrações, tendo apenas as partes dependentes de arquitetura nos mediadores de hardware implementadas de forma distinta. Os conceitos de abstrações e mediadores de hardware serão explicadas nesse capítulo.

Esse capítulo é dividido em 7 partes: A seção 5.1 introduz o conceito de

mediador de hardware, uma forma de representar componentes de hardware no sistema, enquanto a seção 5.2 explica as abstrações independentes de arquitetura disponíveis no EPOS. A seguir, quatro pontos importantes do EPOS serão mostrados no escopo da arquitetura IA32, a fim de possibilitar uma comparação com a arquitetura x86 64 no capítulo posterior: O processo de compilação, na seção 5.3, o processo de inicialização, na seção 5.4, a gerencia de memória, na seção 5.5 e finalmente o funcionamento do sistema de interrupções na seção 5.6.

5.1 MEDIADORES DE HARDWARE

Existem duas abordagens populares para manter portabilidade de sistemas operacionais em diferentes hardwares: Máquinas virtuais e HAL (*Hardware Abstraction Layer*) (POLPETA, FRÖHLICH, 2004).

Para aplicações que utilizam máquinas virtuais (*VM – Virtual Machine*), a própria VM é a camada de abstração de hardware. Para SOs que possuem HAL, como sistemas UNIX, as camadas de abstração de hardware provêm esse serviço por meio de chamadas de software. Uma terceira solução é o uso de mediadores de hardware. A principal vantagem do uso de mediadores é a transparência da implementação de hardware de cada um dos componentes (FRÖHLICH, 2001).

Mediadores são classes criadas para representar algum componente de hardware no sistema operacional. Funcionam de forma equivalente aos *device drivers* dos sistemas UNIX, porém, não constroem a tradicional HAL, ao invés disso, diluem o código do mediador nas abstrações em tempo de compilação por meio de

meta-programação estática e *inline assembly* (GRACIOLI; FRÖHLICH; PELLIZZONI; FISCHMEISTER, 2013). Os mediadores de hardware podem possuir uma interface chamada interface inflada. Cada interface inflada possui os métodos de todas as implementações daquela interface, sendo possível a utilização dos mesmos pelas abstrações.

Por exemplo, pode-se criar um mediador de CPU, que se comporta de de forma distinta para processadores Intel, ARM e AVR. A figura 12 apresenta o diagrama de classes dos mediadores da CPU presentes no EPOS. A classe base *CPU_Common* funciona como uma interface inflada para as diversas implementações do mediador. Quando compilado, o sistema operacional engloba as funcionalidades do mediador no código da aplicação, por meio de meta-programação estática e otimização de código. Tal fato é ilustrado na figura 13, que mostra a arquitetura do sistema antes e depois da compilação do mesmo.

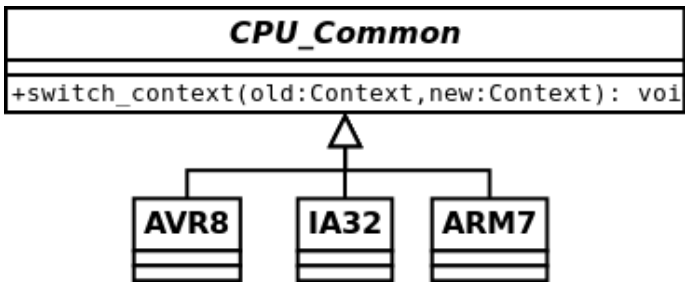


Figura 12: UML dos mediadores da CPU no EPOS. Os atributos e métodos não estão presentes.

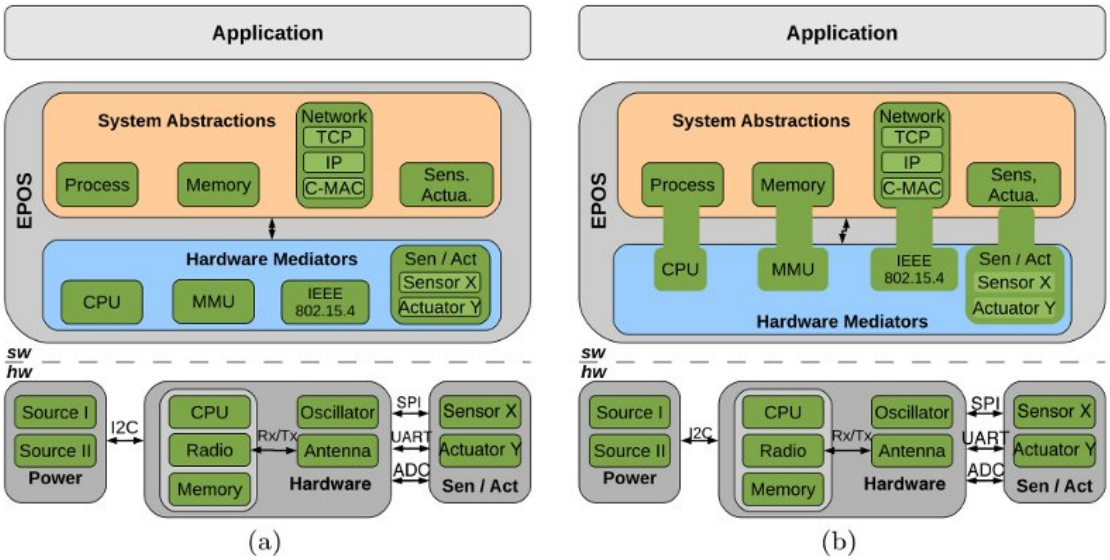


Figura 13: Durante a compilação do sistema, os mediadores de hardware são diluídos nos componentes. (a) Mediadores de hardware antes da compilação e (b) Mediadores de hardware depois da compilação (GRACIOLI; FRÖHLICH; PELLIZZONI; FISCHMEISTER, 2013).

5.2 ABSTRAÇÕES

Softwares aplicativos muitas vezes possuem códigos que podem ser reutilizados em outros aplicativos. Quando isso acontece, é interessante derivar isso do software de modo que seja uma abstração compartilhada. Em questão de sistemas operacionais, podemos retirar abstrações clássicas da literatura e temos aí alguns exemplos. O conceito de T*hread*, por exemplo, é independente de arquitetura e possui uma interface conhecida. De fato, o EPOS possui uma classe *Thread*, que pode ser utilizada pelas aplicações. Por outro lado, a implementação dos métodos da *Thread* podem utilizar informações dependentes de arquitetura. Quando isso acontece, é necessário utilizar um mediador de hardware para manter a

transparência arquitetural da abstração. A figura 14 mostra a dependência da classe Thread de um mediador da CPU para poder efetuar a troca de contexto.

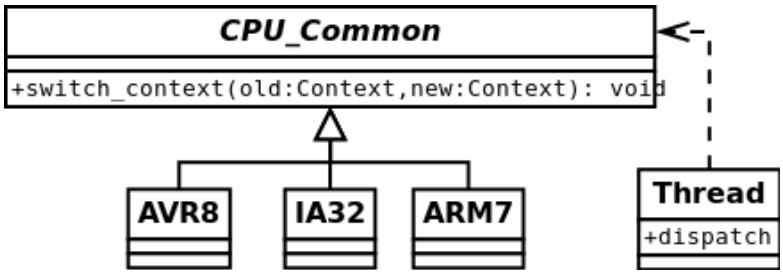


Figura 14: Dependência entre a classe *Thread* e o mediador da CPU.

Outros conceitos clássicos de sistemas operacionais podem se tornar abstrações para o sistema. Por exemplo, o EPOS possui uma abstração chamada *Segment.* É importante notar que essa abstração nada tem a ver com o conceito de segmento da arquitetura dos processadores da Intel. A abstração *Segment* do EPOS representa nada mais que um pedaço de memória.

Essa abstração necessita de conhecimentos sobre o gerenciador de memória presente no sistema. Um mediador de MMU (*Memory Management Unit*) abstrai os detalhes de hardware enquanto a abstração *Segment* usa seus métodos para prover ao programador de aplicação uma forma fácil de representar segmentos de memória na aplicação. Como será visto posteriormente, a MMU também possibilita uma base para a programação de alocação de memória. A figura 15 mostra a dependência entre a abstração *Segment* e o mediador da MMU. O mediador da MMU será melhor detalhado na seção 5.5.

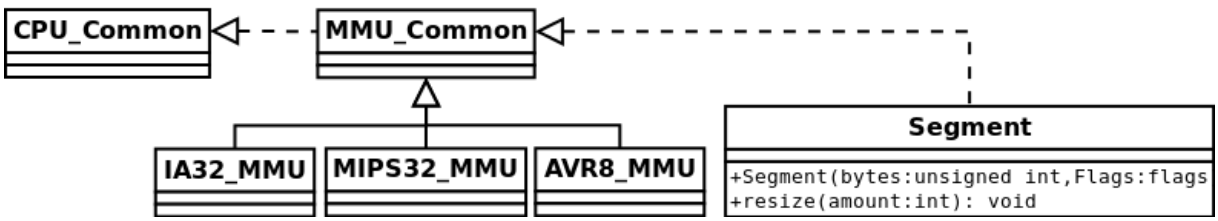


Figura 15: Dependência entre a abstração *Segment* e o mediador da MMU

5.3 PROCESSO DE COMPILAÇÃO

O EPOS utiliza o compilador GCC (*Gnu C Compiler*) compilado com a *newlib* ao invés da GLibc. A GLibc (*Gnu C Library*), assim como a *newlib*, é uma implementação da biblioteca padrão do C. Ela possui algumas definições e funções básicas que podem ser usadas nos programas.

