CENTRO UNIVERSITÁRIO SERRA DOS ÓRGÃOS
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DESENVOLVIMENTO DO SISTEMA OPERACIONAL FESO

Autor:
Hermano Lourenço Souza Lustosa

Orientador:
Frederico Luís Cabral

Teresópolis
Junho 2013

Autor (Lustosa, H. L. S.). **Desenvolvimento do Sistema Operacional FESO**. Teresópolis: Centro Universitário Serra dos Órgãos, 2013.

Orientador: Frederico Luis Cabral, MSc

Monografia – CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO UNIVERSITÁRIO SERRA DOS ÓRGÃOS.

1. Sistema Operacional 2.Microkernel 3.Multiprocessamento

Trabalho de conclusão de curso apresentado ao Centro Universitário Serra dos Órgãos - Curso de Bacharelado em Ciência da Computação - como um dos requisitos para obtenção do título de Bacharel em Ciência da Computação.

ELABORADO POR HERMANO LOURENÇO SOUZA LUSTOSA E APROVADO POR TODOS OS MEMBROS DA BANCA EXAMINADORA. FOI ACEITO PELO CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO.

TERESÓPOLIS, 29 DE JUNHO DE 2013

BANCA EXAMINADORA:

_____
Frederico Luis Cabral (ORIENTADOR), MSc

_____
Chessman Kennedy Faria Corrêa, MSc

_____
Rafael Gomes Monteiro, MSc

Teresópolis
Junho de 2013

DEDICATÓRIA

Dedico este trabalho aos meus familiares, sem os quais nada seria possível.

AGRADECIMENTOS

Agradeço primeiramente a Deus pela conclusão de mais esta etapa. Agradeço aos meus familiares, meus pais, e minhas irmãs, que sempre me apoiaram muito, e me deram a estrutura necessária para concluir este projeto. Aos meus professores do curso de ciência da computação do UNIFESO. Ao meu professor, orientador e amigo, professor Frederico Cabral, quem primeiramente propôs o projeto. Aos meus colegas do curso de ciência da computação da UNIFESO, que estiveram comigo durante todo o curso, e fizeram com que tudo fosse mais divertido.

RESUMO

Este trabalho descreve o desenvolvimento de um sistema operacional para fins educativos. Sistemas operacionais são um componente chave em um sistema computacional. Por isso, é importante para estudantes de computação obter um entendimento mais profundo sobre o funcionamento desse tipo de sistema. O sistema operacional FESO foi desenvolvido no intuito de fornecer aos estudantes um SO simplificado e funcional, que possa ser usado como ferramenta de aprendizado. Neste trabalho, são descritos algumas ferramentas, técnicas e algoritmos empregados durante o processo de desenvolvimento. Além disso, são demostrados os resultados obtidos nos testes de desempenho de aplicações paralelas baseadas no sistema.

Palavras chave: Sistema Operacional, Microkernel, Multiprocessamento.

ABSTRACT

This work describes the development of an operating system for educational purposes. Operating systems are a key component in a computing system. Thus, it is very important for computer science students to acquire a deeper understanding about the internal working of this kind of system. The FESO operating system was developed in order to provide students with a simplified and functional OS, which could be used as a learning tool. In this work, a few tools, algorithms and techniques employed in the development process are described. Besides, this work presents results obtained on performance tests by parallel applications based on the system.

Keywords: Operating System, Microkernel, Multiprocessing

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ATA(PI)	*Advanced Technology Attachment (Packet Interface)*
ACPI	*Advanced Configuration and Power Inteface*
APIC	*Advanced Programmable Interrupt Controller*
CD	*Compact Disk*
CPU	*Central Processing Unit*
ELF	*Executable and Linking Format*
E/S	Entrada e Saída
GCC	GNU *Compiler Collection*
GDT	*Global Descriptors Table*
GRUB	*Grand Unified Bootloader*
IDE	*Integrated Drive Eletronics*
IDT	*Interrupt Descriptors Table*
ISO	*International Organization for Standardization*
MMU	*Memory Management Unit*
NASM	*Netwide Assembler*
PIC	*Programmable Interrupt Controller*
PIT	*Programmable Interval Timer*
RAM	*Random Access Memory*
SO	Sistema Operacional
TLB	*Table Lookaside Buffer*
VFS	*Virtual File System*
VGA	*Video Graphics Array*

1 INTRODUÇÃO

O entendimento de como um sistema operacional funciona é importante para se compreender como um computador opera. Isto porque é o sistema operacional que gerencia os recursos da máquina e fornece as abstrações necessárias para o funcionamento de diversos aplicativos.

Além disso, as técnicas empregadas na construção dos sistemas operacionais, como a gerência de recursos, o uso de abstrações e a programação em baixo nível (mais próxima ao *hardware*) podem ser aplicadas no desenvolvimento de outros tipos de sistemas. Conceitos importantes, como os referentes à comunicação e sincronização de processos podem ser aplicados na criação de programas que utilizam computação paralela.

Através do estudo de sistemas operacionais é possível obter um entendimento de como os aplicativos são carregados na memória e postos em execução. Também é possível compreender como diversos aplicativos executam em aparente simultaneidade, enquanto apenas um processador está disponível, e como ocorrem os acessos desses mesmos aplicativos aos recursos computacionais (memória, dispositivos de entrada e saída, impressoras, monitores, mouses, teclados, etc...).

Contudo, obter esse entendimento sobre sistemas operacionais não é uma tarefa fácil. Pois graças a sua complexa e importante função de gerenciar um sistema computacional, sistemas operacionais bem-sucedidos acabam por ter milhões de linhas de código. Desta forma, compreender exatamente como um determinado sistema cumpre suas funções através do estudo de seu código fonte é uma tarefa árdua, com certeza cansativa e em alguns casos, até mesmo impraticável, já que nem sempre é possível ter acesso aos arquivos fonte do sistema.

Este trabalho foi desenvolvido para amenizar essas dificuldades, e oferecer aos interessados no estudo de sistemas operacionais um SO que sirva como uma ferramenta de

aprendizado. O objetivo desse trabalho é dar origem a um sistema operacional simplificado, que forneça uma base para o estudo, e permita uma melhor compreensão sobre o tema, sem deixar de lado a funcionalidade.

Mesmo sendo uma ferramenta de aprendizado, o SO deve ser funcional o suficiente para permitir que outros *softwares* devidamente construídos funcionem a partir ele. O sistema também deve ser expansível, e permitir que novos componentes sejam adicionados e/ou modificados caso necessário. Futuramente, o SO desenvolvido nesse projeto poderá servir de base para diversos novos projetos, permitindo que outros estudantes possam aprender e então criar suas próprias soluções para os problemas referentes ao desenvolvimento deste tipo de sistema.

Este trabalho está organizado em 6 capítulos. O capítulo 2 trata da fundamentação teórica, contendo conceitos sobre o *hardware* de computadores e os componentes comuns a diversos sistemas operacionais. O capítulo 3 descreve os detalhes de funcionamento do núcleo do sistema operacional. No capítulo 4 são apresentados os *drivers* de dispositivos e sistemas de arquivos desenvolvidos junto com o sistema. O capítulo 5 trata do desenvolvimento de aplicativos de usuário. Finalmente, no capítulo 6 são apresentadas as conclusões e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Sistemas Operacionais

Sistemas operacionais podem ser entendidos como um conjunto de rotinas executadas pelo processador de forma semelhante a programas de usuário (MACHADO & MAIA, 2007, p. 3). De forma prática, essas rotinas são os blocos que compõem o sistema operacional. Elas são executadas para responder a eventos que ocorrem em um sistema computacional. Entende-se aqui que um sistema computacional consiste em processadores, memória, discos, impressoras, teclados, *mouse*, monitor e outros dispositivos de entrada e saída (TANEMBAUM, 2010).

Segundo SILBERSCHATZ *et al.* (2000) é mais fácil definir um sistema operacional pelo que ele faz, do que pelo que ele é. Dessa forma, na literatura, as definições de sistema operacional geralmente mencionam duas funções básicas. São elas: gerenciador de recursos e máquina estendida.

Um sistema operacional tem a função de gerenciar os recursos do *hardware*, isto é, ele deve garantir a utilização adequada da memória, do processador, dos dispositivos de entrada e saída, entre outros elementos pertencentes ao sistema computacional. No caso desses recursos estarem sendo compartilhados entre diversos programas e usuários, faz-se necessária a ação do sistema operacional para arbitrar o uso dos mesmos. Se esta intervenção não ocorresse, é possível que programas não tão bem intencionados, ou simplesmente mal desenvolvidos pudessem, por exemplo, acessar uma posição na memória reservada a outro programa. Isto pode acarretar em mal funcionamento no programa cuja memória foi "invadida". Tomando como exemplo o caso de dois programas que de forma concorrente tentassem imprimir algum texto, enviando informações diretamente para a impressora quase simultaneamente. Ao término do processo, a impressora pode produzir um texto composto por

linhas enviadas pelos dois programas de forma intercalada, ao invés de duas páginas com os respectivos textos em separado. Obviamente, isto ocorreria apenas caso não houvesse um sistema operacional para controlar o acesso à impressora.

A segunda função básica de um sistema operacional é funcionar como uma máquina estendida. Essa máquina estendida tem por função oferecer abstrações que sejam mais convenientes do que os recursos que são proporcionados diretamente pelo *hardware*. Como a arquitetura dos computadores em nível de linguagem de máquina é de difícil programação (TANEMBAUM, 2010), pode-se dizer que o desenvolvimento de programas que exigissem uma comunicação direta com o *hardware* seria consideravelmente mais difícil. Também é válido notar que cada dispositivo e cada fabricante têm suas próprias características. Caso os programadores tivessem que conhecer em detalhes o funcionamento do *hardware* da máquina na qual seu software será executado, o desenvolvimento de programas que funcionem em diversas máquinas com equipamentos de fabricantes diferentes seria impraticável. Assim sendo, o sistema operacional deve esconder os detalhes do funcionamento do *hardware* e oferecer uma interface para que os aplicativos possam acessar os recursos computacionais.

2.2 Conceitos de Hardware

Alguns conceitos referentes ao funcionamento do *hardware* dos computadores são importantes para o entendimento deste trabalho, pois as funções de um sistema operacional estão intimamente ligadas a esses conceitos. Além disso, alguns detalhes específicos dos processadores da família Intel também são parte do escopo desta seção, uma vez que o sistema operacional descrito por este trabalho tem como plataforma alvo os processadores desta família.

2.2.1 Microprocessador

Microprocessadores (ou processadores) são circuitos integrados passíveis de serem programados para executar uma tarefa predefina (TORRES, 2001, p. 17). Um microprocessador, ou CPU, é capaz de executar um conjunto de instruções, e possui um ciclo de execução no qual primeiro ocorre a busca na memória da instrução a ser executada na memória, e em seguida a definição de seus operandos (valores que serão manipulados pela instrução). Uma CPU executa seu ciclo até que todas as instruções de um programa sejam executadas.

O tempo de acesso à memória para buscar uma instrução ou operando é maior do que o tempo para executá-la, assim sendo, todas as CPUs têm registradores para o armazenamento das variáveis e resultados temporários (TANENBAUM, 2010, p. 12). Os processadores da família Intel (arquitetura x86 de 32 bits) possuem 16 registradores básicos para a execução de programas. Estes registradores são divididos em 4 tipos, como pode ser visto na tabela 1.

Tabela 1 - Registradores para execução de programas

Tipo	Nome	Função
Propósito Geral	EAX	Acumulador
	EBX	Ponteiro para dados
	ECX	Contador
	EDX	Ponteiro para E\S
	ESI	Ponteiro para operações com *Strings*
	EDI	Ponteiro para operações com *Strings*
	ESP	Ponteiro para pilha
	EBP	Ponteiro para dados na pilha
Segmento	CS	Registradores para armazenar seletores. Seletores são ponteiros que identificam segmentos de memória
	DS	
	SS	
	ES	
	FS	
	GS	
Status e de Controle	EFLAGS	Armazena *flags* de status, controle e do sistema
Ponteiro de Instrução	EIP	Indica o endereço da próxima instrução a ser executada.

Cada instrução é armazenada na memória, e cada posição da memória possui um endereço. O registrador EIP possuí o endereço da próxima instrução a ser executada pela CPU. Sempre que uma instrução é executada, o registrador EIP tem seu valor atualizado para posição de memória seguinte, a não ser que alguma instrução de transferência de controle seja executada.

As instruções de transferência de controle servem para permitir que o processador execute condicionalmente conjuntos de instruções, execute sub-rotinas, realize laços de repetição, executando um mesmo bloco de comandos diversas vezes. Algumas instruções de transferência de controle avaliam o registrador de status e de controle (EFLAGS) a fim de averiguar se uma determinada condição foi satisfeita pela instrução anterior, de forma a decidir se o valor de EIP e consequentemente o fluxo de execução das instruções deve ser alterado ou não. A CPU possui um conjunto de instruções lógicas e aritméticas, que alteram o valor do registrador EFLAGS de acordo com seu resultado. Desta maneira, o resultado de uma operação pode alterar o fluxo da execução do código pelo processador.

O sistema operacional precisa lidar com esses registradores para permitir que um programa seja interrompido e depois continue a execução exatamente de onde parou. Para tal é executado um procedimento conhecido como troca de contexto que envolve a copia e substituição dos valores contidos nesses registradores.

Além dos registradores, outra estrutura tem fundamental importância na execução de um programa. A pilha (*stack*), composta por um vetor contínuo de posições de memória, é utilizada para o controle das chamadas a sub-rotinas e para a passagem de parâmetros. A pilha é controlada através dos registradores ESP e EBP. A cada programa de usuário, bem como ao próprio *kernel* é atribuído uma espaço de memória que deve ser devidamente referenciado por esses registradores para que este possa servir como pilha.

A pilha é modificada através das instruções *push* (empilha um valor) e *pop* (desempilha um valor). Sempre que uma dessas instruções é executada, o valor contido no registrador ESP é alterado. A figura 1 mostra a estrutura da pilha durante a execução de um programa no qual há um procedimento que realiza a chamada de outra sub-rotina. É possível notar que as variáveis locais, parâmetros e endereços de retorno são armazenados na pilha, pois desta maneira, a sub-rotina chamada pode acessar através dela os parâmetros, e pode posteriormente retornar o controle da execução para o procedimento que faz a chamada.

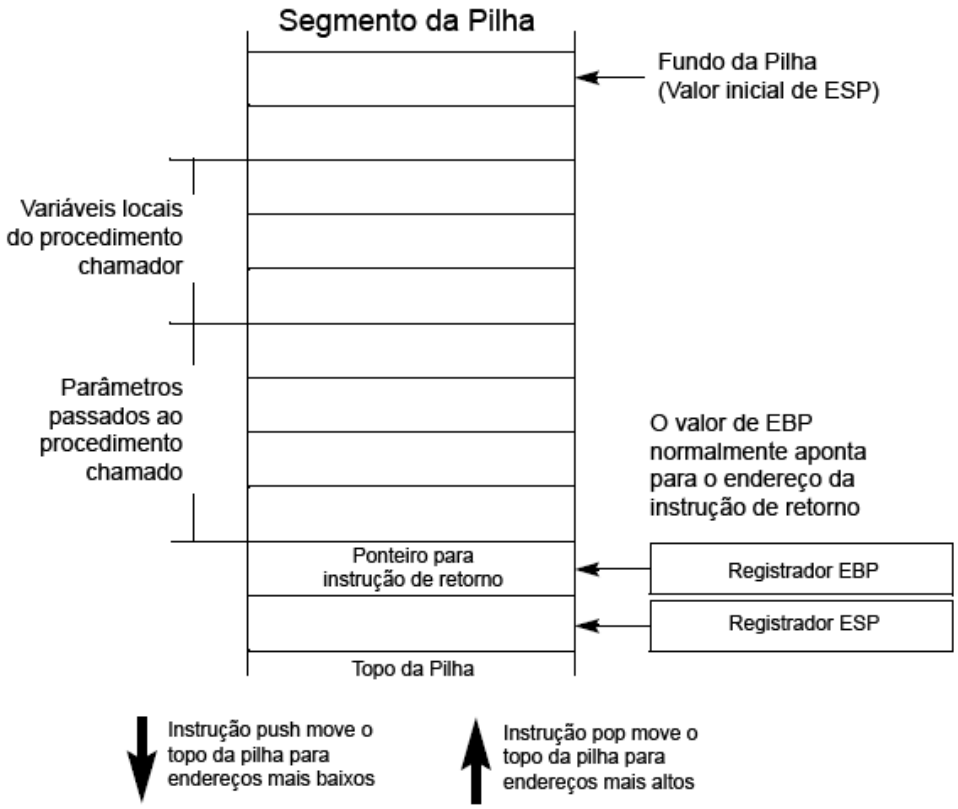

Figura 1 - Estrutura da pilha durante a execução de um programa (INTEL, 2011)

2.2.2 Memória

Outro elemento importante em um computador é a memória. De acordo com o manual para de desenvolvimento de software da Intel (2011), a memória que um processador consegue acessar é chamada de memória física. Essa memória física é organizada como uma

sequência de bytes (compostos por 8 bits). Cada byte pode ser acessado através de um único endereço, conhecido como endereço físico.