Além disso, ele possui uma série de arquivos de construção, os *makefiles*. Esses arquivos são utilizados por um programa chamado *Make*, uma ferramenta que auxilia o processo de compilação. Cada *makefile*, dentro de cada um dos diversos pacotes do EPOS, define como tal pacote deve ser compilado. Por exemplo, o pacote *boot*, que possui arquivos relacionados ao *boot* do sistema operacional, possui um *makefile* específico que gera código de 16bits a partir de um código fonte escrito em *assembly*. Muitas das definições de constantes e regras gerais de compilação são inseridas em um arquivo compartilhado aos diversos *makefiles*, o *makedefs*. Esse arquivo define a arquitetura, a máquina e até o compilador que será utilizado no processo de compilação, com base nas configurações setadas no *Traits<Build>*. A figura 16 mostra uma possível configuração para o *Traits<Build>* quando utilizada a arquitetura x86 64.

```
16 template <> struct Traits<Build>
17 {
18     enum {LIBRARY, BUILTIN};
19     static const unsigned int MODE = LIBRARY;
20
21     enum {IA32, X86_64};
22     static const unsigned int ARCH = X86_64;
23
24     enum {PC};
25     static const unsigned int MACH = PC;
26 };
```

Figura 16: Exemplo de configuração de um Traits de construção do EPOS.

O EPOS conta com duas importantes ferramentas desenvolvidas para serem usadas no processo de compilação, o eposcc e o eposmkbi. O eposcc (*EPOS C Compiler*), é na verdade um *script* para compilar de forma simples aplicações para o EPOS. O script se encarrega de fazer toda a ligação e configurações necessárias para, dado o código de uma aplicação, gerar código que execute tal aplicação no sistema operacional.

A segunda ferramenta, chamada eposmkbi (*EPOS Make Boot Image*), é um programa que, dado uma aplicação para o EPOS, junta a mesma com o sistema operacional, formando uma imagem completa do mesmo (refere-se aqui como imagem a virtualização do conteúdo de um disco. Geralmente possui a extensão *img*), inclusive colocando os números mágicos para disco inicializável (no caso dos PCs, coloca-se o valor hexadecimal 0x55AA nos bytes 510 e 511 da imagem). Adicionalmente, o eposmkbi adiciona metainformações na imagem, como dados do tamanho do segmento de código e de dados da aplicação, dados para organização de memória, entre outros.

5.4 PROCESSO DE INICIALIZAÇÃO

O EPOS possui três principais formas de compilação. A primeira se chama *builtin*, onde o sistema operacional é compilado separadamente da aplicação e a mesma faz chamadas de sistema para ter acesso ao SO. A segunda tem o nome de *library*, que compila a aplicação junto com o sistema operacional, e a aplicação tem acesso a todas as funções do SO. E a terceira modo *kernel*, onde a aplicação e o

kernel do sistema operacional são separados de forma a ser necessário o uso de chamadas de sistema no lado da aplicação para ter acesso às funções do sistema operacional. As três formas possuem inicialização (processo de *boot* e primeiras configurações do sistema) semelhante.

Como será explicado na seção 6.3, ao ligar um processador da arquitetura Intel x86, o mesmo faz alguns testes próprios, inicia registradores, e executa um código em um endereço especifico, que é o endereço de uma memória EPROM da BIOS, que por sua vez carrega os primeiros 512 bytes do primeiro setor do disco no endereço de memória 0x7c00.

Como no *boot* o processador ainda está no modo real, é importante ter o cuidado de compilar tal código com instruções de 16bits. No EPOS, o código do *boot* é um código específico cujo principal objetivo é carregar do disco o resto do código do SO. Adicionalmente, um trabalho é feito para que ainda no *boot* o modo protegido (32 bits) seja inicializado. No final do processo, executa-se um *jump* para o *pc setup,* que é um segundo nível de inicialização onde já existe código compilado com instruções de 32 bits e onde são inicializadas as estruturas de dados e a paginação. Nesse momento, o EPOS calibra os *timers*, constrói os modelos de memória física e virtual, a IDT e a GDT, as tabelas de páginas, inicializa paginação, e carrega o ELF da aplicação. Por último, o endereço de inicialização dos construtores é chamado. Tal processo é melhor ilustrado no fluxograma da imagem 17. É importante mencionar que o fluxograma ignora algumas etapas de pouca importância para o entendimento desse processo, por exemplo, a escrita de informações na tela, a cópia e utilização de metainformações do EPOS, entre outras.

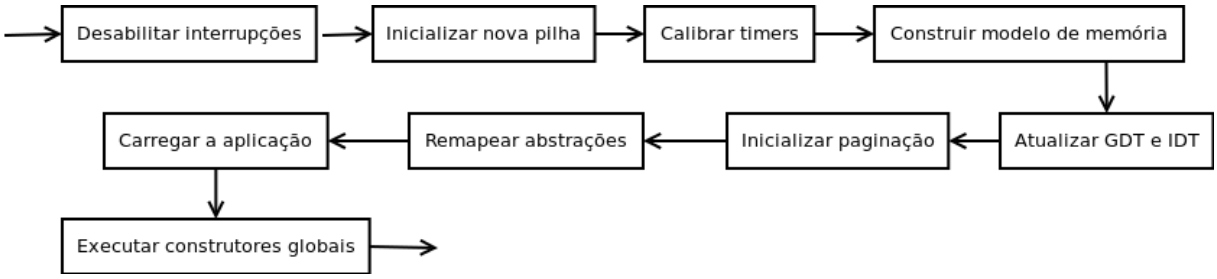


Figura 17: Fluxograma do código executado no PC_Setup

Por causa da ordem em que os objetos são linkados, existe também uma ordem de execução dos construtores globais do sistema. Os construtores mais importantes são o construtor do *First_Object*, cujo propósito é simplesmente servir de uma entrada bem conhecida para o primeiro objeto do sistema, o *Init_System* que, de acordo com as configurações de *traits*, inicia a *heap* do sistema, a máquina e as abstrações por meio de métodos estáticos, e o *Init_Application*, que inicia a *heap* da aplicação. No caso da máquina (*Machine*) PC, são inicializados o vetor de interrupções, o PCI (*Peripheral Component Interconnect*), o temporizador e a *ethernet*. Depois disso, as abstrações do sistema são inicializadas: O alarme e as tarefas (T*asks*).

Terminadas as inicializações do sistema, os construtores globais da aplicação executam, mas antes que a aplicação seja finalmente iniciada, um último construtor é chamado, o chamado *Init_First*. Esse construtor global chama o construtor da primeira *thread* do sistema. Tal *thread* é configurada com o endereço inicial da aplicação, passando assim o fluxo de execução à aplicação. O processo de inicialização do EPOS é mostrado de forma resumida no diagrama da figura 18.

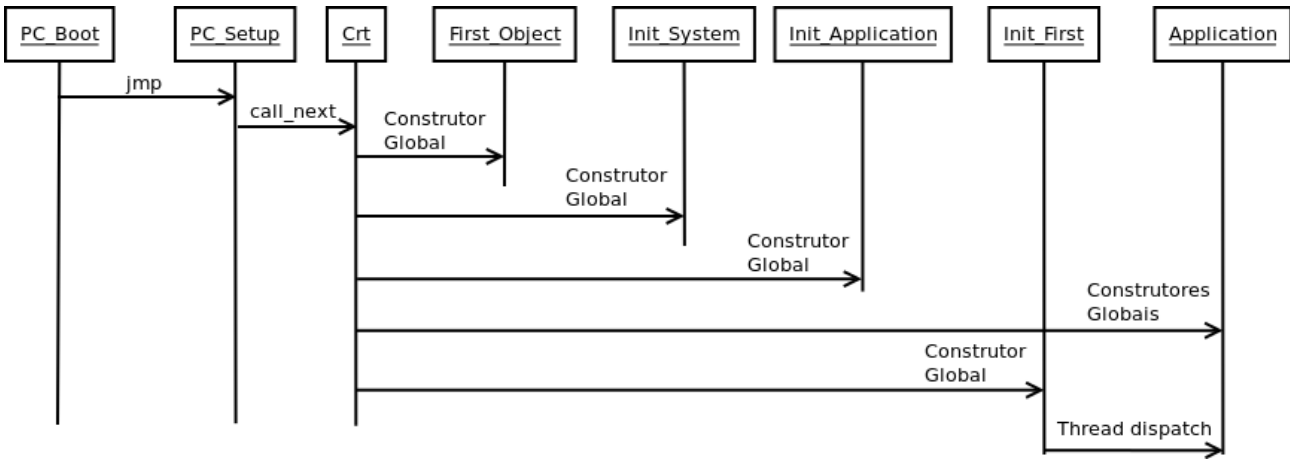


Figura 18: Processo de inicialização do EPOS em alto nível.

5.5 GERÊNCIA DE MEMÓRIA

O EPOS possui um mediador de MMU, que é um *template* para uma classe que possui métodos de gestão de memória (alocar, liberar e fazer cálculos com as páginas de memória). O *template* possibilita a configuração do número de bits de deslocamento, páginas e diretórios, sendo esses valores dependentes dos níveis de paginação da arquitetura-alvo. A memória livre é organizada com ajuda de uma lista de páginas livres.

Como comentado por Hoeller (2004), existe um conjunto de estruturas de dados relacionadas à família de mediadores de memória, onde destacam-se o *Chunk*, a *Page_Table e* o *Page_Directory*. *Chunk* é uma abstração interna da MMU que possibilita o mapeamento de memória em memória física (seja ela contígua ou não). *Page_Table* faz o papel de uma tabela de páginas no conceito do EPOS. Ela é utilizada por *Chunk* e possibilita a alocação de memória virtual no sistema.

Finalmente, *Page_Directory* representa o terceiro e último nível de paginação utilizado, por exemplo, para o esquema de paginação sem PAE (Physical Address Extension) na arquitetura IA-32. Um diagrama de classes das estruturas comentadas está presente na imagem 19.

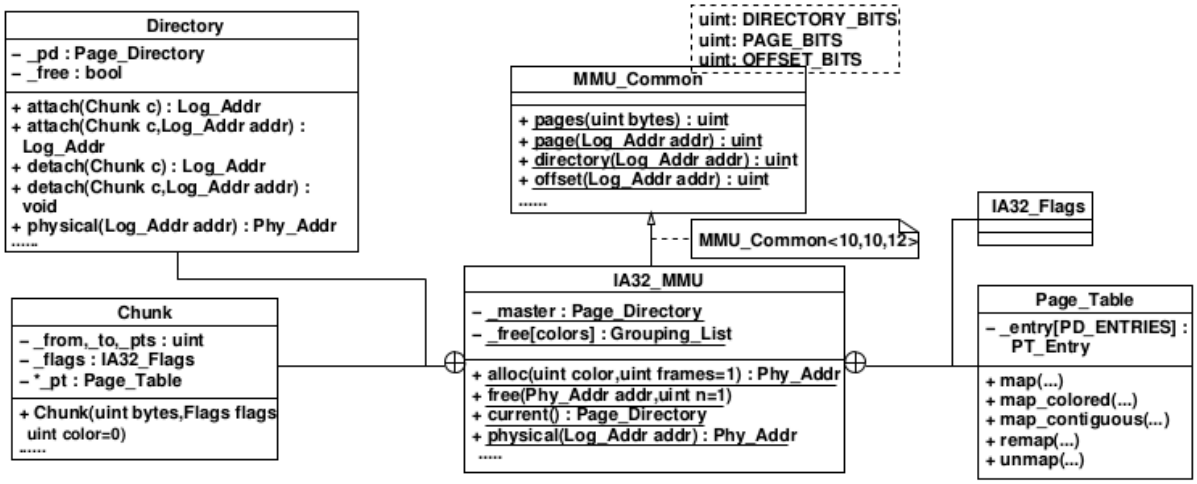


Figura 19: Diagrama de classes do mediador da MMU. (GRACIOLI, FRÖLICH, 2013)

A alocação dinâmica de memória no EPOS é feita por meio das funções *malloc*, *free*, *kmalloc, kfree, new* e *delete*. Essas funções utilizam uma abstração chamada *Heap.* O sistema configura o espaço de *Heap* alocável no construtor *Init_Application* (como comentado na seção 5.4), onde ele define como livre um determinado tamanho de memória disponível, configurável por meio do *Traits<System>::HEAP_SIZE* e *Traits<Application>::HEAP_SIZE*. Dessa forma, quando uma alocação de memória é necessária, a *Heap* consulta sua lista de memória livre disponível e reserva para o usuário, caso possível, um determinado tamanho de memória.

5.6 INTERRUPÇÕES

Na inicialização do sistema, o EPOS evita ao máximo interrupções, cadastrando uma função de pânico caso alguma interrupção ocorra e sempre que possível mantendo as interrupções desabilitadas. As interrupções são inicializadas no construtor *Init_System* (como comentado na seção 5.4), onde é cadastrada uma única interrupção de nome *int_dispatch*, para cada uma das interrupções do sistema. Como mostra a figura 20, existe uma tabela de interrupções do EPOS, que mapeia identificadores de interrupção em funções. *int_dispatch* pega o identificador da interrupção corrente e chama a função determinada na tabela.

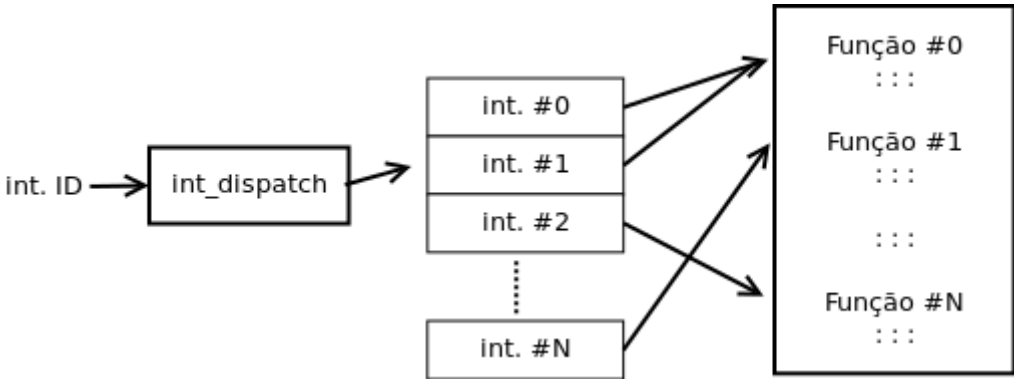


Figura 20:Funcionamento do *int_dispatch* envolve uma tabela de interrupções.

Ainda no *Init_System*, a tabela de interrupções é populada com as seguintes entradas: falta de página (*Page Fault*), dupla exceção (*Double Fault*), exceção geral de proteção (*General Protection Fault*) e *device* não presente (*Device not available fault*), cada uma dessas são mapeadas para uma função específica. Todas as outras exceções e interrupções são mapeadas para uma função de erro genérica.

De forma geral, são esses os principais pontos a se considerar sobre a implementação do EPOS para a arquitetura IA32. Como comentado no início desse

capítulo, tais informações serão utilizadas posteriormente de forma comparativa com a nova implementação do sistema operacional, para a arquitetura Intel x86 64.

6 ARQUITETURAS IA-32 E INTEL 64

A arquitetura IA-32 é encontrada em milhares de computadores pessoais atualmente, em processadores da família Intel como Pentium, Core2Quad e vários outros. Todos os processadores da Intel mantêm compatibilidade com versões anteriores, dessa maneira, é interessante ter um panorama histórico dos produtos e das tecnologias que cada geração inseriu na arquitetura. Informações históricas foram retiradas do manual de desenvolvedor da Intel, volume 1.

Os processadores 8086/8088 de 16 bits precederam a família IA-32. Eles introduziram a ideia de segmentação de memória. Com isso, era possível endereçar um total de 1MB divididos em quatro segmentos de 256KB. Posteriormente, o processador 286 incluiu um novo modo de operação, chamado modo protegido (*protected mode*). Esse modo de operação possui registradores de segmentos, que apontam para tabelas de descritores. Cada descritor provê um endereço de base de 24 bits com até 16MB de memória física, suporte a gerenciamento de memória virtual, mecanismo de proteção como segmentos de apenas leitura ou escrita, entre outros.

O 386 foi primeiro processador de 32 bits da família IA-32. Os registradores de propósito geral aumentaram de 16 para 32 bits, sendo que os primeiros 16 bits de

cada registrador mantiveram suas características, para garantir compatibilidade com as versões anteriores. Outras novidades foram o suporte a endereçar até 4GB de memória tanto em um usando apenas segmentação quanto sem segmentação. Além disso, adicionou paginação, com páginas de tamanho 4KB fixos, e paralelismo interno. Seu sucessor, o 486, aumentou ainda mais esse paralelismo interno, com ajuda de um pipeline de cinco estágios na decodificação e execução de instruções. O 486 também tinha uma cache de 8KB e uma unidade de processamento de ponto flutuante (FPU).

O processador Pentium aumentou a cache interna, adicionou um segundo pipeline de execução, um mecanismo de previsão de *branchs* (*branch-prediction*), uma APIC (*Advanced Process Interrupt Controller*), para suportar múltiplos processadores, entre outras tecnologias. Versões posteriores ao Pentium melhoraram as tecnologias existentes, aumentando o tamanho da cache e adicionando outros níveis de cache, aumentando o paralelismo na execução de instruções, expandindo o conjunto de instruções, etc. Na sua versão 4, o Pentium veio com uma versão com a nova arquitetura, chamada Intel64 ou X86 64. Existem também versões dos processadores com vários núcleos (multicores). A tecnologia multicore depende de inicialização por software, sendo por padrão desabilitada.

Essa nova arquitetura aumentou tanto o espaço de endereçamento virtual, suportando endereços de até 40 bits, quanto o físico, suportando entre 36 até 52 bits dependendo do processador. Além disso, introduziu um novo modo de operação, chamado IA-32e (*Intel Architecture 32 extended*), que é dividido em dois sub-modos: modo compatibilidade, possibilitando a utilização do processador com software de 32bits, e modo 64-bit, que possibilita executar programas de 64 bits.

6.1 VISÃO GERAL DA ARQUITETURA

A arquitetura IA-32 teve início com a família de processadores 386. Ela suporta vários modos de operação (Modo real, modo protegido, modo 8086 virtual e modo de gerenciamento do sistema. Os modos de operação serão vistos mais detalhadamente na próxima seção), métodos de gerenciamento e proteção de memória (seção 6.4), gerenciamento de interrupções e exceções (seção 6.5).

Ambas as arquiteturas são máquinas *little endian*. Isso significa que os bytes de uma palavra são numerados do bit menos significativo para o mais significativo. O conjunto de instruções do processador possui mais de 300 instruções disponíveis, algumas das quais podem ainda conter prefixos para variar sua utilização. Como exemplos de prefixos podem ser citados o prefixo LOCK, que força uma operação que garante uso exclusivo de memória compartilhada, em um ambiente com múltiplos processadores, prefixo de repetição, faz com que uma instrução seja repetida para cada elemento de uma *string* e prefixos de previsão, que podem auxiliar o processador a decidir qual o provável caminho que um *branch* condicional irá seguir. As instruções operam sobre registradores, constantes e variáveis na memória principal.

Existem oito registradores de propósito geral na arquitetura IA-32, sendo eles eax (*accumulator register*), ebx (*base register*), ecx (*counter register*), edx (*data register*), esi (*source register*), edi (*destination register*), ebp (*base pointer register*) e esp (*stack pointer*). O prefixo “e” significa *extended*. Esses são os mesmos registradores da arquitetura anterior de 16 bits estendidos para 32 bits. Os registradores podem também ser chamados simplesmente de r0 até r7. A arquitetura

Intel64 adiciona mais 8 registradores de nome r8 à r15. Além disso, os registradores têm seus prefixos modificados de “e” para “r” (rax, rbx...). Para se referir à pedaços menores de um registrador, é possível dar seu nome de acordo com seu tamanho. Por exemplo, rax é um registrador de 64 bits, eax são os 32 bits menos significativos, ax são os 16 bits menos significativos, ah são 8 bits mais significativos de ax e al são os 8 bits menos significativos de ax. Tal estrutura é ilustrada na figura 21.

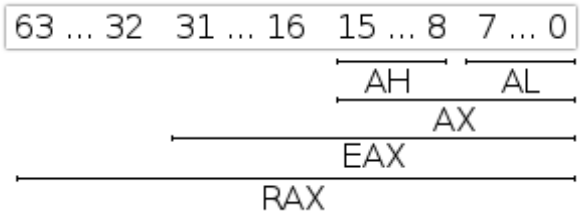


Figura 21: Nome dos registradores de acordo com os bits.