Genericamente, o termo memória RAM (*Random Access Memory*) é utilizado para designar a memória principal do computador. A quantidade de memória física que o processador consegue acessar depende da quantidade de memória RAM que está disponível no sistema. É válido notar que tanto os dados quanto as instruções dos programas ficam armazenados na memória durante sua execução. Isto porque, o processador só é capaz de executar instruções que estejam residentes em memória.

Sendo o processador mais rápido do que a memória RAM, foi preciso que os projetistas desenvolvessem uma solução para evitar que o processador ficasse ocioso até a memória estar em condições de responder a uma solicitação de leitura ou escrita de dados. Para evitar esse problema, os processadores possuem a memória *cache*, que consiste em uma pequena quantidade de memória RAM construída com uma tecnologia diferente (memória estática), sendo mais rápida e mais cara que a memória principal convencional. A memória *cache* funciona como intermediária entre o processador e a memória principal, permitindo que o processador troque dados com a memória em sua velocidade máxima (TORRES, 2001, p. 31).

2.2.3 Memória Virtual

As instruções de um programa devem ser carregadas na memória do computador para que possam ser executadas pela CPU. A princípio, um programa que exigisse mais memória do que a quantidade presente no sistema não poderia funcionar corretamente, pois, certamente, uma parte do mesmo não estaria na memória.

Para permitir que programas fossem postos em execução em computadores que não possuíssem memória suficiente, foi desenvolvida uma técnica conhecida como *overlay*. Os programas eram divididos em pedaços, chamados *overlays*, Os *overlays* eram mantidos em

disco e eram levados para dentro e para fora da memória pelo sistema operacional, dinamicamente, conforme necessário (TANENBAUM & WOODHULL, 2000, p. 218).

O problema com essa técnica era a grande dificuldade de implementação, de modo que, uma nova solução acabou surgindo para facilitar a vida dos desenvolvedores, a memória virtual. Memória virtual é uma técnica sofisticada e poderosa de gerência de memória, onde as memórias principal e secundária são combinadas, dado ao usuário à ilusão de existir uma memória muito maior do que a capacidade real da memória principal (MACHADO & MAIA, 2007, p. 171).

A memória virtual se utiliza de uma abstração, o espaço de endereçamento virtual, que é composto por um conjunto de endereços virtuais. Os programas são criados para fazer referência apenas a endereços virtuais, e não mais aos endereços reais da memória principal. A cada acesso do programa a memória, deve ocorrer a tradução do endereço virtuais referenciado, por um endereço físico. Essa tradução é feita em blocos, de maneira que um bloco de endereços virtuais é traduzido em um bloco contíguo de endereços físicos.

O processo de tradução poderia ser muito custoso em termos de tempo de execução caso tivesse que ser realizado pelo *software.* Por isso, o *hardware* deve fornecer algum recurso que permita a tradução dos endereços virtuais em físicos. Para tanto, duas técnicas são geralmente utilizadas pelos processadores, a paginação e a segmentação. A diferença entre elas é que, na paginação, os blocos que são mapeados têm sempre o mesmo tamanho, enquanto na segmentação, os blocos podem ter tamanhos variados. É válido notar que alguns processadores, como os da família Intel, suportam uma combinação das duas técnicas.

De forma resumida, a paginação funciona da seguinte maneira. Como mostra a figura 2, junto a CPU existe uma unidade de gerenciamento de memória (MMU - *Memory Management Unit*) capaz de traduzir endereços de memória virtuais em endereços físicos.

Essa tradução é feita através de tabelas que podem ser configuradas pelo sistema operacional, permitindo que blocos de memória física (*frames* ou quadros) sejam mapeados em um conjunto de endereços virtuais (páginas). Um endereço virtual é dividido em partes, de maneira que alguns *bits* são utilizados como índice de busca em tabelas de páginas, que armazenam os endereços físicos de memória. A parte final do endereço especifica a posição de memória a ser a acessada dentro do bloco referenciado (deslocamento) pela entrada na tabela de página cujo índice é especificado pela primeira parte do endereço virtual.

Caso um programa de computador seja criado para utilizar endereços virtuais, o sistema operacional pode colocar apenas uma parte do programa na memória física e manter outra parte na memória secundária, e realizar trocas entre o disco e a memória principal à medida que o programa necessite de partes que não estão fisicamente na memória.

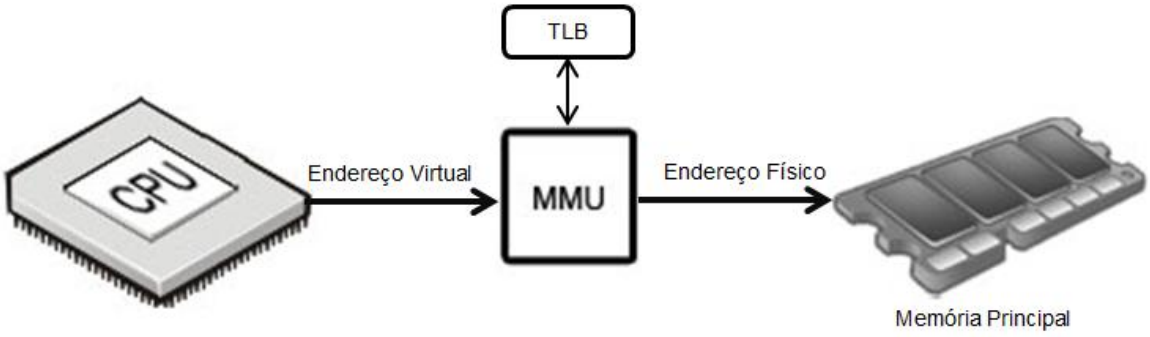

Figura 2 - Processador e MMU

Isto é possível porque o espaço de endereçamento virtual não está limitado à quantidade de memória física disponível na máquina. Os processadores da família Intel de 32 bits, por exemplo, podem ter espaços de endereçamento virtual de até 2^{32} bytes, mesmo que apenas uma parte disso esteja disponível como memória física. O sistema operacional pode manter um programa ocupando uma parte desse espaço de endereçamento virtual, ou mesmo

criar um espaço de endereços isolados para cada programa compartilhando a memória do computador.

Como as tabelas utilizadas no mapeamento contém muitos valores, elas geralmente são armazenadas na memória principal. Isto faz com que, a princípio, seja necessário realizar dois acessos à memória para a execução das instruções. Um para verificar as tabelas de página e realizar a conversão dos endereços virtuais em físicos, e outros para o acesso a instrução propriamente dita. Entretanto, como a velocidade de execução geralmente é limitada pela frequência com que a CPU pode acessar instruções e dados na memória, o fato de haver necessidades de dois acessos por referência à memória reduz o desempenho pela metade (TANEBAUM, 2010, p. 119). A solução para esse problema é colocar uma parte mais utilizada das tabelas em uma memória *cache*, chamada de memória associativa ou TLB (*table lookaside buffer*), de maneira a tornar o processo de mapeamento muito mais rápido.

O mapeamento de endereços ocorre de maneira muito similar à técnica da segmentação. A CPU se utiliza de uma tabela, cujas entradas especificam os segmentos. Cada entrada na tabela possui informações sobre o segmento, como sua posição inicial e seu tamanho. O segmento atualmente utilizado pelo processador é armazenado em um registrador da CPU para que quando um acesso à memória seja feito pelo programa, o endereço acessado seja somado ao endereço da posição inicial do segmento. Desta maneira, é possível fragmentar programas em partes de tamanhos variáveis, e criar um segmento para cada um deles, usando endereços virtuais relativos ao começo do segmento, que serão traduzidos para endereços físicos dependendo da posição em que o segmento estiver na memória física.

2.2.4 Entrada e Saída

Além do processador, um sistema computacional possui dispositivos que permitem a entrada e saída de dados, como teclados, *mouses*, monitores, impressoras, discos rígidos, entre outros. Sem eles não haveria uma forma dos usuários interagirem com o

computador. Os dispositivos de E\S possuem atrelados a si um *hardware* controlador, que oferece uma interface para manipulação do *hardware* pelo *software.* Esses controladores possuem registradores, e a leitura e escrita nesses registradores permite o controle sobre os dispositivos.

TANENBAUM (2010) cita duas formas amplamente utilizadas para permitir o acesso do processador a esses registradores. A primeira envolve a existência de instruções suportadas pela CPU (no caso dos processadores da família Intel, as instruções IN e OUT) que servem para acessar os registradores em um espaço especial de portas de E/S. Este espaço de portas nada mais é do que o conjunto de todos os registradores para acesso aos controladores dos dispositivos de entrada e saída. A segunda forma é o mapeamento deste espaço de portas na memória principal, tornando possível o acesso aos registradores através da leitura e escrita dos dados em posições específicas da memória. Essa técnica não depende de instruções especiais de entrada de saída, porém uma área do espaço de endereçamento é usado para tal.

Além disso, eventualmente, os dispositivos de entrada e saída devem se comunicar com o processador de alguma forma. Tomando como o exemplo o teclado, que deve sinalizar ao processador quando uma tecla for pressionada. Esta sinalização é feita através de interrupções que são tratadas na seção 2.2.5.

2.2.5 Interrupções

Diversos eventos ocorrem em um sistema computacional. Esses eventos causam desvios forçados na execução de um programa (MACHADO & MAIA, 2007, p. 41), as chamadas interrupções. As interrupções podem ocorrer por diversos motivos, tendo algumas origens distintas, como é possível ver na figura 3. Um dispositivo de *hardware* pode utilizar uma interrupção para sinalizar a ocorrência de um evento ao processador. Um exemplo disso

seria o teclado enviando uma interrupção ao processador, indicando o pressionamento de uma tecla.

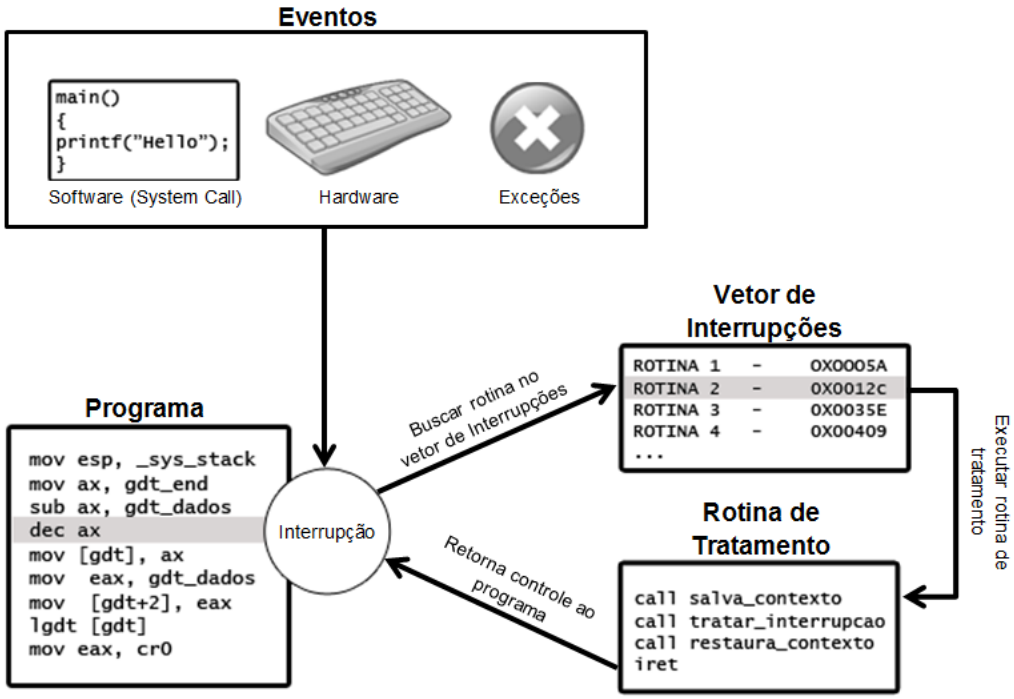

Figura 3 - Tratamento de Interrupções

O *software* também pode gerar interrupções. Um programa de computador que tente, por exemplo, executar instruções inválidas, como um acesso a uma posição de memória de outro programa, ou mesmo realizar uma divisão por zero, podem gerar uma exceção, ou seja, uma interrupção causada por um erro. É possível também, que um programa necessite chamar alguma rotina do sistema operacional, podendo se utilizar de interrupções, que no caso são chamadas ao sistema ou *system calls.*

Os sistemas operacionais modernos são baseados em interrupções (SILBERSCHATZ et al., 2000, p. 16), isto é, após a inicialização, um SO tem suas rotinas executadas apenas quando é necessário tratar alguma interrupção. Interrupções são importantes, pois através delas o sistema operacional consegue, por exemplo, finalizar a execução de programas defeituosos. Os temporizadores do sistema computacional, por sua

vez, geram uma interrupção a cada intervalo de tempo, permitindo que o SO divida o tempo de execução pela CPU entre diversos programas.

A CPU possui uma tabela com ponteiros para rotinas de tratamento de interrupções, chamado vetor de interrupções. Sempre que uma interrupção ocorre, o processador paralisa temporariamente o programa em execução, salva o estado de todos os registradores da CPU, e inicializa a rotina de tratamento referente à interrupção indicada no vetor de interrupções. Após o término da execução da rotina de tratamento da interrupção, os valores contidos anteriormente nos registradores são restaurados para que o programa cuja execução foi interrompida possa continuar de onde parou. Durante o processo de inicialização, o SO configura o vetor com ponteiros para as suas próprias rotinas de tratamento, deste modo, há a garantia de que o SO entrará em ação sempre que uma interrupção ocorrer.

2.2.6 Proteção de Hardware

O *hardware* deve oferecer alguns mecanismos que permitam um maior controle sobre o sistema computacional. Sem esses mecanismos, o sistema operacional não teria como arbitrar a utilização dos recursos computacionais, e qualquer *software* defeituoso ou mal intencionado poderia causar grandes danos a todo o sistema.

O processador usualmente possui modos de execução que permitem a regulação sobre quais tipos instruções são ou não permitidas para o programa em funcionamento na CPU. São necessários pelo menos dois modos de operação separados: o modo usuário e modo monitor (também conhecido como modo supervisor ou modo *kernel*) (SILBERSCHATZ et al., 2000, p. 26) para que o sistema operacional tenha controle efetivo sobre a CPU.

O SO possui um conjunto de rotinas principais que compõem o seu núcleo, ou *kernel.* Essas rotinas executam instruções que alteram o funcionamento interno da CPU, ou mesmo que fazem operações de entrada e saída. Caso essas instruções pudessem ser

executadas por um programa de usuário, este poderia tomar o controle do computador, ou mesmo causar diversas falhas em outros programas em execução.

Essas instruções são conhecidas como instruções privilegiadas, e para garantir que não sejam chamadas por programas de usuário, tais instruções podem ser executadas apenas no modo supervisor, enquanto os programas de usuário só podem ser executados no modo usuário. Antes de colocar um *software* de usuário em funcionamento, o sistema operacional altera o modo de funcionamento através da alteração do valor de algum registrador do processador, impossibilitando o programa de executar instruções privilegiadas.

Para permitir que o sistema operacional volte a ter suas rotinas executadas em modo supervisor, o *hardware* automaticamente altera o modo de funcionamento da CPU sempre que uma interrupção ocorre. Uma vez que as rotinas do sistema operacional são executadas apenas para o tratamento dos eventos causadores de interrupções, existe a garantia de que somente o SO irá operar em modo supervisor.

Além de impedir a execução de instruções privilegiadas, o sistema operacional precisa impossibilitar que os programas acessem posições de memória indevidas. Supondo que um programa de usuário pudesse acessar qualquer posição de memória, este poderia ler e escrever dados nos endereços reservados ao próprio núcleo do sistema operacional. O resultado disto seria imprevisível, e a estabilidade do sistema estaria seriamente comprometida. Para evitar esse tipo de problema, o *hardware* deve implementar alguma forma de proteção da memória do computador.

Uma forma usual é a utilização de registradores que delimitassem as posições de memória que cada programa de usuário pode utilizar. Outra forma mais elaborada é aplicada através da utilização da memória virtual. A segunda forma é utilizada pelo sistema operacional FESO, e é descrita no capítulo sobre gerência de memória.

2.3 Componentes de um Sistema Operacional

Existem alguns componentes básicos que são comuns à maioria dos sistemas operacionais. Para compreender o funcionamento de um sistema é imperativo entender a função de cada um desses componentes, e como eles se relacionam entre si. Nesta seção, são descritos resumidamente alguns dos principais componentes de um sistema operacional.