Além dos registradores de propósito geral, existem registradores importantes de status e de controle do processador, entre os quais é importante citar o registrador de controle FLAGS (EFLAGS na arquitetura IA-32 e RFLAGS na arquitetura Intel64) e os registradores de controle CR0, CR2, CR3 e CR4 *(Control Register* 0 até 4. O registrador CR1 possui todos os bits reservados).

O registrador E/RFLAGS possui bits com informações de status da máquina, por exemplo o bit de overflow, que sinaliza que ocorreu um overflow na execução da última instrução. O registrador CR0 contém algumas flags de controle do sistema e do modo de operação e estado do processador. O CR2 possui o endereço linear que causou a última falta de página. No CR3 está o endereço base para a estrutura de tabelas de página. CR4 contém flags que podem habilitar algumas extensões de arquitetura e indicar suporte de algumas funcionalidades.

6.2 MODOS DE OPERAÇÃO

Existem quatro modos de operação disponíveis nos processadores IA-32. A arquitetura Intel64 adicionou mais um modo de operação, que possui dois sub-modos. Os modos de operação são mostrados em um grafo de transições entre modos na figura 22. O modo real é o modo inicial do processador na inicialização do sistema. Ele coloca a máquina em um estado que executa operações como um processador de 16bits. Esse modo é mantido por compatibilidade. O segundo modo é o modo protegido. Esse é o modo padrão para se trabalhar na arquitetura IA-32. Entre as funcionalidades que a mudança para esse modo possibilita, pode-se citar a execução de instruções e registradores de 32 bits e a possibilidade de utilização do mecanismo de paginação com diretório e tabela de páginas.

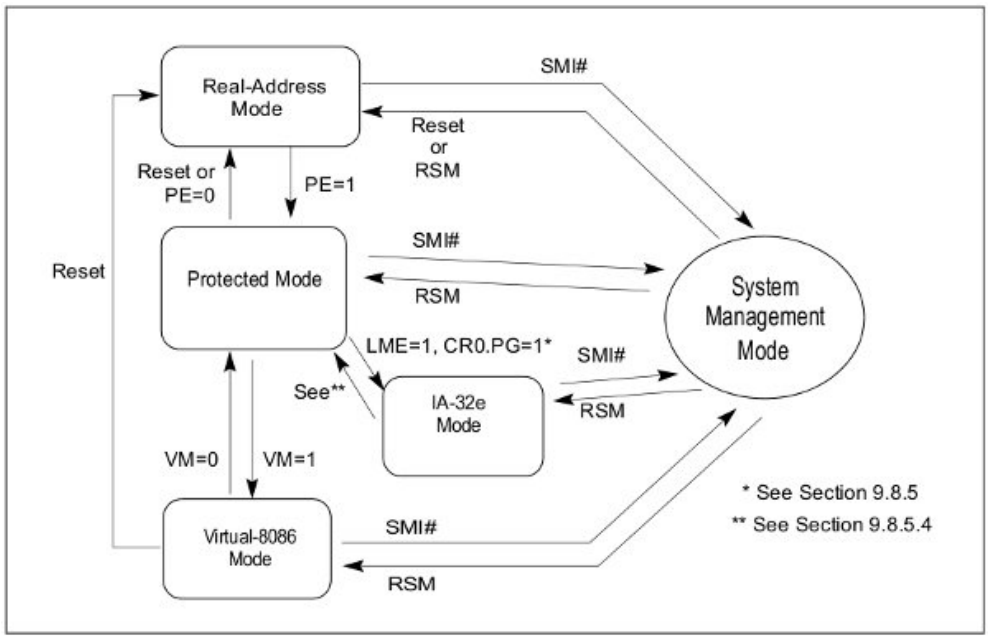


Figura 22: Modos de operação disponíveis na arquitetura Intel64. (Fonte: Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide)

O modo de operação virtual-8086 possibilita o processador que está no modo protegido executar código dos processadores 8086 sem precisar voltar ao modo real. O modo de gerenciamento do sistema (SMM - *System Management Mode*), disponibiliza ao sistema operacional um ambiente alternativo para monitorar e controlar recursos de forma energeticamente melhor. O último modo é o modo IA-32e, que foi adicionado na arquitetura Intel64 e possui dois sub-modos: IA-32e compatibilidade e IA-32e 64bit. Apenas os modos protegido e IA-32e serão abordados mais profundamente a seguir.

Algumas configurações são necessárias antes da mudança de modos de operação. Para entrar no modo protegido, as seguintes estruturas de dados precisam estar presentes na memória: IDT (*Interrupt Descriptor Table*), GDT (*Global Descriptor Table*), TSS (*Task Switch Segment*), LDT (*Local Descriptor Table*, Opcional), pelo menos um diretório e uma tabela de páginas caso queira-se utilizar paginação, um segmento de código a ser executado quando for efetuada a troca de modos, códigos para controle de interrupções e exceções. As estruturas GDT, TSS e LDT e paginação serão melhor abordados em 3.3.4. IDT, interrupções e exceções serão descritas em 3.3.5.

Além disso, os seguintes registradores precisam ser inicializados: GDTR (*Global Descriptor Table Register*), IDTR (*Interrupt Descriptor Table Register*, Opcional), CR1 até CR4, MTRRs (*Memory Type Range Registers*, apenas para Pentium 4, Intel Xeon e família de processadores P6). Tendo todas esses estruturas e registradores prontos, é possível mudar para o modo protegido setando o bit 0 (flag PE, *Protection Enabled*) no registrador CR0.

Para partir para o modo IA-32e, é necessário primeiro estar no modo protegido e com paginação habilitada. A partir desse estado, é exigido que se desabilite o bit de paginação no registrador CR0. Posteriormente, é preciso habilitar o bit PAE (*Physical-Address Extension*) no registrador CR4 e, a seguir, carregar o registrador CR3 com o endereço base físico do mapa de página de nível 4 (PML4). Finalmente, habilita-se o bit LME do registrador IA32_EFER e reabilita-se o bit de paginação no registrador CR0. Duas coisas são importantes nesse momento: A primeira é que a próxima instrução a ser executada esteja em uma posição de memória que esteja configurada na tabela de páginas, para evitar exceção de falta de página, e a segunda é que a mesma instrução esteja em um segmento de código de 32 bits (Portanto, o processador estará no sub-modo IA-32e Compatibilidade).

6.3 PROCESSO DE *BOOT*

Quando ligados, os processadores da Intel fazem uma inicialização de hardware, seguida de um auto teste (chamado BIST - *Built-In Self-Test*). Cada registrador é colocado em um estado inicial conhecido e o processador é posto no modo real, com paginação desabilitada. Todas as caches e buffers são limpos. Um código de inicialização do sistema faz toda a inicialização específica de cada modelo de processador nesse momento. A primeira instrução é executada no endereço físico 0xFFFFFFF0, onde a EPROM contendo um código de inicialização deve estar presente. Geralmente, esse código de inicialização executa um *jump* para uma posição de memória onde está o BIOS.

O BIOS por sua vez faz um processo de inicialização e teste dos componentes de hardware presentes no sistema e tenta ler o primeiro setor dos discos, em busca de um disco bootável. Para que um disco seja bootável, ele deve ter uma assinatura no final do primeiro setor (nos bytes 510 e 511), com o valor 0xAA55. O disco escolhido terá esses 512 bytes copiados para a posição 0x7c00 na memória física e será feito um *jump* para esse local. A partir desse momento, o código de *boot* do sistema operacional será executado.

Como nesse momento o processador está no modo real, o código deve ser composto por instruções de 16 bits. O mais comum é que agora se queira partir para o modo protegido. Para isso, o sistema operacional deve possuir um código que contemple os passos descritos na seção anterior. Enquanto o processador estiver no modo real, é possível utilizar algumas funções auxiliares disponíveis pela BIOS que incluem acesso ao disco e ao vídeo.

6.4 ORGANIZAÇÃO DE MEMÓRIA E PAGINAÇÃO

A arquitetura IA-32 possui duas importantes vertentes sobre organização de memória: segmentação e paginação. Segmentação permite dividir a memória em segmentos separados de código, dados e pilha. Basicamente, convertendo um endereço lógico em um endereço linear. Não existem meios de desabilitar segmentação. Paginação é uma funcionalidade opcional, sendo por padrão desabilitada. Ela permite transformar endereços lineares em endereços físicos, determinando inclusive permissões de acesso e tipo de cache utilizadas para tais

endereços.

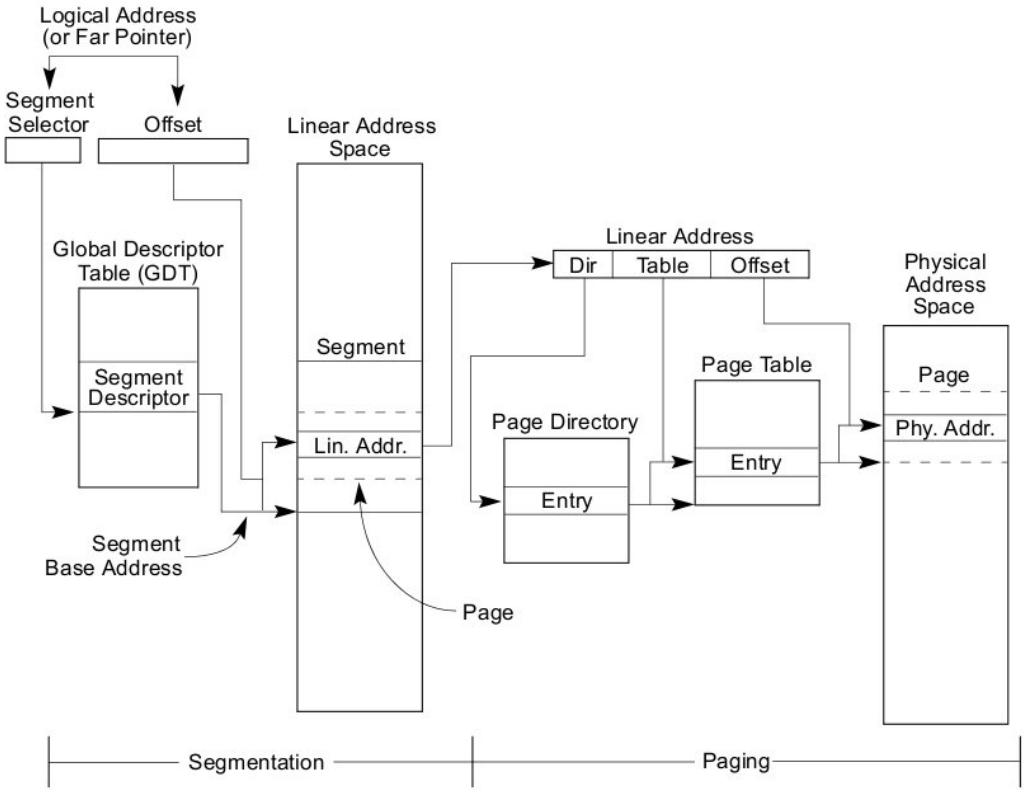


Figura 23: Transformação do endereço lógico em linear e físico. Fonte: Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide

Endereços lógicos consistem em um seletor de segmento e um deslocamento. O seletor do segmento contém um índice de uma tabela onde estão informações daquele segmento. Entre as informações contidas na tabela, existe o endereço base daquele segmento. Esse endereço base é somado ao deslocamento, formando um endereço linear.

Quando se está utilizando paginação, um endereço linear é dividido em várias partes (dependendo do nível de tabelas de página que está sendo usado). Cada uma das partes é convertida em uma entrada em cada um dos níveis das tabelas de páginas. A figura 23 ilustra a conversão de um endereço lógico em um endereço linear e então em um endereço físico utilizando paginação. No exemplo, são

utilizados apenas um diretório de páginas e uma tabela de páginas. Os bits menos significativos do endereço linear são utilizados como deslocamento na página de memória escolhida.

As configurações de segmentação dependem principalmente dos registradores de seleção de segmento, CS (*Code Segment*), SS (*Stack Segment*), DS (*Data Segment*), ES, FS e GS (Registradores para segmentos de propósito geral) e das tabelas de descritores GDT e LDT. Cada um desses registradores é composto por um índice, um bit indicador de tabela (GDT ou LDT) e dois bits de nível de privilégio.

Cada entrada das tabelas GDT e LDT descrevem propriedades do segmento de memória, como o endereço base do segmento, seu tamanho (limite), permissões de leitura e execução, nível de privilégio necessário para acessar o segmento, entre outras. O endereço base da GDT deve ser carregado no registrador GDTR utilizando a instrução *lgdtr.* LDTs são opcionais, os endereços base das LDTs são colocadas na GDT. Um detalhe importante é que as entradas das tabelas quando o processador está no modo IA-32e 64bit consideram a base sempre como zero e o limite é ignorado, criando um endereço linear igual ao lógico, exceto para os registradores de seleção de segmento FS e GS, que funcionam normalmente.

O mecanismo de proteção de segmentos reconhece até 4 níveis de privilégio sendo 3 o menor nível e 0 o maior. O nível de privilégio corrente é mantido nos registradores CS e SS. Ele é comparado com o nível de acesso necessário para um dado segmento de memória, disponível no descritor daquele segmento. No caso de o nível de privilégio ser menor, uma exceção é gerada. Níveis de privilégio são alterados com ajuda de *call gates*. C*all gates* são entradas especiais nas tabelas de

descritores que apontam para uma região de código e, na transição, modificam o nível de privilégio.

No caso do sistema de paginação estiver ativo, endereços lineares são traduzidos para endereços físicos (ou geram exceções, caso a devida página não esteja presente ou não se tenha permissão naquele momento àquela página). Três modos de paginação estão disponíveis: paginação de 32 bits, PAE (*Physical Address Extension*) e paginação IA-32e. A forma de tradução dos modos de paginação é bastante semelhante, modificando-se apenas o número de níveis e o número de bits relacionado a cada nível. Sendo assim, ela será melhor detalhada para a paginação de 32 bits, por ser a mais simples, e explicada de forma mais superficial para as demais, por serem análogas.

A paginação de 32 bits separa o endereço linear em 3 partes, sendo duas de 10 bits nos bits mais significativos e uma de 12 para os bits menos significativos. O primeiro nível é composto pelos 10 primeiros bits, que são um índice para uma tabela que fica na memória principal, chamada diretório de páginas. Existe também um registrador especial (CR3) que aponta para o endereço base de um diretório de páginas. Esse diretório de páginas possui até 1024 entradas. Cada entrada é um ponteiro para o endereço-base de uma tabela de páginas (o segundo nível). Vale lembrar que, para o modo 32 bits, os ponteiros possuem tamanho 4 bytes, ou seja, a tabela possui tamanho 4096 bytes.

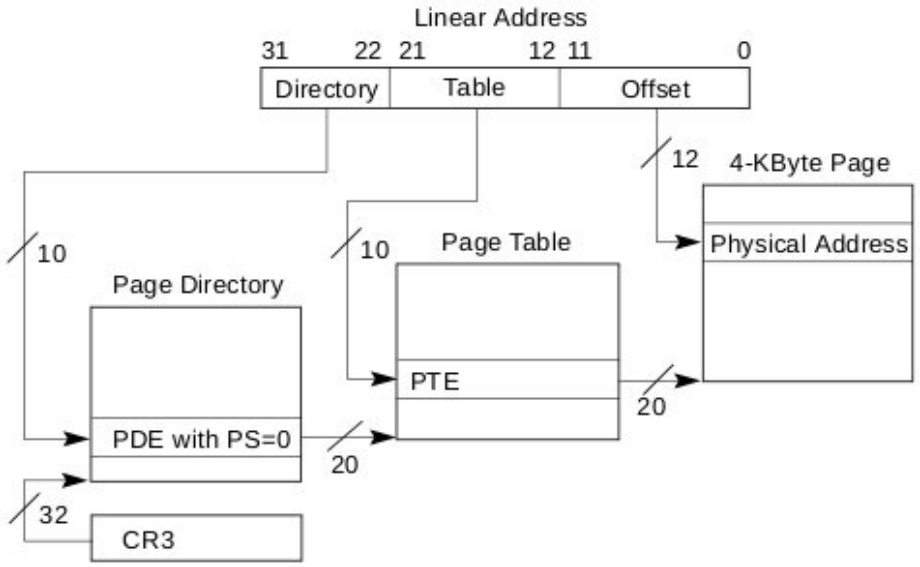


Figura 24: Paginação 32 bits com página de 4KB. Fonte: Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide

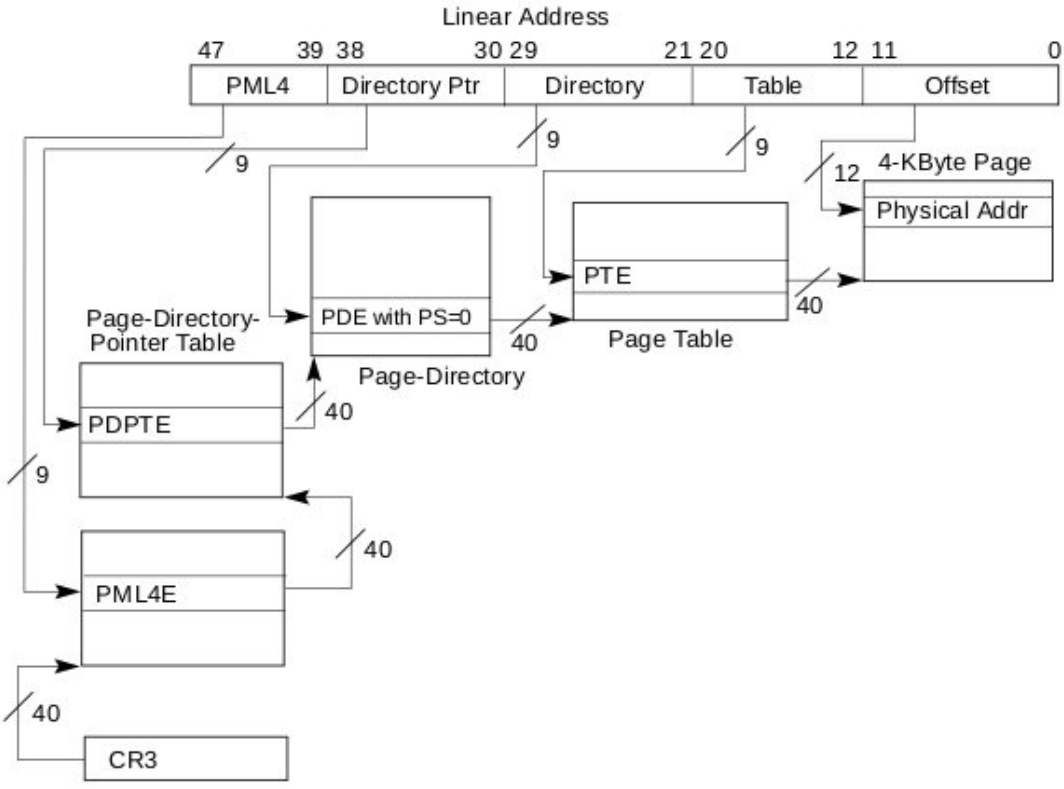


Figura 25: Paginação 64 bits com página de 4KB. Fonte: Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide

Os 10 bits seguintes possuem o índice da tabela de páginas, onde é guardado o endereço base da página. Finalmente o terceiro e último nível possui 12 bits de deslocamento dentro da página, que somado ao endereço base produz o endereço físico na memória principal. A figura 24 detalha essa estrutura.

Existe também a possibilidade de trabalhar com páginas de 4MB. Nesse caso, o endereço linear é separado em apenas dois níveis: 10 bits para a tabela de páginas e 22 bits para as páginas. A utilização de PAE permite estender um sistema de 32 bits para endereçar 52 bits de endereço físico. O endereço linear é separado em 4 partes de 2, 9, 9 e 12 bits, porém, também é possível usar PAE com páginas de 2MB.

A paginação para IA32-e com páginas de 4KB é a com mais níveis na arquitetura. Ela possui 5 níveis para o endereço linear (ou seja, o mesmo é separado em 5 partes). Esse modo só está presente em processadores de 64 bits. Apenas os 48 bits menos significativos são usados do endereço linear. Bits a partir do último devem ser repetidos, até o bit 64, ou o processador gera uma exceção. Endereços nessa forma são chamados de endereços canônicos. A figura 25 exemplifica a tradução de um endereço linear em endereço físico utilizando paginação para IA32-e. É possível também utilizar páginas de tamanho 2MB e 1GB.