2.3.1 Gerência de Memória

A memória do computador pode ser entendida como um grande vetor, cada uma das posições desse vetor têm um endereço, e em cada uma delas um valor (geralmente 1 byte) pode ser armazenado. É papel do sistema operacional controlar o acesso a memória, uma vez que os programas necessitam dela para funcionar. Segundo STALLINGS (2011), alguns dos requisitos que devem ser atendidos por um sistema de gerência de memória são: Proteção, Compartilhamento e Organização física e lógica.

O sistema de gerência de memória deve controlar quais áreas de memória podem ser usadas, e quais não estão livres. Deve permitir a alocação e liberação da memória na medida em que ela é usada pelos programas. Deve tornar possível o carregamento de novos programas e fornecer algum mecanismo para alocação de posições de memória em tempo de execução.

No caso de sistemas que suportem mais de um programa em execução ao mesmo tempo, a proteção do espaço de memória ocupada por cada programa é fundamental. Sem essa proteção, um programa poderia inadvertidamente acessar uma posição de memória que não lhe pertence, e causar um mau funcionamento no programa cuja memória foi acessada sem permissão.

Em outros casos, pode ser necessário que dois programas compartilhem dados em uma mesma posição da memória, devendo a gerência de memória oferecer algum mecanismo

que permita esse tipo de compartilhamento. De qualquer maneira, mesmo esse acesso às áreas compartilhadas de memória deve ser devidamente controlado pelo sistema operacional.

Os processadores da família Intel oferecem alguns mecanismos que auxiliam o sistema de gerência de memória a preencher tais requisitos. O principal deles é a memória virtual, que permite ao *software* organizar a memória física de uma forma mais flexível.

2.3.2 Gerência de Processos

Um programa de computador geralmente é expresso através de um código, que é transformado e armazenado de forma que uma CPU possa executá-lo. Programas são armazenados em arquivos, e então carregados na memória para serem executados. Um processo pode ser considerado como um programa em execução (SILBERSCHATZ et al., 2000, p. 33), muito embora, o programa em si seja apenas o conjunto de instruções, e o processo seja uma instância daquele programa na memória do computador.

Para cada programa que é executado em um computador, o sistema operacional cria ao menos um processo para controlar sua execução. O sistema operacional precisa armazenar diversas informações sobre cada processo. O SO necessita do estado dos registradores para que seja possível interromper o programa e colocá-lo em execução exatamente de onde parou. Além disso, é preciso saber quais blocos de memória o processo ocupa, e até mesmo se ele está pronto para ser executado ou se está esperando por outro processo. O sistema deve possuir também uma lista dos processos prontos para a execução para decidir qual será o próximo a ter permissão de usar a CPU.

Contudo, é usual que algum componente do sistema operacional seja responsável por controlar os processos. Este componente é conhecido como a gerência de processos. O papel da gerência de processos é criar e remover processos da memória, bem como suspender a execução de um processo, permitindo que cada um deles tenha uma chance de ser executado

pela CPU. Além disso, a gerência de processos pode incorporar mecanismos para realizar a comunicação e sincronização dos processos.

2.3.3 Gerência de Arquivos

Uma das necessidades mais básicas dos usuários de computador é manter um conjunto de informações e instruções para acesso futuro. Esses dados e instruções ficam salvos em algum sistema de armazenamento secundário, como discos magnéticos, CDs, memórias *flash* entre outros. Os dados são armazenados de forma diferente em cada um desses dispositivos. Porém, o sistema operacional abstrai as propriedades físicas de seus dispositivos de armazenamento para definir uma unidade lógica de armazenamento, o arquivo (SILBERSCHATZ et al., 2000, p. 35).

Dessa forma, um arquivo é uma abstração que representa um conjunto de dados ou instruções armazenado em algum dispositivo. O arquivo é composto por um conjunto dados, sendo que o significado e a posição desses dados e instruções dentro do arquivo dependem geralmente única e exclusivamente de seu criador.

O sistema de gerência de arquivos é o conjunto de sistemas de *software* que fornece aos usuários e aplicativos serviços para a utilização de arquivos (STALLINGS, 2011, p. 524). De forma resumida, sistema de gerência de arquivos deve permitir a criação, exclusão, leitura, escrita e modificação de arquivos.

2.3.4 Sistema de E/S

Uma das funções do sistema operacional é esconder os detalhes de funcionamento do *hardware* e fornecer abstrações que sejam mais convenientes de se utilizar. O sistema de E/S, composto principalmente por *drivers* de dispositivos, é responsável por realizar essa tarefa.

Os *drivers* de dispositivo têm como função receber comandos gerais sobre acessos aos dispositivos e traduzi-los para comandos específicos, que poderão ser executados pelos controladores (MACHADO & MAIA, 2007, p. 233). São os *drivers* de dispositivo que conhecem os detalhes do funcionamento dos dispositivos de *hardware*. Cabe aos *drivers* se comunicarem com os controladores de cada periférico permitindo assim que eles sejam corretamente utilizados.

Cada *driver* de dispositivo implementa os detalhes de comunicação para o *hardware* que devem controlar. Para tanto, os *drivers* de dispositivo eventualmente precisam ser escritos em uma linguagem de programação de mais baixo nível, como o *assembly*, uma vez que as instruções que realizam entrada e saída de dados não existem em linguagens de alto nível como o C ou C++.

2.4 Arquitetura e Tipos de Sistemas Operacionais

Existem diversos tipos de sistemas operacionais. Existem aqueles que são projetados para computadores pessoais, outros para servidores e computadores de grande porte, outros que são utilizados em dispositivos móveis, e até mesmo sistemas operacionais embarcados. É possível classificar tais sistemas de acordo com a sua capacidade de compartilhar os recursos computacionais entre diversos programas ao mesmo tempo.

São chamados de sistemas monotarefa ou monoprogramáveis aqueles que permitem que somente um programa utilize os recursos computacionais em dado instante de tempo. Um problema com esse tipo de sistema é a subutilização dos recursos computacionais. Como os recursos estão alocados exclusivamente para um programa, muitas vezes não são utilizados integralmente. É o caso, por exemplo, do processador que fica ocioso enquanto o programa espera por uma operação de entrada ou saída. Ou mesmo da memória, quando o programa não a ocupa totalmente. Uma vantagem dos sistemas monotarefa é que eles são de

simples implementação em comparação com outros tipos de sistema, não existindo muita preocupação com problemas decorrentes do compartilhamento de recursos, como memória, processador e dispositivos de E/S (MACHADO & MAIA, 2007, p. 15).

Com a evolução dos sistemas operacionais, surgiram os chamados sistemas operacionais multitarefa ou multiprogramáveis. Primeiramente vieram os sistemas em lotes que permitiam diversos programas (neste contexto, são chamados de *jobs*) compartilhando a memória simultaneamente. Esses sistemas eram capazes de colocar um determinado *job* em execução enquanto outro estava ocupado realizando uma operação de E/S de dados. Com isso, a CPU não permanecia ociosa enquanto existia algum *job* pronto para a execução.

Porém, os sistemas em lotes eram utilizados em tarefas que não necessitavam de constante interação com usuários. Os *jobs* eram postos em execução e poderiam ficar longos períodos em execução até que estivessem prontos para realizar a saída de algum dado. Para permitir a interação dos usuários com o programas em sistema multiprogramáveis, surgiram os sistemas de tempo compartilhado. Esses sistemas possibilitam que diversos programas interajam com os usuários, dando a cada um dos programas uma pequena fatia de tempo para que sejam executados pela CPU. Os sistemas de tempo compartilhado alternam a execução entre programas rapidamente, de forma a dar aos usuários a impressão de que existem diversos programas sendo executados ao mesmo tempo. O sistema operacional deve garantir que os programas não interfiram uns nos outros e deve executar o planejamento da execução de cada um deles (SHAY, 1996, p. 10).

Outra maneira de classificar sistemas operacionais é de acordo com a estrutura interna do seu núcleo ou *kernel*. TANENBAUM (2010) classifica a estrutura dos sistemas operacionais em seis tipos básicos: sistemas monolíticos, sistema de camadas, micronúcleos, sistemas cliente-servidor e máquina virtual. Para os objetivos desse trabalho, faz-se necessária a definição de sistemas monolíticos e micronúcleos, uma vez que o sistema operacional

descrito neste trabalho é estruturado de acordo com características dessas duas arquiteturas. A figura 4 demonstra algumas diferenças entre as duas arquiteturas.

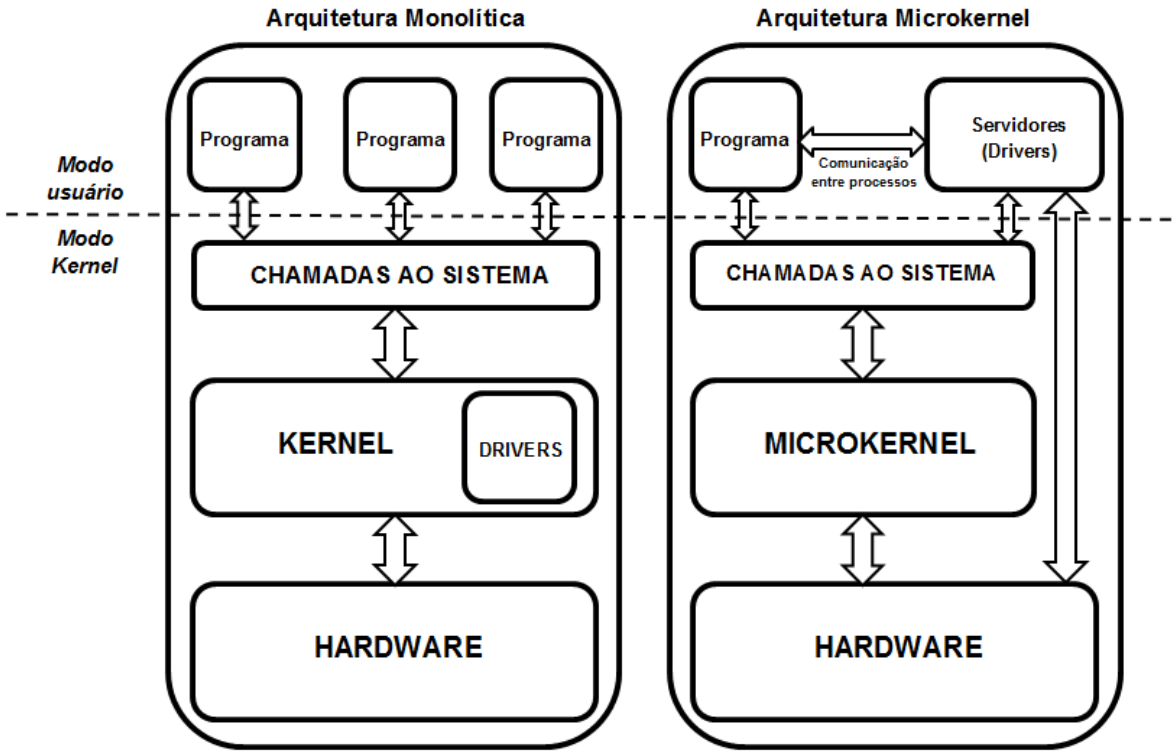

Figura 4 - Arquitetura monolítica e microkernel

Os sistemas monolíticos são aqueles que, quando postos em execução, funcionam como um único programa na memória do computador, não existindo isolamento entre o *kernel* e os drivers de dispositivos durante a execução do sistema. Para criar esse tipo de sistema é necessário que todos os seus módulos sejam compilados e unidos (*linkados*) em um único arquivo-objeto. Sistemas monolíticos são mais simples e possuem um bom desempenho em comparação com outras arquiteturas, mas como não existe uma estrutura bem definida, e não há isolamento entre as rotinas, a manutenção desse tipo de sistema acaba sendo dificultada.

Os micronúcleos ou *microkernels*, por sua vez, são uma forma diferente de arquitetura do sistema operacional. A ideia por trás dos micronúcleos é de manter no *kernel*

apenas algumas funcionalidades principais, de forma que outras funcionalidades sejam executadas externamente na forma de programas de usuário. Nesta abordagem, os *drivers* de dispositivos ou servidores, que funcionam fora do *kernel*, utilizam alguma forma de comunicação entre processos para atender as solicitações dos outros aplicativos de usuário (clientes).

Existe pouco consenso com relação à quais serviços devem permanecer no *kernel* e quais devem ser implementados no espaço de usuário. Em geral, os *microkernels* geralmente fornecem gerência mínima de memória e processos, além de um recurso de comunicação (SILBERSCHATZ *et al.*, 2000, p. 49).

Em seu artigo sobre a construção de *microkernels*, LIEDTKE (1995) cita algumas vantagens desse tipo de abordagem. Entre elas, estão o isolamento dos servidores e o *kernel*, e a possibilidade da coexistência de diferentes estratégias e interfaces para comunicação com os servidores.

Apesar das vantagens, a arquitetura *microkernel* possui também um ponto fraco. Micronúcleos são de difícil implementação, pois existe um problema de desempenho, devido à necessidade de mudança de modo de acesso a cada comunicação entre clientes e servidores (MACHADO & MAIA, 2007, p. 62).

3 DESENVOLVIMENTO DO KERNEL

3.1 Metodologia

O desenvolvimento do sistema operacional FESO exigiu a pesquisa e o entendimento de diversos conceitos relacionados ao funcionamento de sistemas operacionais. Neste ponto, a literatura forneceu as bases teóricas para a construção do sistema. Entretanto, a consulta a diversos manuais e documentações técnicas também foi necessária, uma vez que estes tipos de documentos contém detalhes imprescindíveis, sobretudo, no desenvolvimento de *drivers* de dispositivos, e no controle do processador.

As linguagens escolhidas para a codificação do sistema FESO foram o C/C++ (CPLUSPLUS, 2013) e o *assembly*. De acordo com TANEMBAUM (2010), sistemas operacionais são normalmente grandes programas em C (ou algumas vezes C++). O C é uma linguagem imperativa que já foi utilizada no desenvolvimento de outros sistemas operacionais e, por isso, foi a escolha natural para a escrita do sistema FESO.

Porém, a linguagem C, por ser de alto nível, foi desenvolvida para ser independente da arquitetura de máquina. Linguagens de alto nível raramente levam em conta características especiais da máquina na qual seu programa irá funcionar (HYDE, 2003). Desta forma, o uso da linguagem *assembly* se faz necessário em algumas rotinas para a execução de instruções específicas dos processadores da família Intel (plataforma alvo do sistema FESO). Algumas instruções, como as utilizadas para o controle da CPU, acesso a registradores e entrada e saída de dados, não podem ser expressas diretamente em C e são suportadas apenas na linguagem *assembly*.

De forma similar a outros *softwares*, um sistema operacional é primeiro codificado em uma linguagem e depois é transformado em código de máquina através do uso de um compilador. Os compiladores usados na criação do FESO foram o GCC (GNU

compiler collection) (FREE SOFTWARE FOUNDANTION, 2013) e o NASM (*netwide assembler*) (THE NASM TEAM, 2012) para as linguagens C/C++ e *assembly* respectivamente. Porém, alguns cuidados especiais são utilizados na compilação dos arquivos fonte do sistema.

Primeiramente, o FESO utiliza uma versão especialmente produzida do GCC para realizar a compilação cruzada (*cross-compilation*). A compilação cruzada é a construção em uma plataforma de um arquivo binário que será executado em outra plataforma (FREE SOFTWARE FOUNDATION, 2012). Os arquivos binários que são utilizados no FESO não são do mesmo formato que os suportados nativamente pelo sistema operacional no qual o seu desenvolvimento foi feito. Assim sendo, a compilação cruzada é utilizada para permitir a geração correta dos arquivos binários.

Além disso, nenhum recurso ou biblioteca da linguagem C/C++ que exija suporte de sistema operacional pode ser utilizado, pois, este suporte não existirá para o FESO. Um sistema operacional, diferentemente de um programa de usuário, deve funcionar de forma independente. Qualquer biblioteca ou função da linguagem que exija suporte do SO deve ser primeiramente desenvolvida ou portada para funcionar com o FESO, pois é o próprio sistema que deve dar suporte a este tipo de recurso.

De forma resumida, o desenvolvimento do sistema operacional se deu a partir de seu núcleo (*kernel*) que é composto pelas rotinas mais importantes do sistema. Essas rotinas possuem dependências entre si de forma que a ordem do desenvolvimento dos componentes do sistema FESO foi escolhida com base nessas dependências. Por exemplo, a gerência de processos depende da alocação de memória. O sistema de comunicação entre processos, por sua vez, depende da gerência processos. A interface de chamadas ao sistema faz uso desses e de outros componentes. Desta maneira, a ordem na qual essas rotinas foram desenvolvidas e testadas foi definida por suas dependências.

Estando o núcleo capaz de oferecer um conjunto suficiente de funcionalidades, novos programas foram criados para funcionar sobre ele. À medida que novas funções eram implementadas no núcleo, era preciso desenvolver algum aplicativo para testá-la. Esses testes foram realizados com a utilização de uma máquina virtual, e em alguns casos, em *hardware* real. A utilização da máquina virtual permitiu que novas rotinas sejam testadas sem a necessidade de realizar a inicialização do sistema no *hardware* real, que é consideravelmente mais demorada. Porém, testes em *hardware* real são necessários, uma vez que existem algumas diferenças entre o funcionamento das máquinas virtuais e dos dispositivos físicos.