6.5 INTERRUPÇÕES E EXCEÇÕES

O fluxo de execução básico de um processo é a realização de instruções de maneira sequencial pelo processador. Uma das formas de fluxo alternativo é a

produção de uma interrupção ou uma exceção. Interrupções podem ser externas ou internas. As interrupções externas acontecem quando algum sinal é emitido ao processador em anúncio de um evento (ou seja, são geradas por hardware). Já as internas são geradas pelo programador através da instrução *int* (ou seja, são geradas por software). Na verdade, qualquer exceção pode ser simulada por software com essa instrução, porém, algumas delas podem precisar de um cuidado especial (como a preparação da pilha de chamada de função) antes de serem utilizadas.

Exceções acontecem quando o processador se depara com algum erro do qual ele não sabe como proceder. Um exemplo clássico é a tentativa de divisão por zero, que gera a exceção #DE (*Divide Error Exception*). As exceções são divididas em três grupos: Faltas (Faults), armadilhas (Traps) e abortos (Aborts). Algumas exceções podem pertencer a mais de um grupo, dependendo da situação em que elas acontecem.

Faltas são exceções que podem ser corrigidas. Se for o caso, é possível retornar à mesma instrução para tentar executa-la novamente. Um exemplo é a exceção por falta de página (*#PF – Page Fault*), que acontece quando a paginação está ativada e é feita uma tentativa de acesso à uma posição de memória não válida para aquele momento (pode ocorrer por falta de privilégio, página não mapeada ou outros motivos).

Armadilhas podem acontecer depois da execução de uma instrução. Diferente das faltas, o retorno de uma armadilha é a instrução posterior à atual (ou seja, o fluxo de execução continua normalmente depois que a exceção é tratada). A exceção de overflow (*#OF – Overflow Exception*) é um exemplo de armadilha, onde

ao se detectar que o bit de overflow foi setado, a exceção pode ser gerada.

Finalmente, abortos são exceções que não possibilitam um retorno ao fluxo normal de execução do código original. Eles desencadeiam quando um erro grave ocorre, como por exemplo, uma falta dupla (*#DF – Double Fault Exception*), ou seja, uma exceção acontece na execução do processo de recuperação de outra exceção. Quando uma nova exceção ocorre, uma exceção ao tentar executar a recuperação de uma falta dupla, o processador gera uma falta tripla (*Triple fault*) e desliga.

Quando uma interrupção ou exceção ocorre, o processador precisa de uma forma de saber como proceder. A arquitetura disponibiliza uma estrutura de dados, chamada IDT (*Interrupt Descriptor Table*), que pode ser populada com entradas de 8 bytes (no modo 32bits protegido) ou 16 bytes (no modo 64bits). Cada uma das entradas da IDT possui a configuração exata que diz como o processador deve reagir, incluindo o endereço lógico do início do código de recuperação. As interrupções que não possuem tratamento devem conter o bit de presença (*present flag*) com o valor 0. Um registrador adicional, chamado IDTR, deve ser carregado com o endereço base e o tamanho da IDT. Como cada interrupção possui um identificador único, é possível saber a posição do descritor na tabela multiplicando esse identificador pelo tamanho de uma entrada da tabela. Por exemplo, se for necessário inserir uma entrada para a exceção de overflow (*#OF – Overflow Exception, Interrupt 4*), sendo que a IDT está na posição 0x1000 e se esteja em um ambiente de 64bits, tal descritor deve estar no endereço 0x1000 + 3 * 16 bytes = 0x1030 (lembrando que a primeira interrupção reside no índice 0). Existem no máximo 256 interrupções (portanto o tamanho máximo da IDT será de 4096 bytes, se utilizarmos o mesmo exemplo).

7 EPOS64 - UMA VERSÃO DO EPOS COM SUPORTE À ARQUITETURA INTEL64

O EPOS possui implementação para diversas arquiteturas entre 8 até 32 bits. A proposta desse trabalho é adicionar o suporte à arquitetura Intel 64, também chamada de x86 64, mantendo a transparência arquitetural. Esse capítulo explica que modificações foram necessárias no sistema para estender o suporte do EPOS para a nova arquitetura.

A seção 7.1 apresenta as mudanças nos mediadores de hardware e nas abstrações, dando ênfase em o que foi feito para manter a transparência arquitetural do sistema utilizando esses conceitos. A seção 7.2 comenta brevemente mudanças no processo de compilação, seguida pela seção 7.3, onde o processo de inicialização do EPOS para a arquitetura Intel64 é comparado com a inicialização da arquitetura IA-32. A seção 7.4 mostra as modificações nas classes referentes à gerencia de memória, mostrando desde o projeto até as diferenças de implementação. A seção 7.5 apresenta diferenças no tratamento de interrupções. Finalmente, a seção 7.6 descreve os testes feitos, na tentativa de validar a nova implementação.

7.1 TRANSPARÊNCIA ARQUITETURAL

A preocupação com a transparência arquitetural foi constante no desenvolvimento do projeto. Para possibilita-la, alguns pontos foram importantes. Os

mediadores de hardware no EPOS funcionam como a camada de abstração do mesmo, possibilitando uma interface única que pode ser usada pelas abstrações, como discutido nas seções 5.1 e 5.2. De forma utópica, nenhuma mudança é necessária nas abstrações, com a única exceção de quando alguma delas tem conhecimento de alguma informação dependente de arquitetura. Nesse último caso, deve-se levar em consideração a possibilidade de mover tal informação para um mediador de hardware. Por fim, o uso de meta-programação estática (seção 3.2) é essencial para a configuração das abstrações e mediadores, de forma a possibilitar otimizações em tempo de compilação.

Com esses pontos em mente, a estratégia para manter a transparência arquitetural no EPOS foi a extensão dos mediadores de hardware existentes para a arquitetura IA-32. Sobre o mediador da CPU, a interface *CPU_Common* (Seção 5.1) foi tomada como base para a criação da nova classe abstrata *CPU_Intel*. Dessa, duas classes concretas ficaram disponíveis: A classe *IA32* e a classe *X86_64*. A nova estrutura está disponível abaixo em formato UML na figura 26.

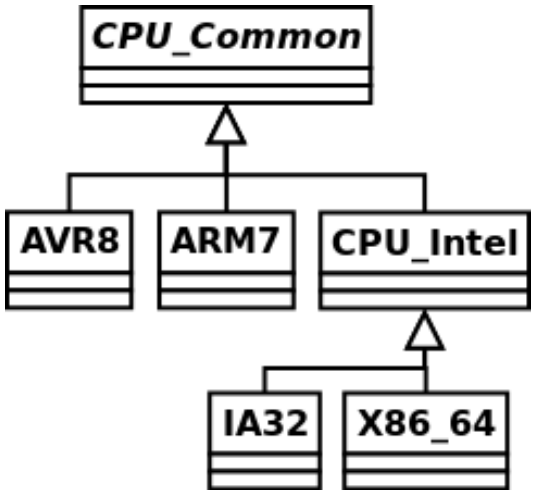


Figura 26: Mediadores de hardware da CPU

O desafio para criar o novo mediador envolveu mudanças nas estruturas de dados (GDT e IDT), o aumento do tamanho de ponteiros, que foi resolvido de forma simples e elegante por meio da definição de tipos primitivos escolhidos em tempo de compilação utilizando meta-programação estática, a reimplementação de métodos que estavam presentes no mediador IA32 mas que, por mudanças da arquitetura, não era mais válidos, e a implementação de novos métodos, por exemplo, a possibilidade de troca de modo de operação de forma transparente. Esse último é necessário no processo de inicialização do sistema (*PC_Setup*).

Também houveram mudanças no mediador da MMU, em virtude do aumento dos níveis de paginação na nova arquitetura. Como o mediador da MMU possui alguns pontos mais profundos, suas mudanças serão discutidas apenas na seção 7.4.

Foi possível a utilização das abstrações do EPOS sem a necessidade de mudanças diretamente nelas (Mudanças dependentes de arquitetura puderam ser movidas para os mediadores). Sendo assim, como será mostrado na seção 7.6, as aplicações continuam utilizando essas abstrações sem problemas de compatibilidade. Assim, como foi explicado no início dessa seção, a transparência arquitetural se deu pela inserção e modificação das famílias de mediadores, com auxilio de técnicas de meta-programação estática, possibilitando a compatibilidade das diversas abstrações para as aplicações.

7.2 PROCESSO DE COMPILAÇÃO

Por causa da mudança de arquitetura, foi necessário utilizar um novo compilador para o EPOS, com arquitetura-alvo a x86 64. Foi modificado o arquivo makedefs (Seção 5.3), adicionando as constantes para a nova arquitetura, e incluindo o compilador. Em virtude da adição de variáveis de pré compilação no *boot* do sistema, foi necessária uma pequena atualização no makefile do mesmo, onde é passada uma opção adicional para definir para qual arquitetura o *boot* está sendo gerado. Nada foi modificado na ferramenta *eposcc*. A ferramenta *eposmkbi* foi modificada de forma a considerar a existência da nova arquitetura, porém, nenhuma mudança significativa foi feita nessa ferramenta.

7.3 PROCESSO DE INICIALIZAÇÃO

O processo de inicialização foi uma das partes que mais necessitou de atenção pela alta dependência arquitetural. Para tentar manter a maior estrutura possível existente (Seção 5.4), o novo processo foi construído em cima do antigo, modificando-se apenas partes críticas. Em primeiro lugar, foram adicionadas variáveis de pré-compilação (macros do preprocessador C) no arquivo de *boot*, de forma que dependendo da arquitetura alvo, esse é o momento que se muda para o modo protegido ou o modo Intel64 (Também chamado modo *long*. Veja a seção 6.2). Como o modo *long* necessita de paginação ativa, é criada uma pequena estrutura de

páginas temporária, apenas para possibilitar o processo de *boot* e *setup* do sistema.