3.2 Arquitetura

O sistema FESO foi desenvolvido usando alguns princípios presentes em sistemas com arquitetura *microkernel.* Os *drivers* de dispositivos funcionam fora do núcleo do sistema, e possuem seu próprio espaço de endereçamento. O núcleo, por sua vez, oferece apenas um limitado conjunto de funcionalidades básicas, que são utilizadas tanto pelos *drivers* quanto pelos aplicativos de usuário.

As rotinas do núcleo ou *kernel* são as mais importantes do sistema. Essas rotinas precisam ter total controle sobre o *hardware*, podendo assim executar todo o tipo de instrução, incluindo as instruções privilegiadas da CPU, e as instruções para realizar a entrada e saída de dados. O *kernel* do sistema FESO possui 4 componentes principais, são eles: Gerência de Memória, Gerência de Processos, Sistema de Troca de Mensagens e Sistema de Arquivos Virtual. Cada um desses componentes será detalhado em uma seção a parte.

Os *drivers* de dispositivos são implementados como programas de usuário. Cada um deles possui seu próprio espaço de endereçamento, e acessam as funções do núcleo através de chamadas ao sistema. Os *drivers* de dispositivo não têm permissão para executar instruções privilegiadas da CPU. Porém, eles podem executar instruções de entrada e saída

para possibilitar o acesso aos controladores dos dispositivos de E/S. Alguns exemplos de *drivers* desenvolvidos junto com o sistema FESO são: *driver* do relógio (TIMER), *driver* do console (vídeo e teclado), *driver*s para sistema de arquivos, entre outros.

Em um menor nível de privilégio estão os aplicativos de usuário. Esse tipo de *software* não pode executar instruções privilegiadas, ou instruções de entrada e saída. Isto porque, caso aplicativos de usuário pudessem executar este tipo de instrução, eles seriam capazes de tomar o controle do computador, ou mesmo controlar indevidamente os dispositivos de entrada e saída. Exemplos de aplicativos de usuário são: Interpretador de Comandos (SHELL) e editor de texto.

Os mecanismos de proteção do processador reconhecem 4 níveis de privilegio numerados de 0 a 3 (Intel, 2011). Quanto menor o nível, mais privilégios ele tem. A figura 5 mostra a distribuição dos componentes do sistema FESO através dos níveis de privilégio.

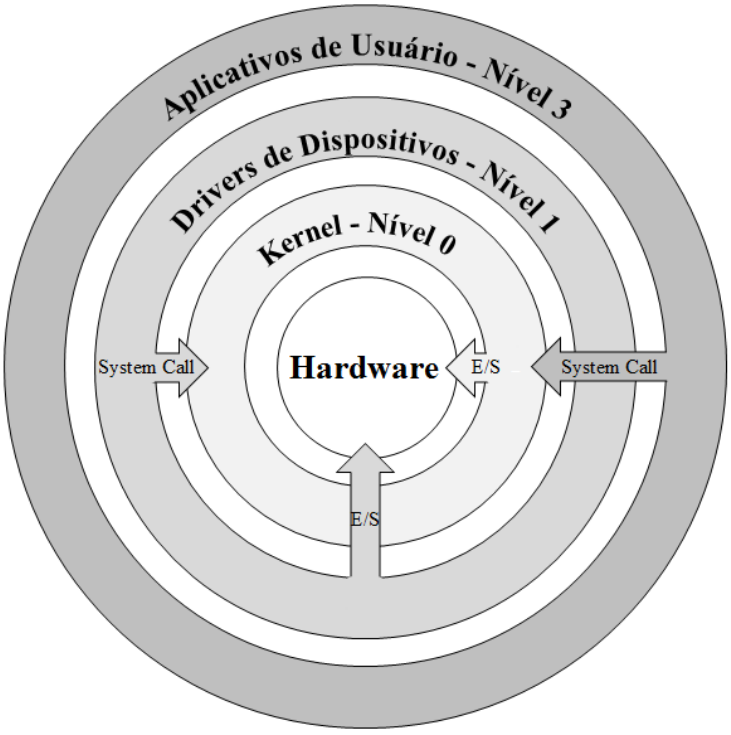

Figura 5 - Arquitetura do sistema FESO

O núcleo do sistema operacional tem o nível de privilégio 0, podendo executar todas as instruções da CPU. Os *drivers* de dispositivos funcionam no nível de privilégio 1, podendo executar instruções de entrada e saída, e consequentemente realizar a comunicação com o *hardware*. Porém, não são capazes de executar instruções privilegiadas. Finalmente, no nível 3 estão os aplicativos de usuário, que não podem executar instruções privilegiadas nem de entrada e saída de dados. O nível 2 não é utilizado pelo sistema FESO, pois apenas três níveis de privilegio são suficientes, para o núcleo, os *drivers* e os aplicativos de usuário.

3.3 Processo de Boot

O sistema operacional executa uma série de rotinas antes que o primeiro programa de usuário possa ser posto em execução. Essas rotinas envolvem a configuração correta das estruturas utilizadas pela CPU, a inicialização dos subsistemas do SO, a detecção e inicialização de novos processadores, entre outras tarefas.

De maneira resumida, os passos para a inicialização do FESO são os seguintes:

- Inicialização da gerência de memória
- Instalação das tabelas de descritores e interrupções
- Inicialização dos demais subsistemas (processo, mensagens e VFS)
- Carregamento dos módulos (Drivers de Dispositivos e Aplicativos)
- Detecção de inicialização de processadores

Após a execução de todos esses passos, o sistema FESO habilita as interrupções da CPU, e fica em estado de espera até que uma interrupção ocorra, e coloque o algoritmo de escalonamento de processos em execução. Este algoritmo coloca em execução os processos que foram carregados como módulos. Após isso, o sistema está em pleno funcionamento. Esta seção detalha algumas fases do processo de inicialização do sistema FESO.

3.3.1 GRUB

O sistema FESO utiliza o GNU GRUB (*GRAND UNIFIED BOOTLOADER*) como seu *boot loader*. Um *boot loader* é o primeiro programa a ser executado quando um computador é inicializado. Ele é responsável pelo carregamento e transferência de controle para o *kernel* do sistema operacional (FREE SOFTWARE FOUNDATION, 2010).

O GRUB carrega o sistema FESO na memória do computador, junto com uma série de módulos que são indicados através de um arquivo de configuração. Além disso, o GRUB cria na memória estruturas que permitem ao sistema operacional conhecer as posições nas quais esses módulos foram carregados. O GRUB entrega o controle do computador ao SO em um estado conhecido, e também cria uma estrutura acessível ao sistema com um mapa das regiões de memória que podem ser utilizadas por ele.

A utilização do GRUB dispensa a necessidade de desenvolvimento de rotinas de inicialização do SO que sejam genéricas o suficiente para funcionar em diversos computadores diferentes. Sempre que alguma alteração no SO precisa ser testada, uma nova imagem de CD de inicialização com o sistema FESO e o GRUB é criada, e partir dessa combinação, é possível testar o sistema, tanto em máquinas reais quando em máquinas virtuais.

3.3.2 Tabelas de Descritores

Após a entrega do controle do computador ao núcleo do sistema operacional, uma série de rotinas é executada. Cada um dos subsistemas do núcleo é iniciado. As funções para inicialização do sistema de gerência de memória, gerência de processos, troca de mensagens e sistema de arquivos virtual são devidamente executadas para realizar as configurações iniciais.

Entretanto, após a inicialização do sistema de gerência de memória, algumas estruturas responsáveis pelo controle da CPU devem ser corretamente criadas na memória, e

suas posições iniciais devem ser informadas ao processador. Essas estruturas são conhecidas como tabelas de descritores. As duas principais tabelas de descritores utilizadas em processadores da família Intel são a GDT e IDT.

A GDT (*Global Descriptors Table*) é uma tabela que contém entradas chamadas de descritores de segmento. Os descritores de segmento fornecem o endereço base do segmento, os direitos de acesso, tipo e outras informações (INTEL, 2011). A GDT permite que a memória seja separada em segmentos, porém o sistema FESO configura esta tabela de forma a possibilitar a existência de apenas um grande segmento que ocupe todo o espaço de endereçamento.

A GDT possui diversos tipos de descritores, de maneira que, para cada nível de privilégio usado pelo SO existam 2 deles, um para código e um outro para dados. O controle do nível de privilégio se dá através de alguns bits nos chamados registradores de segmento da CPU, que apontam para uma determinada entrada na GDT. A CPU sabe o nível de privilegio do código, graças aos dados contidos na GDT.

Cada uma das entradas da GDT possui 64 *bits* e são dispostas na memória de acordo com as especificações da mostradas na figura 6. É possível notar a existência de dois campos principais: o *base* e o *limit*. O campo *base* indica qual é o primeiro endereço do segmento, e o *limit* indica a sua posição final. Além disso, o campo *Access Byte* indica se o descritor é de um segmento de código, ou se é um segmento de dados, e também, qual é o nível de acesso do segmento.

O sistema FESO cria na memória uma GDT com 6 descritores de segmento. Como são utilizados 3 níveis de privilégio (níveis 0, 1 e 3), são criados dois descritores para cada um: o primeiro é o descritor de segmento de código, e o outro, de dados. Uma vez que a segmentação não é utilizada, o campo *base* fica em 0 e campo *limit* é colocado em 4 GB para

todos os descritores, fazendo com que o processador veja a memória como um único segmento.

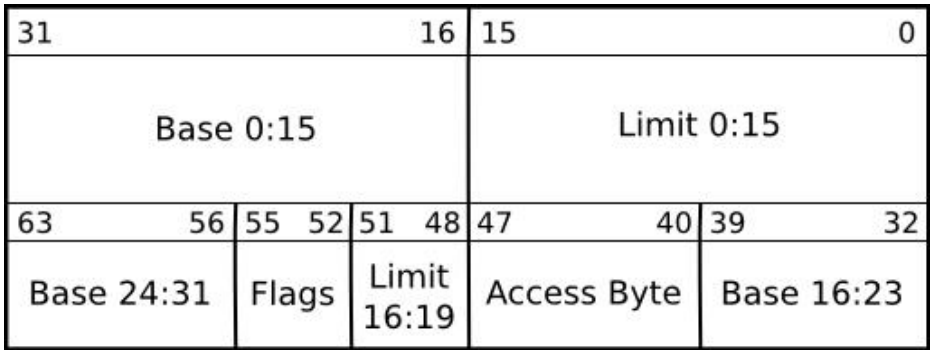

Figura 6 - Estrutura do descritor da GDT (OSDEV WIKI, 2012)

Cada descritor de 64 *bits* fica disposto em sequência na memória. Além dos descritores, outra estrutura é criada, e nela é colocado um ponteiro para o começo do primeiro descritor, e a quantidade de *bytes* ocupados pela GDT. Após isso, uma instrução especial da CPU, a LGDT, é chamada, tendo como operando o endereço dessa estrutura. Essa instrução carrega um registrador especial da CPU, responsável por armazenar a posição da GDT atualmente em uso.

Estando a GDT devidamente instalada, é possível utilizar os registradores de segmento para acessar blocos de memória descritos por ela. Alguns *bits* dos registradores de segmento apontam para uma entrada da GDT. Dessa maneira, é possível especificar diversos segmentos na GDT, e utilizá-los através da configuração correta dos registradores de segmento. O sistema FESO não faz uso da segmentação. Desta forma, as entradas na GDT são utilizadas apenas para o controle dos níveis de privilégio. Os registradores de segmento são configurados de acordo com o nível de privilégio do código em execução. Quando as rotinas do núcleo estão em execução, os registradores de segmentos apontam para as entradas da GDT correspondentes ao nível de privilégio máximo (nível 0), e o mesmo ocorre para *drivers* de dispositivos e aplicativos de usuário com seus respectivos níveis de acesso.

A CPU verifica se a próxima instrução a ser executada está acordo com o nível de privilégio descrito pela entrada na GDT especificada nos registradores de segmento. Caso a instrução exija um nível de acesso maior do que o nível corrente, uma exceção é gerada pela CPU, e o sistema operacional tem condições de interromper o programa que está tentado realizar uma tarefa para a qual ele não possui permissão.

Obviamente, as instruções para alterar a GDT, e para alterar o valor do nível de privilégio do código atual são instruções privilegiadas que só podem ser executadas pelo sistema operacional. Isto impede que um programa altere seu nível de privilégio e passe a controlar o processador. Além disso, para permitir que o núcleo do sistema execute com o nível de privilégio máximo, sempre que uma interrupção ocorre, uma rotina do sistema operacional é chamada automaticamente pelo processador. Essa rotina é executada com privilégio máximo, pois o próprio processador se encarrega de mudar o nível de privilégio antes da chamada de uma das rotinas do núcleo. Cabe ao sistema operacional alterar o nível de privilégio do código novamente, antes de retornar o controle ao programa previamente interrompido.

A segunda tabela de descritores está intimamente ligada ao processo de tratamento de interrupções. A IDT (*Interrupt Descriptors Table*) armazena os descritores de porta (*gate descriptors*) que fornecem o acesso aos manipuladores de exceções e interrupções (INTEL, 2011). Esses manipuladores nada mais são do que rotinas chamadas para o tratamento de eventos. Durante o processo de inicialização, o sistema FESO cria uma IDT cujos descritores apontam para cada uma de suas funções de tratamento. Desta forma, sempre que um evento ocorre, é uma rotina do sistema FESO que é posta em execução para tratá-la.

A figura 7 mostra como é a estrutura de um descritor da GDT. Cada descritor também possui 64 *bits*, o campo *offset in segment* possui 32 *bits* ao todo, nele fica armazenado o endereço da rotina de tratamento da interrupção. O campo *segment selector*

aponta para um índice da GDT responsável por especificar em qual segmento o código de tratamento da interrupção está. Todas as interrupções tem seu segmento configurado para a entrada da GDT que especifica o segmento de código de nível de privilégio máximo. Deste modo, sempre que uma interrupção for chamada, a CPU automaticamente altera o registrador de segmento fazendo com que a rotina de tratamento tenha privilégio máximo.

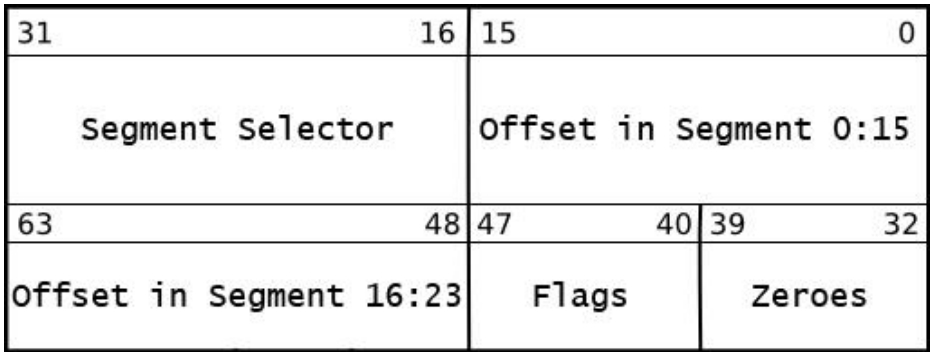

Figura 7 - Estrutura do descritor da IDT

De forma muito similar a GDT, os descritores da IDT são armazenados na memória em sequência. Uma estrutura com um ponteiro para o primeiro descritor e com tamanho total da IDT é criada, e uma instrução, a LIDT, é chamada para configurar o registrador responsável por armazenar a posição da IDT. Esta também é uma instrução privilegiada, e apenas o SO tem condições de instalar suas próprias rotinas de tratamento de interrupções.

Cada interrupção possui um número, que é utilizado como índice dentro da IDT para a definição, por parte da CPU, de qual rotina deve ser chamada para o que ela seja tratada. A IDT possui 256 entradas, tanto para interrupções de *hardware, software* ou exceções. As faixas de valores para cada tipo serão tratadas em uma seção a parte.

3.3.3 Instalação das Interrupções

O sistema operacional FESO instala as rotinas de tratamento de interrupções durante o processo de inicialização. O sistema lida com 3 tipos distintos de interrupções: as

interrupções de *hardware* geradas pelos dispositivos de E/S. As interrupções de *software* geradas pelos programas quando estes precisam de algum serviço do SO, e finalmente, as exceções que ocorrem quando algum programa tem algum comportamento indevido, tal como executar uma instrução para a qual ele não tem permissão.

As primeiras 32 entradas da IDT, ou seja, as interrupções cujos índices vão de 0 até 31, são exceções. As exceções são geradas pela própria CPU, e cabe ao sistema operacional tomar a decisão de qual ação deverá ser tomada para tratá-las. Por padrão, o sistema FESO finaliza o processo em execução sempre que uma exceção é gerada. Após isso o algoritmo de escalonamento é executado, e um novo processo é escalonado.