Ainda sobre o *boot*, houve a necessidade da adição do suporte a *boot* por drives diferentes do drive de disquete *(*Por exemplo*, pendrive* e HD – *Hard Disk*). Para a arquitetura IA32, existiam duas formas de *boot*: Por disquete e por rede. O *boot* por outros dispositivos foi interessante para a execução dos testes com máquina real.

Como mostra o diagrama da figura 27, o PC_Setup também foi alterado (Compare com o diagrama na seção 7.3), com a necessidade de adição de uma estrutura de decisão meta-programada para auxiliar no processo. Dessa forma, a paginação não é novamente ativada. Quando no modo *long*, ela apenas é recarregada com a estrutura de páginas correta do sistema, esquecendo totalmente a anterior. Outro detalhe importante é que o modo de operação *long* não possibilita a execução das instruções *lgdtr* e *lidtr*, necessárias para o carregamento da GDT e da IDT, respectivamente. Por isso, foi adicionado um código extra para entrar em modo compatibilidade antes de executar tais instruções, sendo, por isso, preciso adicionar partes de código de 32bits.

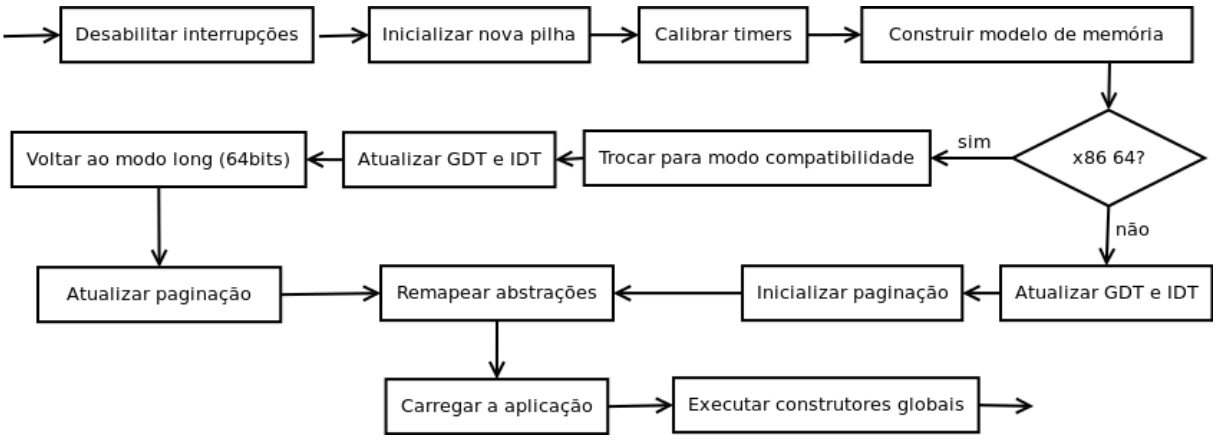


Figura 27: Novo fluxograma do PC_Setup

Os construtores globais funcionam, de forma simplificada, por meio de uma lista criada pelo compilador e, para que executem, códigos de inicialização de construtores são necessários. Esses códigos, chamados de *crt*'s, existem na linguagem C++ para a execução de construtores e destrutores globais. Os mesmos são escritos totalmente em *assembly*, e não puderam ser totalmente reaproveitados da arquitetura IA32. Houve então uma cópia de tais arquivos, que foram modificados de acordo com a necessidade para a nova arquitetura. Mudanças se deram principalmente pelo tamanho dos ponteiros nas estruturas de dados e pela nova convenção de passagem de parâmetros presente nos compiladores de 64 bits.

O *System V ABI* (*System V Application Binary Interface*) é um documento que padroniza convenções para softwares, o que auxilia na interoperabilidade daqueles que a seguem. O compilador GCC segue tais convenções, e uma delas define que, para sistemas de X86 64, a passagem de parâmetros deve ser feita parcialmente por registradores, ao invés de totalmente pela pilha, como era nos sistemas IA32. Esse foi o motivo da mudança comentada no parágrafo anterior e de algumas outras, como no mediador da CPU, para ser utilizado na criação de um contexto, e no controlador de interrupções, que será melhor discutido na seção 7.5.

7.4 GERENCIAMENTO DE MEMÓRIA

Existem dois pontos a serem discutidos sobre as alterações na gerência de memória do EPOS para a nova arquitetura: Mudança no mediador da *MMU* e o

esperado *overhead* de consumo de memória por conta da tabela de páginas. Como comentado na seção 6.4, a arquitetura Intel de 64 bits utiliza paginação em 5 níveis, quando utilizada com páginas de 4KB. Para evitar maiores mudanças, foi escolhido que se mantivesse esse tamanho de páginas para a nova arquitetura.

Por isso, houve um aumento no número de parâmetros do *template* da *MMU*, que eram 3 (veja seção 5.5) e passou a ser 5, sendo os dois últimos o número de bits para os ponteiros de diretório e o número de bits para o *PML4*. A escolha dos valores para os parâmetros do *template* da *MMU* é feita com auxilio da definição de um tipo-base, que se modifica automaticamente de acordo com a arquitetura. A *MMU* utilizada no sistema é uma subclasse da MMU_Common (veja o diagrama da imagem 28 e 29).

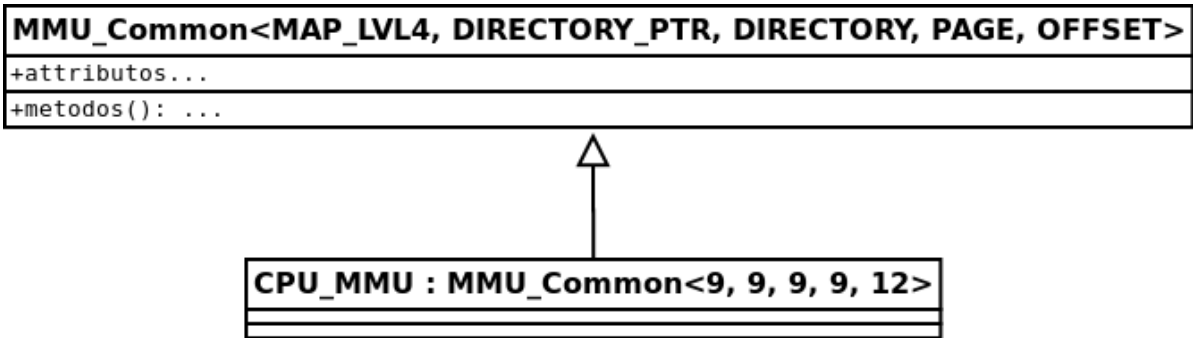


Figura 28: Herança da classe MMU quando compilada para a arquitetura X86 64

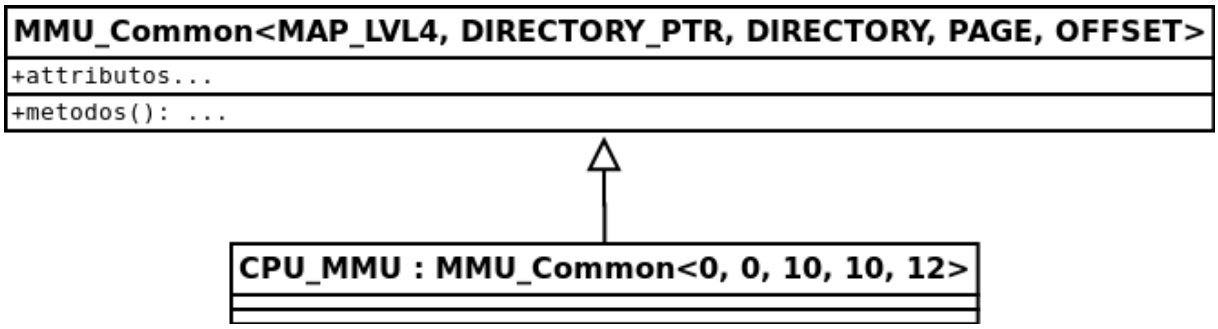


Figura 29: Herança da classe MMU quando compilada para a arquitetura IA32

A fim de manter compatibilidade com as versões anteriores, é possível simplesmente setar esses parâmetros para 0, de forma a ignorá-los. Essa modificação possibilita que as mesmas configurações existentes no EPOS sejam aproveitadas, fixando os dois últimos níveis de paginação e fazendo com que que apenas os primeiros 4GB de memória virtual fiquem endereçáveis no EPOS.

A partir das modificações descritas anteriormente, duas abordagens são possíveis para a utilização de paginação de memória no EPOS. A primeira abordagem é a utilização do mesmo modelo de memória presente na arquitetura IA32, o que possibilita um *overhead* pequeno de tabelas de página na memória, pois são necessárias apenas mais duas páginas no sistema, sendo uma para manter a tabela de ponteiro para diretórios (utilizada pelo *PML4*) e a outra para manter a tabela de diretórios (utilizada pelos ponteiros de diretório), totalizando 8KB. Também por isso, não é necessário adicionar novos métodos na *MMU*, mantendo-se exatamente a mesma estrutura. Um ponto fraco dessa decisão é que não é possível utilizar memória acima de 4GB com esse modelo, porém, para algumas aplicações isso pode não ser importante.

A segunda abordagem envolve a utilização de toda a memória física presente, ou seja, no máximo 64GB de memória física, para a maioria dos processadores (Veja a seção 6.4). Essa abordagem possui a desvantagem no *overhead* das tabelas de página, que aumentam para aproximadamente 128MB o espaço necessário para acomodar toda a estrutura de paginação. O mediador da MMU é equivalente para ambas as propostas, sendo apenas a configuração da mesma alterada (A segunda proposta necessita o cadastro de mais páginas).

7.5 INTERRUPÇÕES

Levando em consideração as mudanças no mediador da CPU (seção 5.2), não houve necessidade de mudanças na configuração da IDT, pois tal suporte foi englobado pelo mediador.

Modificações mínimas foram feitas no *int_dispach* (seção 5.6), em virtude da mudança do conjunto de instruções disponíveis nas arquiteturas. De forma mais objetiva, o problema era a instrução *pushal*, indisponível na nova arquitetura. Essa instrução coloca todos os registradores correntes na pilha, para evitar que, durante a execução do tratamento da exceção, os registradores da aplicação sejam modificados. A nova implementação necessitou simular a mesma funcionalidade fazendo *push* individual para cada registrador.