As interrupções de *hardware* são tratadas primeiramente por um *chip* presente nos computadores, o PIC (*Programmable Interrupt Controller*). O PIC gerencia diversos pedidos de interrupção e os repassa ao processador dependendo de sua prioridade (MESSMER, 1999). O PIC possui apenas 8 entradas, porém, como podem existir mais fontes de interrupção, outro PIC é colocado em cascata na entrada 2 do primeiro. O primeiro PIC é conhecido como *master*, e o segundo, ligado a entrada dois, é o *slave.*

O PIC possui internamente uma tabela na qual é possível configurar qual índice de vetor será enviado ao processador na ocorrência de uma interrupção. É possível configurar essa tabela através da escrita e leitura de portas de E/S especificas para cada um dos PICs. O sistema FESO configura os vetores das interrupções de *hardware* a partir do número 32. Isto porque os 32 primeiros vetores estão reservados para as exceções. Após a chamada de uma função para o tratamento de uma interrupção de *hardware*, o PIC deve ser sinalizado. O PIC só enviará o próximo sinal de interrupção, quando o último sinal enviado tiver sido tratado pela CPU. A sinalização também é feita através do uso de portas de E/S e instruções especiais da CPU.

Finalmente, a entrada 80 da IDT é configurada com a função de tratamento de interrupções de *software* ou *system calls* (chamada ao sistema). A rotina de inicialização do FESO instala uma função de tratamento padrão na entrada correspondente da IDT. Os programas de usuário e *drivers* de dispositivo podem gerar uma interrupção chamando uma instrução em *assembly*, a INT. O operando da instrução é o número 80, o que faz o processador buscar a entrada correspondente na IDT e coloca-la em execução. Essa função verifica o estado dos registradores, que foram previamente configurados pelo programa que gerou a interrupção, e desta maneira o sistema operacional sabe como proceder para atender a chamada.

Todas as interrupções devem estar devidamente configuradas antes que o sistema operacional carregue na memória e entregue o controle da CPU a algum programa. Uma vez que todas as interrupções estão devidamente configuradas, existe a garantia de que o sistema operacional irá entrar em execução sempre que um evento ocorrer, podendo assim, manter seu controle sobre o sistema computacional.

3.3.4 Detecção e Inicialização de Múltiplos Processadores

O sistema FESO tem suporte a processadores *multicore*. Um computador *multicore* combina dois ou mais processadores em uma única peça de silício (STALLINGS, 2010). Este tipo de sistema permite que mais de um programa seja executado simultaneamente em um computador, ou mesmo, que um programa seja dividido em partes e executado por diversas CPUs em simultâneo.

Durante o processo de inicialização, o SO faz a detecção da existência de múltiplos processadores através das tabelas ACPI encontradas na memória. O padrão ACPI (*Advanced Configuration and Power Intefacer*) envolve, entre outras funções, o fornecimento das *Multiprocessor Specification Tables* (tabelas para especificação de múltiplos processadores) (ACPI 2011). Através da busca a análise dessas tabelas, o sistema operacional

consegue descobrir quantos processadores existem no sistema, e como eles podem ser acessados.

Uma vez que o SO tem acessos às informações contidas nas tabelas, ele pode inicializar os processadores através do envio de IPIs ou interrupções entre processadores. Sempre que um computador com múltiplos processadores é iniciado, um dos processadores é selecionado para inicializar o computador, este processador é chamado de BSP (*boot strap processor*), enquanto os outros processadores são chamados de AP (*application processors*). As IPIs são enviadas a partir do BSP para realizar a inicialização de todos os APs presentes no sistema.

O sistema operacional carrega previamente em um espaço no inicio da memória um pequeno programa responsável por configurar corretamente os processadores. É o chamado código trampolim. O endereço do ponto de entrada desse programa é enviado junto com as IPIs para cada um dos APs. Ao receber a IPI o *application processor* executa a partir da posição recebida junto com a interrupção, ficando corretamente configurado após a execução do código trampolim. Em seguida, os APs entram em estado de espera por uma interrupção, para só então começarem a executar algum programa.

Enquanto processadores com apenas um núcleo possuem um par de PICs ligados a eles, sistemas com múltiplos núcleos precisam de outro tipo de *chip* para funcionar corretamente. Este chip é o *Advanced Programmable Interrupt Controller* (APIC). Sistema com múltiplos núcleos possuem uma estrutura como mostrada na figura 8.

Conectado a cada processador existe um *Local* APIC ou LAPIC responsável por receber e enviar as interrupções. Cada *Local* APIC está conectado ao barramento ICC, bem como o I/O APIC. O I/O APIC é responsável por receber os sinais das interrupções de *hardware* e repassá-los aos outro LAPICs, permitindo que mais de um processador possa tratar um pedido de interrupção de *hardware.*

Cada LAPIC possui um TIMER, que pode ser configurado para disparar uma interrupção a cada intervalo de tempo predefinido. Esse TIMER é configurando pelo sistema operacional, para que cada núcleo interrompa a execução do processo em execução e permita que um novo processo tenha sua fatia de tempo na CPU.

Como cada processador pode estar executando um programa diferente, e como cada um deles tem seu próprio TIMER, gerando interrupções a todo o momento, é altamente provável que dois ou mais processadores tentem acessar as rotinas do *kernel* simultaneamente. Caso dois ou mais processadores estejam executando o código do núcleo inadvertidamente ao mesmo tempo, é possível que ambos acessem alguma variável compartilhada e que acabem por deixar as estruturas do núcleo em um estado inconsistente.

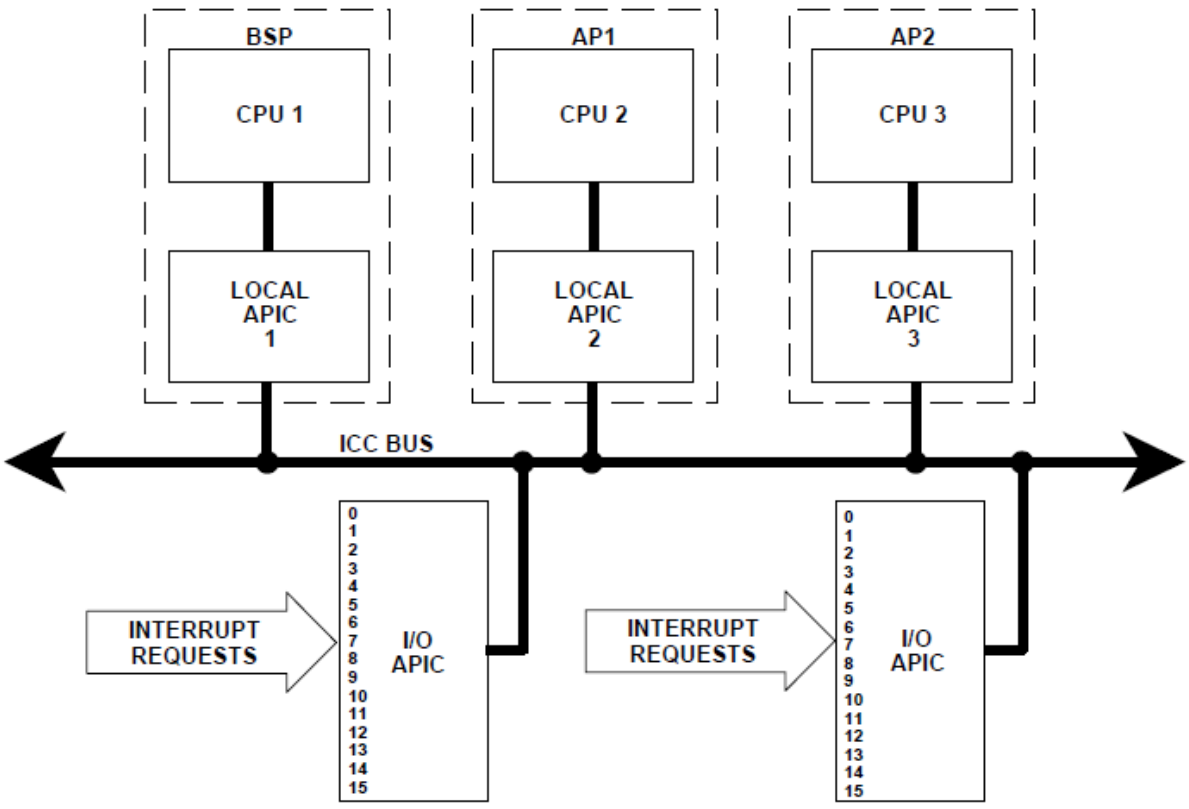

Figura 8 - Sistema com múltiplos processadores (INTEL, 1997)

Para evitar esse problema, o núcleo do sistema FESO tem uma grande trava. Essa trava é implementada através de uma variável que é lida por todos os processadores antes que

eles comecem a executar alguma rotina do núcleo. Uma variável de trava que usa a espera ocupada é chamada de trava giratória (s*pin lock*) (TANENBAUM, 2010, p. 73). Caso o valor da variável esteja em 1, significa que há um outro processador executando o código do núcleo, e que é necessário aguardar. Os processadores ficam em um laço de repetição testando o valor da variável de trava, até que este esteja em 0. Quando isso ocorre, o processador altera o valor da variável para 1, e passa a executar o código do núcleo. Após terminar de executar o código do núcleo, o processador coloca novamente o valor da trava em 0.

A operação de leitura e atualização da variável de trava deve ser indivisível. Isto é, enquanto um processador estiver lendo e possivelmente atualizando a variável de trava, os outros processadores não podem ter acesso a ela. Caso fosse possível que dois processadores lessem o valor 0 na variável de trava e tentassem atualizá-la ao mesmo tempo, ambos começariam a executar o código do núcleo em simultâneo, o que não é desejável.

Para evitar esse cenário, o *hardware* do processador oferece a possibilidade de se utilizar uma instrução que realizar o teste e atualização da variável de trava de maneira atômica. Os processadores da família INTEL de 32 bits suportam operações atômicas em locais da memória do sistema (INTEL, 2011). Essas instruções são utilizadas para garantir a atomicidade das operações que fazem o acesso a variável de trava. A instrução recebe um prefixo *lock*, que faz com que o barramento da memória, que é compartilhado pelos processadores, seja travado, garantindo que apenas um processador acesse a variável em um dado instante de tempo.

3.4 Gerência de Memória

A gerência da memória do FESO funciona em diversos níveis. No nível mais baixo, existe a gerência da memória física, no qual é feito o controle de quais blocos da memória estão disponíveis para a alocação. Sobre a gerência da memória física, existe o

controle da memória virtual, responsável por mapear os endereços dos *frames* (quadros) de memória física, em endereços virtuais. E ainda acima desta, existem os algoritmos de alocação de memória, que se utilizam de uma chamada ao sistema para solicitar mais memória ao SO, e controlam blocos de memória de tamanhos arbitrários. Esta seção trata do funcionamento de cada um desses níveis.

3.4.1 Gerência de Memória Física

A gerência de memória física funciona da seguinte maneira: após o processo de inicialização do sistema, uma área de memória, localizada após a imagem do sistema operacional é reservada. Está área, cujo tamanho depende do tamanho da memória real instalada no sistema, é utilizada para armazenar um mapa de *bits* utilizado no controle da memória física. Cada *bit* no mapa representa uma área de memória de 4KB como ilustra a figura 9. Caso o estado do bit em uma determinada posição esteja em 1, o *frame* está ocupado, caso o *bit* esteja em 0, o *frame* está vazio.

A memória disponível para utilização começa na primeira posição alinhada em 4KB após o mapa de *bits.* Isto porque, o uso da paginação nos processadores da família Intel requer endereços alinhados em 4KB, ou seja, com os 12 *bits* menos significativos iguais a 0. Como cada bloco tem 4KB, todos eles começam em um endereço de memória alinhado em 4KB. Sempre que é necessária a alocação de mais memória, ocorre uma busca no mapa de *bits* iniciada geralmente pelo último endereço retornado pelo alocador. A busca é feita usando um algoritmo *first-fit*, de maneira que o endereço do primeiro conjunto de blocos contíguos do tamanho solicitado ao alocador é retornado. É possível saber o endereço inicial do *frame* com base na posição do *bit* dentro do mapa.

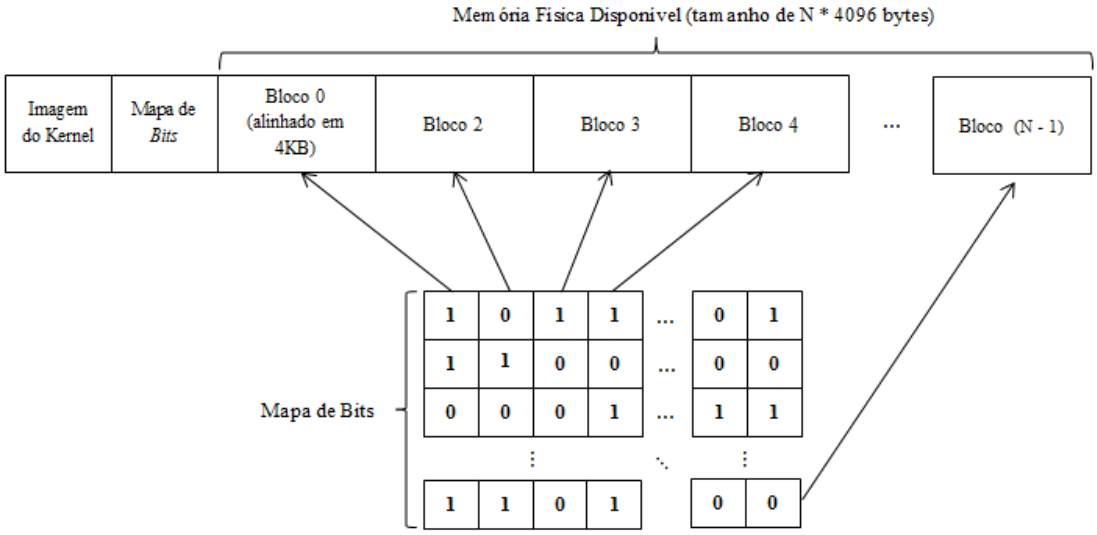

Figura 9 - *Bitmap* para controle da memória física

3.4.2 Gerência de Memória Virtual

Em outro nível, a utilização da memória virtual permite a criação de um espaço de endereçamento virtual composto por um conjunto de endereços virtuais. Esses endereços virtuais são mapeados em endereços físicos, de forma a dar a impressão aos programas de usuário que eles possuem toda a memória da máquina apenas para eles.

O *loader* do sistema FESO é muito simples, e não suporta a relocação de programas. Entretanto, a utilização de espaços de endereçamento virtuais permite que diversos programas utilizem o mesmo conjunto de endereços virtuais, que são mapeados em *frames* de memória física distintos para cada um deles. Como existe um espaço de endereçamento personalizado para cada aplicativo de usuário ou *driver* de dispositivo, não é preciso realizar a relocação dos endereços para fazer com que todos esses aplicativos ou *drivers* possam coexistir na memória.

O espaço de endereçamento virtual nos processadores da família Intel de 32 bits é de 2^{32}, ou seja, 4 GB. O *kernel* fica mapeado da mesma forma nos endereços do último *gigabyte* de todos os espaços de endereçamento. Isto é necessário, pois as rotinas do *kernel* devem estar

sempre acessíveis, não importando qual programa está em execução. Resta a cada programa de usuário um espaço de endereçamento de 3GB.

À medida que programas de usuário (que ocupam quantidades diferentes de memória e são geralmente carregados em posições adjacentes) são iniciados e finalizados, surgem na memória áreas fragmentadas, que podem dificultar o carregamento de novos programas maiores. Este problema é conhecido como fragmentação externa, e também é resolvido com o uso da paginação. Uma vez que a paginação permite que *frames* não contíguos de memória física possam ser mapeados em páginas contiguas no espaço de memória virtual, não importa em quais frames de memória física os programas são carregados, pois, eles são mapeados em páginas virtuais adjacentes.