Além disso, por causa do problema da passagem de argumentos citada na seção 7.3, houve uma modificação na passagem de argumentos para a função de interrupção, que agora são colocados nos registradores antes que a chamada seja feita. Toda a implementação das funções de interrupção foram mantidas, sendo que as alterações citadas foram as mais notáveis na questão do controle de interrupções.

7.6 AMBIENTES DE TESTE

O EPOS possui uma base de testes que são executados quando uma nova funcionalidade é adicionada no sistema. Eles testam o comportamento das *threads*,

a gerência de memória, os semáforos, os alarmes, entre outras abstrações e mediadores de hardware. Tais testes foram reproduzidos em máquinas virtuais e uma máquina real, com o objetivo de validar a implementação. Os testes passaram com sucesso em todos os ambientes. Existem basicamente dois tipos de máquinas virtuais: os emuladores e os virtualizadores.

Emuladores simulam o ambiente real de uma determinada arquitetura em software. Sua principal desvantagem é a eficiência reduzida em virtude dessa simulação de hardware. Foram utilizados dois emuladores para os testes. O primeiro foi o qemu, um emulador amplamente utilizado pelos pesquisadores do LISHA. O qemu tem a vantagem de emular outras arquiteturas, como ARM, Power PC, e outras. O segundo emulador escolhido foi o bochs. Esse é um emulador específico para processadores IA32 e x86 64 da Intel. O critério para a escolha dos emuladores foi simplesmente por sugestão de outros pesquisadores, não havendo uma vantagem conhecida sobre qualquer outros emuladores. Tanto o qemu quanto o bochs possuem várias configurações, por exemplo, o tamanho de memória, adicionar ou remover funcionalidades da CPU e adição de drives de disco.

Virtualizadores possibilitam à máquina virtual a utilização dos recursos do processador real. Existe a necessidade do processador para esse suporte. A Intel chama essa tecnologia de VT-X. O virtualizador utilizado para os testes foi o VirtualBox. Novamente, não há vantagem conhecida dessa máquina virtual em comparação com qualquer outra. O uso da tecnologia VT-X tem a vantagem que a simulação tem uma eficiência maior em comparação aos emuladores. Além disso, como os componentes de hardware são reais, essa simulação fica mais próxima a uma máquina real. Por exemplo, para testes envolvendo interrupções de *timer*, não

é necessário simular o *timer* em software.

Máquinas reais são computadores reais utilizados no dia a dia. Apenas uma máquina real foi utilizada para os testes. Ela possuía um processador Intel Pentium E5400 (2.7GHz) Dual-Core (64bit). Como a máquina não possuía drive de disquete, o *boot* do EPOS foi incrementado com o suporte a *boot* por HD e pendrive, possibilitando a execução dos testes por meio de um pendrive carregado com a imagem do EPOS. Não houveram diferenças perceptíveis em comparação com os outros dois ambientes, sendo que os testes executaram normalmente.

A bateria de testes do EPOS possui testes para várias abstrações do sistema. A execução de um teste que retorne os resultados esperados como saída é um indício de que a implementação do projeto foi um sucesso. Alguns dos testes serão comentados aqui para ilustrar o que foi testado.

O *ostream_test* testa se a saída para o terminal está funcionando corretamente. Existem duas formas de rodar esse teste em uma máquina do tipo PC (arquiteturas IA-32 e X86 64): A utilização de um display serial, e o acesso direto ao vídeo por meio de escritas em memória. Esse teste é importante inclusive para a execução dos outros testes, pois necessitam de alguma forma de saída para os resultados.

Existem alguns testes de funções específicas e de estruturas de dados no EPOS. O próprio *ostream_test* é um caso desse, que simplesmente testa uma função (nesse caso, de escrita na saída padrão). *handler_test*, *malloc_test*, *list_test* e *vector_test* são outros exemplos. Esses testes interagem com alocação e desalocação de memória, sendo úteis para testar códigos que necessitam do mediador da MMU.

Os testes *thread_test*, *semaphore_test*, *mutex_test*, *alarm_test* e *clock_test* são exemplos de testes para as abstrações. Sua execução correta é um indício de que a implementação do mediador da CPU está correta (por exemplo, ele é usado nas abstrações de t*hread* e *semaphore*). Basicamente, cada um desses testes cria uma instância de cada abstração e tenta fazer chamadas aos métodos disponíveis em cada uma delas. Por exemplo, o *thread_test* instância duas *threads* que executam funções diferentes paralelamente. A criação de uma *thread* testa a criação de um novo contexto, o que é feito pelo mediador da CPU. A execução paralela eventualmente necessita de troca de contexto, outro método do mediador da CPU.

A fim de obter uma comparação com a implementação do EPOS para a arquitetura IA-32, foram extraídos alguns testes e comparados quanto ao seu tamanho de código e de dados. A figura <> mostra as diferenças.

	IA32			X86_64		
Teste	.text	.data	.bss	.text	.data	.bss
Alarm	18716	12	464	18373	16	880
Semaphore	29215	12	656	28164	16	1232
Mutex	28399	12	624	27124	16	1168
Thread	21903	12	464	21700	16	912

* Todos os valores estão em bytes.

Figura 30: Comparação dos tamanhos dos segmentos *text*, *data* e *bss* nas arquiteturas IA-32 e X86 64

A comparação é separada em três informações: O tamanho do segmento *text* (tamanho do código executável), o tamanho do segmento *data* (tamanho dos dados inicializados) e o tamanho do segmento *bss* (tamanho dos dados não inicializados). Percebe-se um aumento principalmente dos segmentos *data* e *bss*. Isso acontece pelo fato de que esses segmentos incluem ponteiros para locais de memória. Os

ponteiros têm tamanhos diferentes em cada arquitetura, sendo de 4 Bytes na IA-32 e 8 Bytes na X86 64. O valor não é exatamente o dobro pois alguns dos dados não são ponteiros. Por exemplo, o tipo primitivo *char* continua tendo o mesmo tamanho em ambas as arquiteturas.

A sutil diminuição do tamanho do segmento *text* se dá provavelmente pela nova convenção de chamada de métodos utilizado pelo compilador. Como comentado na seção 7.1, alguns parâmetros são passados por registrador, e não por pilha. Isso possibilita que instruções do tipo *push* e *pop* para a chamada de função sejam desnecessárias, diminuindo levemente o tamanho do código.

8 CONCLUSÃO

O EPOS é um sistema operacional desenvolvido no LISHA, onde conta com a contribuição de vários estudantes de computação e engenharia. Antes do presente trabalho, existiam versões do EPOS para diversas arquiteturas entre 8 até 32 bits, incluindo AVR, ARM, IA32, Power PC, entre outras. O sistema operacional sempre primou pela transparência de arquitetura, possibilitando que as abstrações de software fossem reutilizadas.

O presente trabalho estendeu o suporte do EPOS para uma arquitetura de 64 bits, conhecida como x86 64, da Intel, mantendo o suporte a compilar o S.O. para todas as outras arquiteturas. Isso significa que o EPOS atingiu transparência arquitetural dos 8 aos 64 bits. O conjunto de testes ajudou a validar a

implementação, dando mais confiabilidade ao trabalho.

Como resultados adicionais, foram utilizados dois compiladores para trabalhar com a nova arquitetura, sendo um deles um *cross compiler* de computadores IA32 para X86 64 e outro um compilador X86 64. Um resultado não previsto foi o suporte a *boot* por HD e Pen Drive, uma necessidade para executar os testes em máquinas reais.

Como trabalhos futuros, é possível sugerir um suporte a MSRs (*Model Specific Registers*) ao mediador de CPU das arquiteturas da Intel. Os MSRs contém informações adicionais sobre as capacidades de cada processador, e não foram utilizados no desenvolvimento desse trabalho. Isso possibilitaria configurações e validações em tempo de execução sobre diversos aspectos específicos de cada processador. Uma segunda sugestão é programar também o modo *multicore*, atualmente presente apenas na versão IA32 do EPOS, e, apesar de estar disponível em vários processadores na arquitetura Intel, é por padrão desabilitado e não foi abordado nesse trabalho. O desenvolvimento do modo *multicore* depende principalmente de modificações no processo de *boot* e inicialização do EPOS.

9 REFERÊNCIAS BIBLIOGRAFICAS

Beuche, D. Fröhlich, A. A. **On architecture transparency in operating systems.** Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system, 2000. p.147-152

Mooney, J. D. **Strategies for supporting application portability.** Computer, v.23, n.11, p.59-70, Nov. 1990

Intel Corp. **Intel® 64 and IA-32 Architectures Software Developer's Manual.** Ago. 2012

Czarnecki, K Eisenecker, U. W. **Generative Programming.** 2000

Booch, G. **Object-Oriented Analysis and Design with Applications.** AddisonWesley, 2 edition

Ludwich, M. K. **Método para Abstração de Componentes de Hardware para Sistemas Embarcados.** Florianópolis: Federal University of Santa Catarina, 2012

Fröhlich, A. A.; Schröder-Preikschat W. **Scenario Adapters: Efficiently Adapting Components.** Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics, Orlando, USA, 2000

Anderson, T. E. **The case for application-specific operating systems.** Workstation Operating Systems, 1992.

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

Stroustrup, B. **The C++ Programming Language.** 3 ed. [S.l.]: Addison-Wesley, 2000.

Polpeta, F. V.; Fröhlich A. A. M. **Hardware Mediators: a Portability Artifact for Component-Based Systems.** Proceedings of the International Conference on Embedded and Ubiquitous Computing, volume 3207 of Lecture Notes in Computer Science, pages 271-280, Aizu, Japan, 2004. Springer.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.** Addison-Wesley, 2005.

DIJKSTRA, E. W. **A Discipline of Programming.** Prentice-Hall, 1976.

HOELLER, A. S. **Famílias de Abstrações de Gerência de Memória para o EPOS .** UFSC, 2004.

GRACIOLI, G.; FRÖLICH, A. A. **An Experimental Evaluation of the Impact of Cache Partitioning on Global, Partitioned, and Clustered Real-Time**

Schedulers . ACM Trans. Embedd. Comput. Syst., 2013