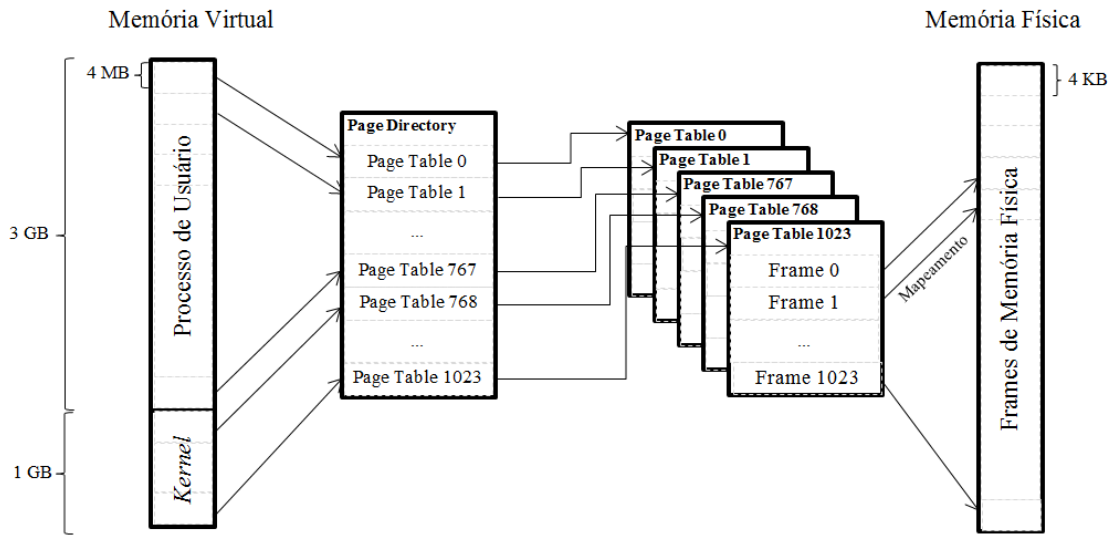

Figura 10 - Paginação em Processadores da Família Intel

Nos processadores da família Intel o mapeamento de endereços virtuais em físicos é feito através do uso de dois níveis de tabelas de páginas, como demonstra a figura 10. No primeiro nível existe a *Page Directory*, que possui 1024 entradas. Cada uma dessas entradas aponta para uma *Page Table.* As *Page Tables*, por sua vez, também possuem 1024 entradas, e cada uma delas aponta para um bloco de 4KB de memória física. Dessa forma, o mapeamento

de endereços virtuais em físicos é feito da seguinte forma: os 10 primeiros *bits* mais significativos do endereço virtuais de 32 *bits* representam um índice na *Page Directory*. Os 10 *bits* anteriores a estes representam um índice na *page table*, e os 12 *bits* menos significativos representam o deslocamento dentro do *frame* de memória.

Uma limitação da gerência de memória do sistema FESO é a não utilização de técnicas para troca entre programas, ou partes de programas, da memória para alguma forma de armazenamento secundário ou vice-versa. Caso não haja memória física disponível para um programa, este simplesmente não é executado pelo sistema.

3.4.3 Alocação de Memória

Diversos programas de computador são criados de maneira que a quantidade total de memória necessária em sua execução não é definida durante seu desenvolvimento. Isto é, não há como saber exatamente quanta memória o programa irá precisar, ou mesmo a quantidade de memória pode variar a cada execução.

Neste caso, os programas se utilizam da alocação dinâmica de memória, que consiste na expansão da memória utilizada por um programa na medida em que esta é necessária. O sistema operacional deve controlar quanta memória cada programa está utilizando, e permitir que mais memória seja alocada durante sua execução.

A alocação dinâmica é utilizada tanto pelo núcleo e *drivers* de dispositivos, quanto pelos aplicativos de usuário. Cada programa possui uma área reservada em seu espaço de endereços para a alocação dinâmica. Está área está é conhecido como *heap* e está também presente no espaço de endereçamento do núcleo.

Na linguagem C, as funções *malloc e* free são as principais responsáveis pela alocação e liberação de blocos de memória. O sistema operacional oferece uma estrutura base na qual essas funções podem operar. Cada processo, assim como o *kernel*, possui variáveis para o controle do *heap.* Sempre que a função *malloc* (ou sua versão implementada dentro do

kernel, a *kmalloc*) necessita de mais memória, é preciso aumentar o *heap*, e retornar o endereço do primeiro *byte* de memória da área recém-expandida.

No caso do *kernel*, as rotinas da gerência de memória virtual alocam um novo bloco de memória física e aumentam a área do *heap*. Os programas, por sua vez, fazem uma chamada ao sistema operacional, solicitando uma nova área de memória. Cada processo de usuário possui suas próprias variáveis para controle de quanta memória está sendo utilizada.

A função *malloc* utiliza uma lista encadeada com blocos livres de memória como é possível ver na figura 11. Sempre que a função é chamada, a lista é percorrida, e o primeiro bloco com tamanho igual ou maior ao solicitado que for encontrado é alocado, e seu endereço é retornado.

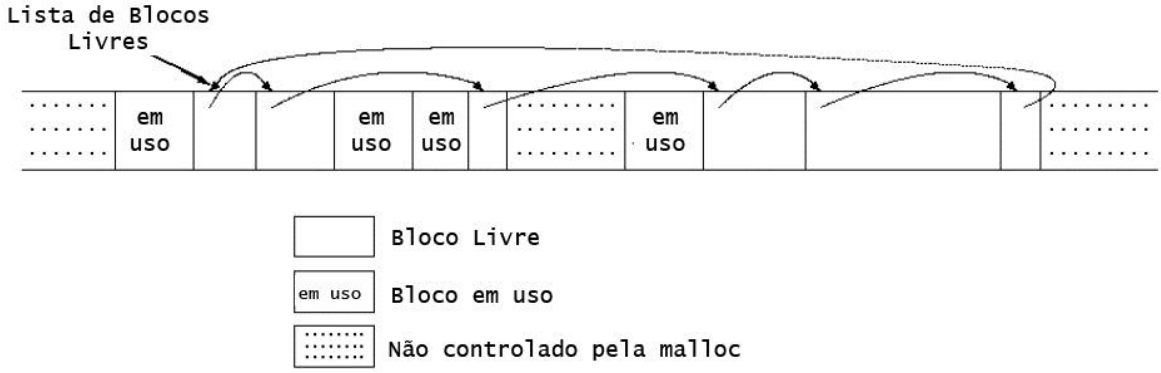

Figura 11 - Lista de blocos livres (KERNINGHAN & RITCHIE, 1988, p. 150)

Cada bloco possui em seu começo um pequeno cabeçalho, no qual existe um ponteiro para o próximo bloco livre, e o tamanho do bloco atual, como mostra a figura 12. Quando parte de um bloco é alocada, este é dividido em dois, e um novo cabeçalho e criado, com o tamanho do bloco. Este novo bloco não está ligado à lista encadeada, pois não está livre, porém seu cabeçalho é criado para que seja possível determinar seu tamanho.

A função *free* recebe como parâmetro o endereço inicial do bloco que deve ser liberado. Quando isto ocorre, a função procura o cabeçalho que deve existir antes do endereço

passado como parâmetro do para saber o tamanho do bloco. Então, a função varre a lista de blocos livres para saber em que posição da lista encadeada o novo bloco deve ser colocado. Caso o bloco seja adjacente a outros blocos livres, eles devem ser unidos em um novo bloco de tamanho maior.

As funções *malloc* e *free* funcionam em espaço de usuário. Desta maneira, a chamada ao sistema para a alocação de mais memória no *heap* ocorre apenas quando toda a lista encadeada é percorrida e um nenhum bloco livre de tamanho adequado é encontrado. Neste caso, o programa solicita um novo bloco de memória ao sistema operacional, e este é adicionado à lista de blocos livres, gerenciada pela função *malloc*.

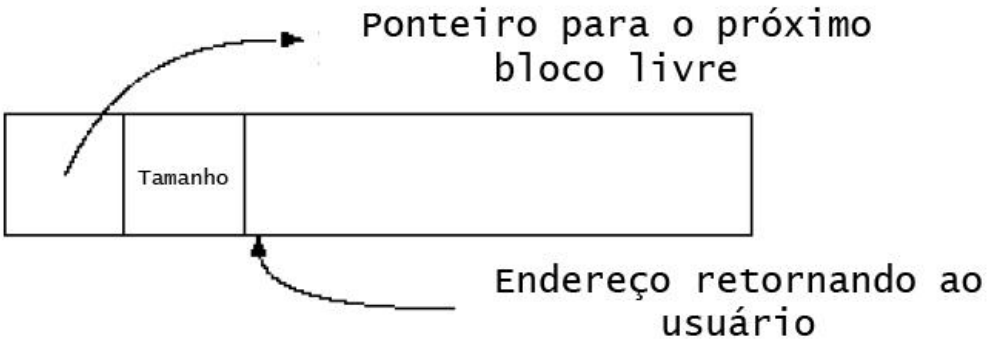

Figura 12 - Cabeçalho do Bloco (KERNINGHAN & RITCHIE, 1988, p. 151)

3.5 Gerência de Processos

O sistema de gerência de processos é o componente do SO responsável pela criação, eliminação e escalonamento dos processos e *threads*. Nesta seção são tratados os detalhes do funcionamento da gerência de processos do sistema FESO.

3.5.1 Processos e Threads

Processos são abstrações utilizadas para o controle da execução dos programas pelo SO. Para cada processo presente no sistema FESO, diversos dados precisam ser

armazenados. Entre eles, o estado dos registradores da CPU, os blocos de memória física alocados e outras informações utilizadas no controle do processo.

O sistema FESO possui um suporte rudimentar a *threads*. Os *threads* são unidades básicas de alocação da CPU (SILBERCHARTZ, 2000, p.82). Os processos possuem uma lista de *threads* associados, cada *thread* possui sua própria estrutura com o conjunto de estados dos registradores. Os *threads* em um processo podem estar executando trechos de código diferentes entre si, porém todos compartilham o mesmo espaço de endereçamento e mesmas variáveis globais. Entretanto, existe uma pilha associada a cada *thread*, na qual são armazenadas as variáveis locais e as estruturas de controle a chamada de sub-rotinas.

A figura 13 mostra como está dividido o espaço de endereçamento virtual de um processo. Os programas desenvolvidos para funcionar sobre o FESO precisam possuir seus endereços resolvidos a partir da posição 0 da memória. No começo do espaço de endereçamento, ficam o código e os dados do programa.

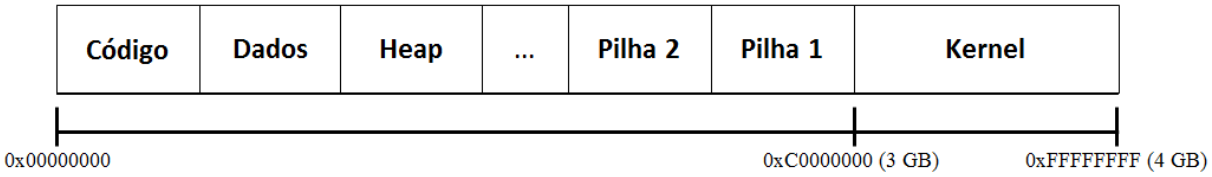

Figura 13 - Espaço de Endereçamento de um Processo

Imediatamente após os dados, começa o *heap*, posição onde ficam mapeados os *frames* de memória alocados dinamicamente pelo programa. A região de memória entre o *heap* e as pilhas é reservada para o crescimento da quantidade de pilhas, que é feita em posições de memória menores, e para a expansão do *heap* que é feita para posições de memória maiores.

No final do espaço de endereçamento de um processo, ficam as pilhas dos *threads*. Ao contrário do *heap* que é único para todos os *threads* de um mesmo processo, as pilhas são individuais. O tamanho da pilha é fixo, e não é alterado durante a execução de um programa. Quando um *thread* é finalizado, o espaço de sua pilha não e desalocado, pois ele poderá ser utilizado como pilha de um novo *thread* a ser criado no futuro.

O núcleo do sistema armazena os processos e *threads* em árvores, ordenados pelos identificadores. Cada processo possui um identificador, e cada *thread* também. Assim que são criados, os processos possuem um *thread* principal, que possui o mesmo identificador do processo ao qual está atrelado. O sistema FESO armazena em duas estruturas básicas, uma para processos e outra *threads*, algumas informações utilizadas no controle do uso da CPU e dos recursos computacionais.

Para os processos, o SO precisa armazenar uma lista com todos os *frames* de memória ocupados. Isto é necessário para que eles possam ser liberados quando o processo terminar. O sistema operacional também armazena o endereço da *Page Directory*, que é fundamental na troca de contexto. Além disso, se faz necessário o armazenamento de algumas variáveis para o controle do *heap*, um *flag* para indicar o nível de privilégio do processo, uma lista com pilhas alocadas pelos *threads*, um ponteiro para a próxima posição livre na área reservada a pilhas, entre outras informações.

As estruturas que armazenam *threads* são um pouco diferentes. Elas possuem campos para armazenar o estado dos registradores da CPU, o estado de execução em que o *thread* se encontra (pronto para a execução, em espera, ou em execução) e seu nível de privilégio. Os *threads* possuem também uma lista de mensagens, que é utilizada pelo sistema de comunicação entre processos do FESO.

Diversas operações são realizadas sobre os processos e *threads*. Processos e *threads* podem ser criados ou destruídos pelo sistema a qualquer momento. O sistema

operacional fornece um conjunto mínimo de chamadas ao sistema que permitem a realização dessas operações. A criação de processos é feita através do carregamento de arquivos executáveis no formato ELF (*Executable and Linkable Format*). Porém, o sistema FESO ainda não suporta o carregamento de bibliotecas compartilhadas, então é imperativo que o arquivo executável esteja com todas as suas bibliotecas estaticamente *linkadas* em um único arquivo binário.

3.5.2 Algoritmo de Escalonamento

O número total de *threads* no sistema, na maioria dos casos, é maior do que a quantidade de CPUs disponíveis. Por este motivo, o sistema operacional deve permitir que cada *thread* tenha uma fatia de tempo para ser executado pela CPU. O sistema FESO se utiliza de um algoritmo de escalonamento por chaveamento circular, conhecido como *round-robin* para escalonar as CPUs para os diversos *threads* presentes no sistema. A figura 14 mostra um exemplo de execução o algoritmo de escalonamento, bem como um diagrama das possíveis trocas de estado entre *threads.*

Existem basicamente 2 tipos de threads presentes no sistema FESO, os *threads* de aplicativos de usuário (com prioridade 1) e os threads dos processos servidores (prioridade 0). O algoritmo de escalonamento do FESO utiliza 2 filas de *threads* com prioridades diferentes, e o algoritmo *round-robin* é executado nessas duas filas. O algoritmo de escalonamento verifica, inicialmente, na primeira fila, de prioridade 0, se existe algum *thread* pronto para a execução. E caso exista ele é escalonado. Caso não exista, a mesma verificação é executada na segunda fila.

O algoritmo de escalonamento é executado constantemente, seja por causa de uma interrupção causada pelos temporizadores do sistema, ou porque algum *thread* executou uma chamada bloqueante. Em ambos os casos, o processo tem sua execução interrompida, e o estado de seus registradores salvo. A diferença é que *threads* cuja fatia de tempo se encerrou

ainda estão prontos para execução, e só não estão alocados pois o sistema decidiu permitir que outro *thread* executasse. O caso do *thread* que realizou uma chamada bloqueante é diferente, pois a continuação de sua execução só será possível, depois da ocorrência do evento pelo qual ele está esperando. O algoritmo de escalonamento sempre armazena o estado de todos os registradores da CPU no momento em que um *thread* é interrompido, e quando um *thread* é posto de volta em execução, o estado de seus registradores é restaurado. Este processo é chamado de troca de contexto, e permite que a execução de um *thread* seja retomada exatamente de onde parou.

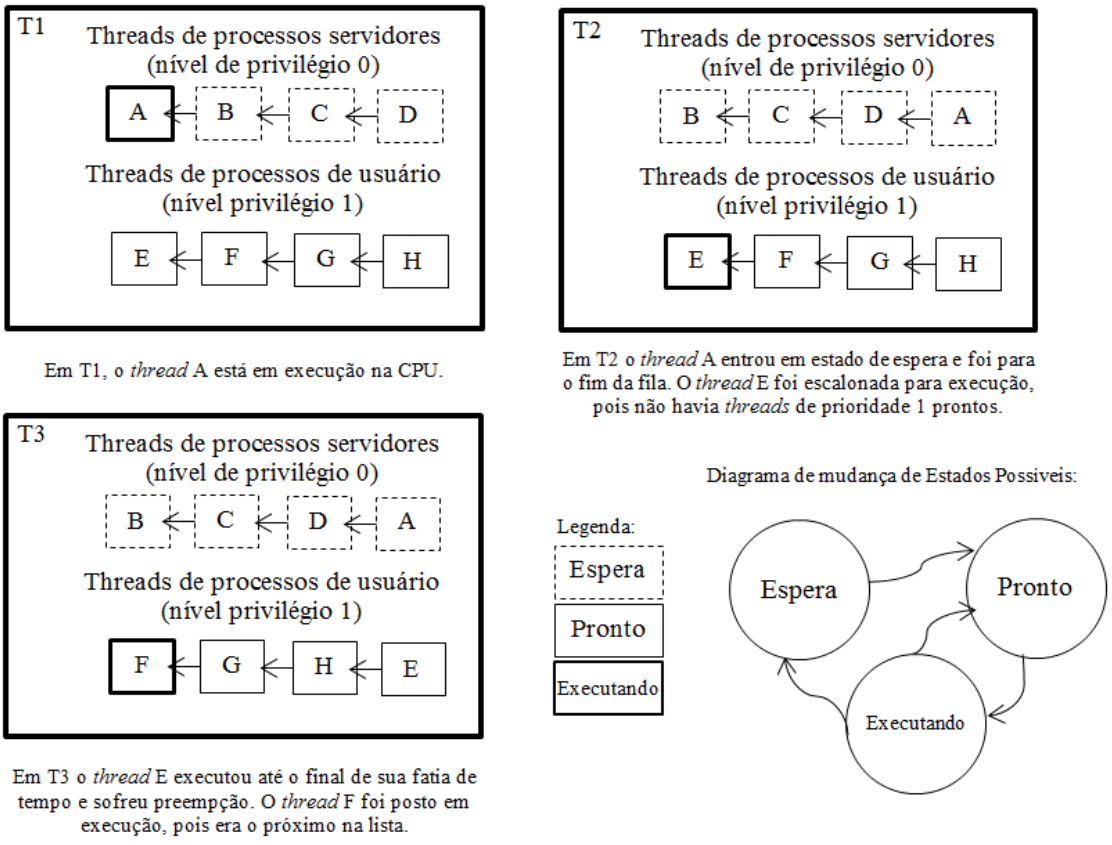

Figura 14 - Exemplo de escalonamento e diagrama de troca de estados

O algoritmo de escalonamento coloca o *thread* em execução no final de sua fila, sempre que ele sofre preempção, ou seja, tem sua execução interrompida, ou executa uma chamada bloqueante. Desta maneira, todos os *threads* tem a chance de obter uma fatia de tempo para execução na CPU. Entretanto, como os *threads* de processo servidores têm uma

prioridade maior que *threads* de processos de usuário, caso um *thread* servidor nunca entre em estado de espera, qualquer *thread* de usuário fica indefinidamente impedido de executar. Para evitar que isto ocorra, os processos servidores, que só podem ser criados durante a inicialização do sistema, devem ser desenvolvidos de forma a não ocuparem a CPU indefinidamente.

Há ainda, um *thread* especial, criado pelo SO durante o processo de inicialização. É o *thread IDLE*, ou ocioso. Este *thread* é escalonado sempre que não existe nenhum outro *thread* pronto para a execução no sistema. Este *thread* simplesmente coloca a CPU em um estado espera, até que uma interrupção ocorra e o algoritmo de escalonamento seja novamente executado.

3.5.3 Carregamento de executáveis ELF

O formato ELF (*Executable and Linking Format*) foi originalmente desenvolvido e publicado pelo UNIX *Systems Laboratories* como parte do *Application Binary Interface.* (TIS, 1993). O ELF foi o formato de arquivo executável escolhido para a criação dos executáveis do sistema FESO. Tanto o núcleo do sistema, quanto os *drivers* de dispositivos e aplicativos foram desenvolvidos nesse formato.

O sistema FESO possui suporte a programas criados no formato ELF que tenham sido especialmente compilados por um compilador cruzado (*cross-compiler*). Os executáveis do sistema FESO devem usar apenas as bibliotecas e recursos da linguagem para as quais existe suporte. Outra restrição, é que os aplicativos devem estar estaticamente *linkados*, isto é, todas as bibliotecas utilizadas pelo programa devem estar no próprio arquivo. O sistema operacional ainda não suporta o uso de bibliotecas compartilhadas.

A figura 15 mostra a estrutura de um arquivo ELF. O processo de carregamento começa com a leitura do arquivo executável, e a criação de sua imagem na memória do

computador. Em seguida, as estruturas contidas no arquivo são analisadas, para que seja possível posicionar o código executável corretamente na memória.

A primeira estrutura analisada é o cabeçalho (ELF *header*) posicionado no começo do arquivo. Neste cabeçalho, encontram-se, entre outras informações, o ponto de entrada do arquivo executável, que nada mais é do que o endereço da primeira instrução do programa. No cabeçalho existe ainda a posição inicial de outra estrutura, a tabela de cabeçalho de programa (*Program Header Table*).

A tabela de cabeçalho de programa contém uma lista de estruturas que especifica cada um dos segmentos (*segments*) do arquivo. Os segmentos são partes do programa executável, que contém o código e os dados. O algoritmo de carregamento de programas percorre a tabela, e analisa cada uma de suas entradas. A estrutura de uma entrada dessa tabela contém a posição do segmento dentro do arquivo executável, o tipo do segmento e em qual endereço de memória ele deve ser carregado.

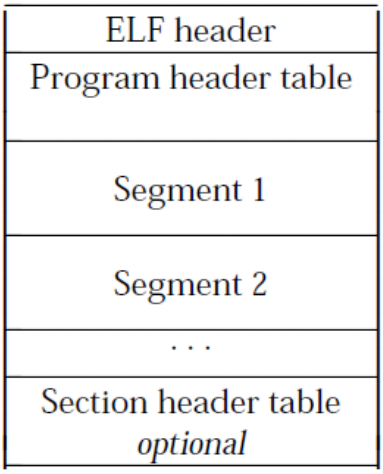

Figura 15 - Estrutura de um arquivo ELF (TIS, 1993)

A partir dessas informações, o algoritmo de carregamento é capaz de criar um novo espaço de endereçamento para o novo processo, alocando *frames* de memória física, os mapeando neste novo espaço e carregando os segmentos do arquivo para as suas devidas

posições. Uma vez que as seções estejam copiadas, uma nova estrutura de processo e outra de *thread* são criadas e adicionadas as suas respectivas árvores.

Os estados dos registradores dentro da estrutura do novo *thread* são configurados, de forma que o EIP apontará para instrução indicada como o ponto de entrada do arquivo. Os registradores da pilha são ajustados para as posições finais do espaço de endereçamento. Os *frames* das pilhas também precisam ser alocados, e mapeados em suas posições corretas no espaço de endereçamento. A pilha é ajustada para receber os parâmetros passados ao programa, quando estes existem, e o *heap* do novo processo é devidamente inicializado, ficando pronto para ser expandido à medida que necessário.

Quando todo esse processo é completado, o novo *thread* é adicionado à lista de *threads* do algoritmo de escalonamento. No momento de sua primeira execução, o algoritmo de escalonamento simplesmente coloca os valores armazenados nas estruturas nos registradores da CPU, e altera o espaço de endereçamento (configurando um registrador da CPU, com o endereço da *Page Directory* recém-criada). Finalmente, o novo programa estará em execução, começando pelo seu ponto de entrada especificado nas estruturas do arquivo ELF.

3.6 Comunicação entre processos

O método de comunicação entre processos, chamado de troca de mensagens, utiliza duas primitivas, *send* e *receive* (enviar e receber) (TANEMBAUM & WOODHULL, 2000). O sistema FESO possui chamadas ao sistema para possibilitar o uso dessas primitivas, e consequentemente permitir que os processos e *threads* troquem informações e também possam ser sincronizados.

3.6.1 Portas

O sistema FESO utiliza o conceito de portas. As portas possuem um número e funcionam de forma análoga a uma caixa postal. Um *thread* pode alocar uma porta e desta maneira, sempre que uma mensagem for enviada para a porta alocada, será redirecionada para o *thread* que a alocou. Com isso, um *thread* de um processo servidor pode alocar sempre as mesmas portas, que são previamente conhecidas pelos seus clientes, e que sempre envião suas solicitações para ela.

Os *threads* também utilizam portas para serem sinalizados no caso da ocorrência de algum evento no sistema. Por exemplo, as 20 primeiras portas estão relacionadas às IRQs (*interrupt request*), ou seja, aos sinais enviados ao processador pelo hardware para relatar o acontecimento de eventos. Sempre que uma IRQ ocorre, o sistema operacional verifica se a porta referente aquele IRQ está alocada a algum *thread*. Se uma IRQ ocorrer, uma mensagem é enviada ao *thread* informando a ocorrência da interrupção. Está é a maneira que os processos servidores para drivers de dispositivos têm de serem sinalizados sempre que uma interrupção for disparada.

As portas também são utilizadas para que os processos possam ser avisados de eventos que ocorrem dentro do *núcleo* do sistema. Algumas portas são reservadas exclusivamente para esses eventos, e sempre que eles ocorrem, uma mensagem é encaminhada para avisar os *threads.* A tabela 2 mostra uma lista de portas, e seus eventos correspondentes.

Em um dado instante de tempo, apenas um *thread* pode ter uma determinada porta alocada. A única exceção são as portas de eventos do *kernel,* que permitem que diversos *threads* a aloquem ao mesmo tempo. As primeiras 1000 portas estão reservadas para *drivers* de dispositivo, e *threads* de processos de usuário não podem utilizá-las. Uma vez que um *thread* aloca uma porta, está fica alocada por ele, até que o *thread* termine sua execução.

Tabela 2 - Portas e Eventos do *Kernel*

Porta	Evento
20	Processo Criado
21	Processo Finalizado
22	Porta Alocada
23	Porta Desalocada
24	Evento do Sistema de Arquivos Virtual
25	Exceção

3.6.2 Envio e Recebimento de Mensagens

O sistema de troca de mensagens fornece algumas chamadas básicas para a comunicação. Uma para alocação de portas, uma para o recebimento, e duas para o envio. A chamada para alocação permite que um *thread* especifique qual porta ele pretende alocar.

A chamada para o recebimento é simples. Os processos especificam se existe alguma restrição de destinatário na hora da chamada de recebimento, e entram em estado de espera por uma mensagem. Caso exista uma restrição de porta, o *thread* só voltará a executar quando uma mensagem vinda de uma determinada porta for recebida (no caso de clientes esperando respostas de servidores) ou quando uma mensagem recebida tiver sido destinada a uma determinada porta (no caso de servidores esperando por clientes). Quando não há restrições o *thread* retoma sua execução quando mensagem chega, independente de sua origem. As mensagens tem um limite de 100 *bytes*, e cabe ao processo especificar um *buffer* para o armazenamento do conteúdo da mensagem durante seu recebimento.

Para o envio de mensagens, existem duas opções. Uma mensagem pode ser enviada para uma porta ou diretamente para um *thread*. Se a mensagem for enviada a uma porta, e ela estiver alocada por um servidor, o *thread* emissor fica obrigatoriamente esperando por uma mensagem de resposta. Existe também a opção do envio de uma mensagem endereçada diretamente a um *thread* especifico através de seu identificador. Esta modalidade de envio não bloqueia o *thread* emissor, e é utilizada para que processos servidores consigam

se comunicar com seus clientes. A figura 16 mostra um exemplo de comunicação entre um cliente e um servidor, utilizando as chamadas disponibilizadas pelo sistema operacional.

As mensagens são utilizadas para comunicação entre os *drivers* de dispositivos e seus clientes. As mensagens possuem um tamanho fixo de 100 *bytes* para simplificar a implementação e evitar a troca de uma quantidade maior de dados através do sistema de mensagens. Quando os processos precisam trocar uma maior quantidade de dados entre si, eles podem utilizar um arquivo criado no sistema de arquivos virtual, para o qual não há limite de tamanho. A mensagem pode ser utilizada para especificar o caminho do arquivo, ou para trocas de outras informações que sejam importantes para os *threads* envolvidos.

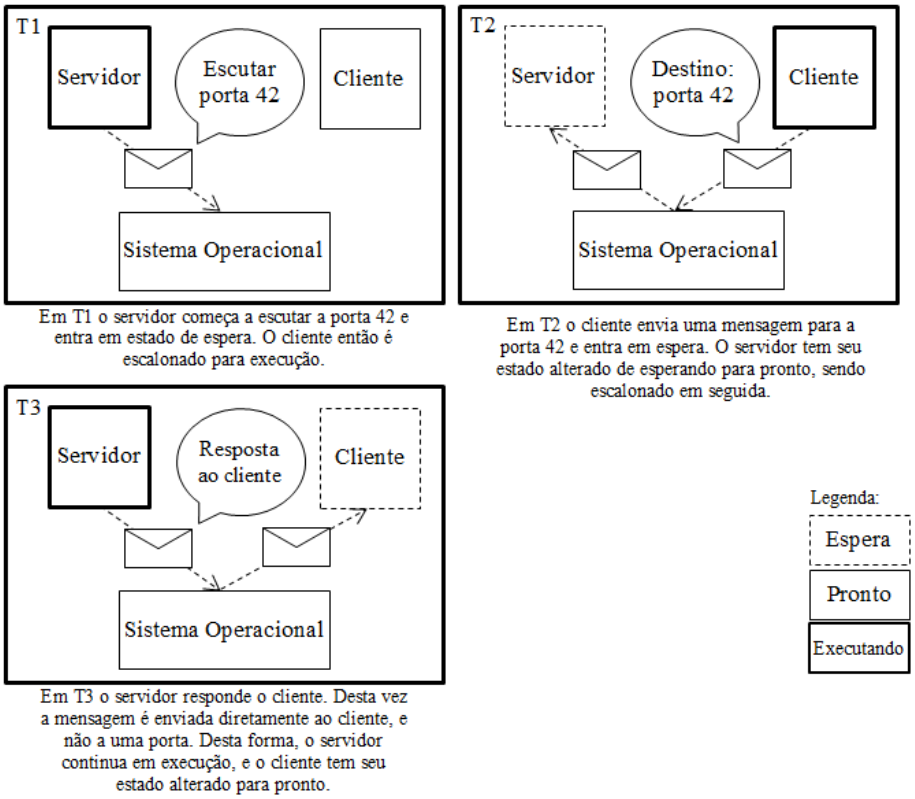

Figura 16 - Exemplo de Troca de Mensagens

3.7 Sistema de Arquivos Virtual

O núcleo do sistema possui um suporte básico a sistemas de arquivos externos ao *kernel*, através de uma abstração chamada de sistema de arquivos virtual ou VFS (*virtual file*

system). A ideia principal do VFS é abstrair a parte comum aos diferentes sistemas de arquivo e colocar o código em uma camada separada que chama o sistema de arquivos subjacente para fazer o gerenciamento dos dados (TANENBAUM, 2010, p. 178).

O sistema de arquivos virtual permite que arquivos sejam carregados na memória e acessados de forma uniforme, independente de como os dados estão estruturados nos sistemas de arquivos externos. Diferentes *drivers* para sistemas de arquivos podem criar pontos de montagem em uma hierarquia de diretórios que é comum a todos. Do ponto de vista do usuário, existe apenas uma hierarquia de diretórios embora na prática diferentes componentes de *software* tenham que trabalhar em conjunto para possibilitar o acesso aos dados.

3.7.1 Estrutura e representação dos Arquivos

O sistema FESO trata os arquivos como apenas um conjunto de *bytes*, não atribuindo nenhum significado em especial a eles. Cabe ao criador do arquivo designar o significado de cada *byte*, em cada posição do mesmo. Isto permite uma grande flexibilidade por parte dos programas, que podem fazer uso desta abstração da maneira que for mais conveniente.

Cada arquivo do sistema está em uma mesma estrutura hierárquica, formada por arquivos e diretórios, como é possível ver na figura 17. Os diretórios são arquivos especiais, que agrupam um conjunto de arquivos e outros diretórios, e formam a estrutura do sistema de arquivos.

A estrutura começa a partir do diretório raiz (representado pelo caractere '/'), que possui vários outros diretórios. Alguns deles são diretórios locais, e estão armazenados apenas na memória. Outros são arquivos externos, que ficam em dispositivos de armazenamentos, e são carregados para a memória apenas quando preciso.

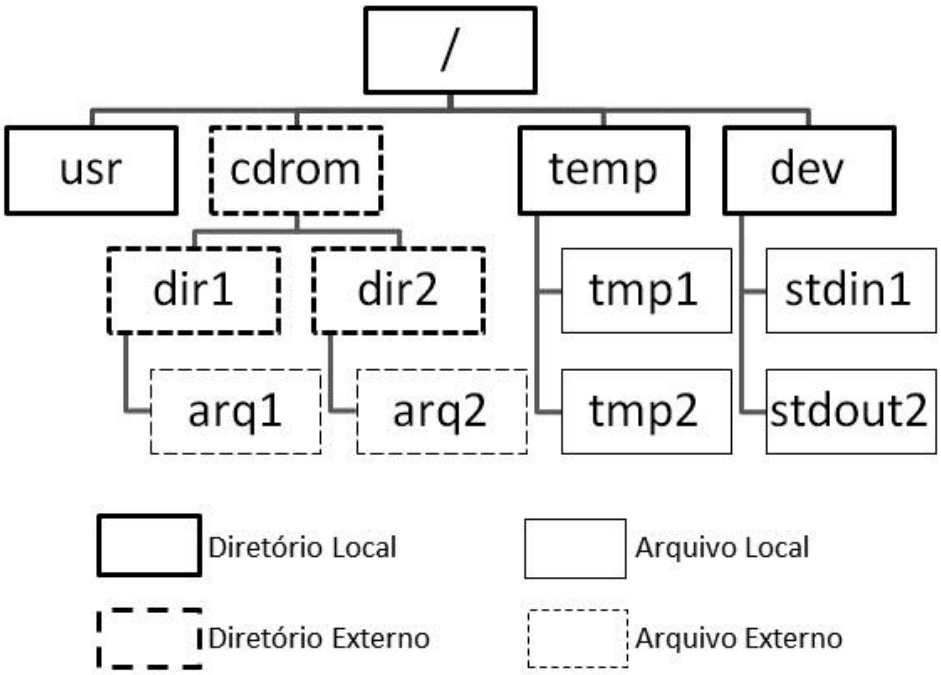

Figura 17 - Hierarquia do VFS

Cada arquivo ou diretório é tratado como um nó no sistema de arquivos. Para cada nó existe um conjunto de informações que é mantida pelo SO, tais como: o tipo do nó (arquivo comum, diretório), o nome, o tamanho, seu local (memória ou dispositivo de armazenamento), e uma lista *clusters* com os dados armazenados.

Essas estruturas ficam na memória, em uma árvore balanceada. Elas são ordenadas pelo descritor do nó ao qual representam. O descritor é um número sequencial utilizado na identificação de cada um dos nós. Tanto nós de arquivos como de diretórios possuem um descritor, e ele é utilizado também pelos programas para realizar operações sobre esses nós.

A figura 18 mostra como os arquivos ficam estruturados na memória quando carregados. Os diretórios do sistema de arquivo virtual do FESO possuem apenas uma unidade de armazenamento (*cluster*) de 512 *bytes*. Este *cluster* contém um vetor de números

inteiros de 4 *bytes*. A primeira posição desse vetor contém um inteiro N, que armazena o número de outros nós contidos naquele diretório. As N posições seguintes armazenam o identificador de outros nós dentro do diretório. Como cabem apenas 128 valores inteiros em um *cluster* de 512 *bytes*, e como o primeiro é utilizado para indicar a quantidade de nós, os diretórios estão limitados a uma quantidade máxima de 127 nós filhos. A partir da leitura do *cluster* único contido nos nós que representam diretórios, é possível identificar quais outros nós o diretório contém.

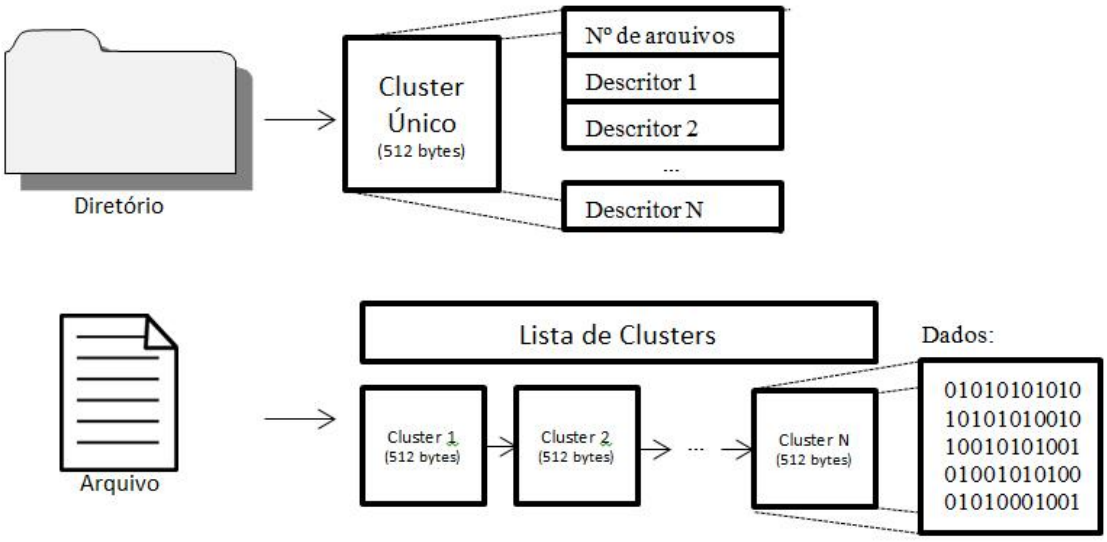

Figura 18 - Estrutura dos arquivos e diretórios

Os arquivos comuns, por sua vez, possuem uma quantidade arbitrária de *clusters* de 512 *bytes*. Assim como as estruturas dos nós, os dados contidos nos *clusters* ficam armazenados na área de memória que é alocada dinamicamente pelo núcleo. Sistemas de arquivo externos carregam os dados nessa memória, e os tornam acessíveis a outros programas.

O sistema de arquivos virtual permite que um determinado conjunto de operações seja executada sobre os nós. Existem chamadas do sistema para a criação de nós de arquivos

comuns e de diretórios. Há ainda, chamadas tanto para criação de nós locais, que ficam apenas na memória e são perdidos quando o computador é desligado, quanto os externos, armazenados em um dispositivo de armazenamento, e gerenciados por um servidor de sistemas de arquivos que funciona de forma externa ao *kernel*.

Existem chamadas também para a leitura, escrita, fechamento e remoção de arquivos. Essas chamadas são realizadas diretamente pelas rotinas do núcleo quando feitas sobre um arquivo local. No caso de um arquivo externo, o núcleo informa ao cliente qual é o servidor responsável, e o próprio cliente solicita a execução da operação desejada sobre o arquivo. Isto é necessário, pois o *núcleo* não tem acesso às rotinas do servidor do sistema de arquivos externo, e apenas ele pode executar as operações de maneira correta.

As funções para tratamento de arquivos em espaço de usuário estão diretamente relacionadas às chamadas ao sistema oferecidas pelo VFS. A função de abertura/criação de arquivos prevê alguns modos de abertura. Alguns servem para a abertura de diretórios, outros para arquivos.

Há também, modos para abertura de arquivos, para a leitura e para a escrita. A diferença desses dois últimos é o posicionamento do cursor que existe para cada arquivo. Este cursor é uma variável que indica em que posição do arquivo a próxima operação de leitura ou escrita deve ser realizada. No caso da abertura do arquivo para escrita, o cursor é posicionado por padrão no final do arquivo, e para leitura no começo. De qualquer forma, existe uma chamada do sistema responsável por alterar a posição do cursor dentro do arquivo, permitindo o acesso a qualquer posição.

Uma boa parte das funções e chamadas do sistema que trabalham sobre arquivos se utilizam do descritor para identificá-los. Isto é, o descritor é geralmente um parâmetro para essas chamadas. Uma exceção é a chamada de abertura/criação que se utiliza de uma cadeia de caracteres com o nome do arquivo acompanhado de seu caminho completo a partir do

diretório raiz. Este caminho é composto pelo nome de todos os subdiretórios ordenados de acordo com sua posição na hierarquia do sistema de arquivos.

O arquivo *arq1* indicado na figura 18, por exemplo, teria o seguinte caminho: */cdrom/dir1/arq1*. Começando pelo diretório raiz (/), passando pelo diretório *cdrom* e *dir1* (que está contido em *cdrom*). Entre os nomes de cada um dos diretórios existe uma barra (/) para indicar a separação entre eles. Quando uma chamada para abertura/criação é bem-sucedida, ela retorna um número inteiro que é o identificador do arquivo, e este sim é utilizado nas outras chamadas.

3.7.2 Integração com Sistemas de Arquivo Externos

O sistema de arquivos virtual oferece uma estrutura básica para que arquivos sejam manipulados pelos programas. No caso de arquivos locais, todas as operações sobre os arquivos são feitas diretamente pelas rotinas do *kernel*. Os arquivos locais ficam salvos apenas na memória do computador, e são perdidos quando este é desligado.

Além disso, o sistema de arquivos virtual oferece uma estrutura para que arquivos externos sejam manipulados. Esses arquivos externos ficam salvos primeiramente em algum dispositivo de armazenado, como um CD ou um disco rígido. E sua manipulação não é feita apenas pelas rotinas do núcleo, existindo um programa em espaço de usuário que funciona como um servidor, acessando o dispositivo de armazenamento, e lidando diretamente com as peculiaridades de cada sistema de arquivos em particular.

A figura 19 mostra um exemplo básico de leitura de arquivo externo. Primeiramente, um cliente (um programa de usuário qualquer), tenta abrir um arquivo externo. Os servidores de sistema de arquivos devem, em seu processo de inicialização, criar nós no VFS que indiquem a existência de arquivos em outros dispositivos de armazenamentos. A principio, estes nós contém apenas uma lista de *cluster* vazia, sem nenhum dado.

Neste ponto, o cliente se comunica com o servidor de sistemas de arquivos, e espera que este realize a operação solicitada sobre o arquivo. No caso da abertura, os sistemas de arquivos externos carregam os dados do arquivo para a memória, o que permite que eles sejam acessados pelo cliente. Quando os dados são escritos, eles ficam salvos apenas na memória, e no momento em que o arquivo é fechado, o servidor entra em ação e move os dados recém-escritos para o dispositivo de armazenamento.

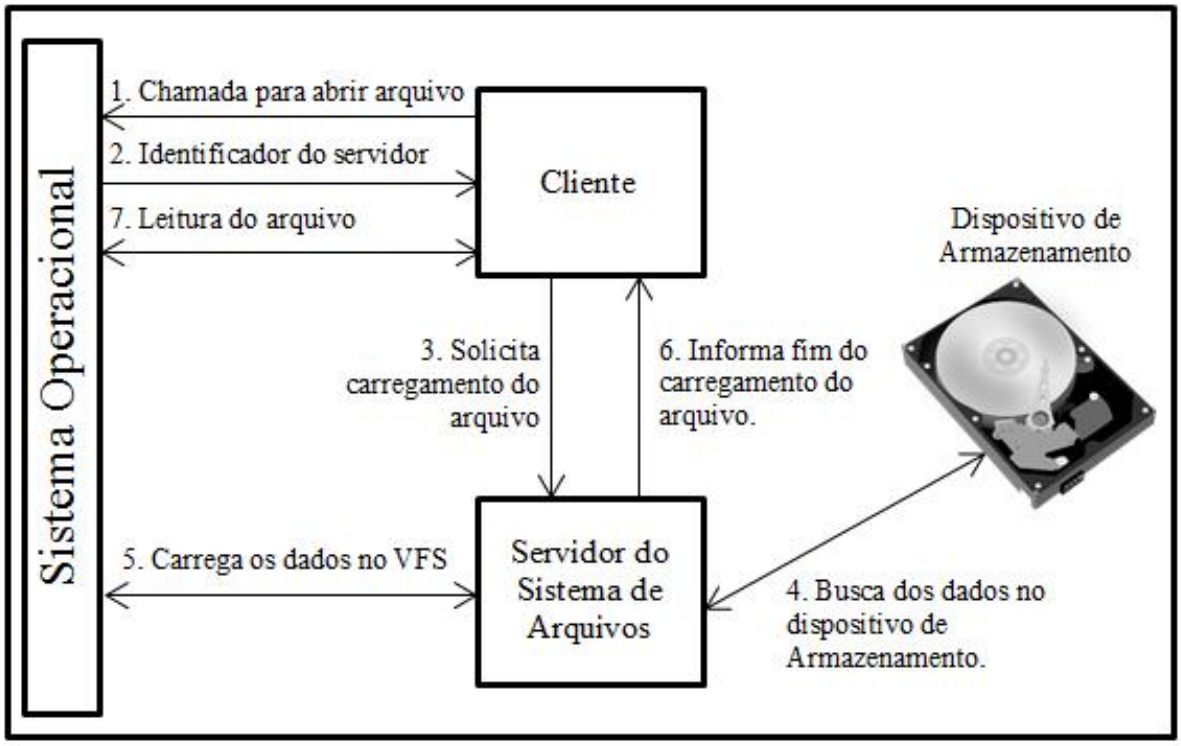

Figura 19 - Abertura de um arquivo externo

Na ocorrência de uma tentativa de qualquer operação sobre um arquivo externo, o núcleo apenas informa ao cliente que a operação não pode ser realizada diretamente, e retorna o identificador do processo responsável pelo arquivo. Quando isto ocorre, as funções da biblioteca padrão do sistema FESO automaticamente tentam entrar em contato com um servidor de arquivos. A operação solicitada pelo cliente é então repassada ao servidor, que deve possuir uma implementação especifica para cada caso, seja abertura/criação, leitura,

escrita, exclusão ou fechamento de arquivo. Após o término da operação, o servidor se comunica com o cliente, e este continua sua execução normalmente.

Este modelo permite que diversos sistemas de arquivos diferentes funcionem em uma mesma estrutura, e isola cada sistema de arquivos em um programa de usuário a parte. Um detalhe a ser acrescentado é que a figura 19 esconde o intermediário que pode existir entre o servidor do sistema de arquivos e o dispositivo de armazenamento. Os sistemas de arquivos desenvolvidos até o momento não acessam o dispositivo diretamente. E eles entram em contato com outro *driver* de dispositivo, e este faz a comunicação direta com o dispositivo.

3.8 Testes

O sistema operacional FESO foi testado tanto em máquinas virtuais quanto em algumas máquinas reais. Os testes em máquinas virtuais são mais rápidos, e auxiliam no processo de desenvolvimento. Entretanto, os testes em máquinas reais são necessários, pois o funcionamento das máquinas virtuais não é exatamente igual ao de máquinas reais.

O depurador, que é um recurso oferecido pelo *software* de virtualização VIRTUAL BOX, foi extensamente utilizado durante o processo de desenvolvimento. O depurador permite não só que o estado dos registradores da CPU durante a execução do código seja verificado, como também possibilita que o estado das tabelas de descritores seja conferido.

Alguns testes básicos de funcionalidade foram feitos em *hardware* real. O sistema FESO foi inicializado em máquinas com processadores INTEL e AMD, e algumas operações básicas, como carregamento de programas, abertura de arquivos entre outras funções foram testadas.

3.8.1 Rotinas de Testes

Algumas rotinas de testes foram desenvolvidas para automatizar a verificação da consistência do comportamento de algumas funções do *kernel* Essas rotinas não verificam todos os casos possíveis exaustivamente. Elas apenas verificam se algumas funções básicas do *kernel* estão funcionando como esperado. Entretanto, os resultados dos testes devem ser verificados sempre que alguma alteração no código fonte do núcleo for feita.

Existem dois grupos de rotinas de teste, as que funcionam dentro e fora do núcleo. As que funcionam dentro do núcleo testam as funções básicas da gerência de memória, e de processos. As que funcionam fora testam as chamadas ao sistema operacional.

Os testes dentro do núcleo que verificam a memória executam inicialmente um determinado conjunto de ações para determinar se o alocador de memória física está funcionando corretamente. Um conjunto de alocações e liberações de bloco é realizada e depois o estado dos blocos é verificado pelas rotinas. Há ainda a verificação do alinhamento dos blocos, e a tentativa de liberação de blocos inexistentes.

Posteriormente, as rotinas responsáveis pela gerência de memória virtual são testadas. Um novo espaço de endereçamento é criado, e nele são mapeados alguns endereços. Depois, acessos a esses endereços ocorrem, e caso nenhuma exceção seja gerada, é porque as rotinas estão mapeando os endereços corretamente.

Dentro do núcleo, ainda há testes da gerência de processos. As rotinas para criação e eliminação de processos são testadas. Um novo *thread* é adicionado ao processo e depois é removido. Os resultados são contabilizados, e o identificador do novo processo é verificado. A rotina de teste também tenta realizar a criação de um processo a partir de um arquivo inválido.

Todos os resultados, de todas as rotinas no interior do núcleo são armazenados em arquivos de texto no VFS. Se os resultados puderem ser corretamente lidos após a sua

execução, também já é um indício de que as rotinas do sistema de arquivos virtual estão funcionando corretamente.

As rotinas em espaço de usuário por sua vez executam outros tipos de verificação. Primeiramente, a rotina para expansão do *heap* do processo é testada, e as posições retornadas são analisadas para verificar se a distância entre as posições está de acordo com a quantidade de memória solicitada.

Após o teste da memória, ocorrem os testes de criação de *threads* e comunicação. Um novo *thread* é criado, desta vez em espaço de usuário. O *thread* principal então entra em estado de espera por uma mensagem, e quando esta chega, é porque foi enviada a partir do *thread* que acabou de ser criado. O conteúdo da mensagem é verificado, para garantir sua consistência.

As chamadas para o sistema de arquivos virtual também são testadas fora do núcleo. Alguns arquivos e diretórios são criados, e diversas operações são feitas sobre eles. Dados são escritos e posteriormente lidos, e os arquivos são apagados. Tentativas de leitura e escrita em arquivos inválidos são executadas, para verificar se as chamadas ao sistema se comportam corretamente.

Embora todos esses testes não possam garantir o correto funcionamento de absolutamente todas as rotinas do núcleo, eles já fornecem uma boa ideia da consistência de algumas funções principais deste. Além disso, por causa da grande dependência entre os componentes, caso alguma alteração comprometa seriamente o comportamento de uma função, dificilmente o resultado não será sentido quase que imediatamente.

3.8.2 Testes de Desempenho com Aplicações Paralelas

Alguns testes foram realizados com aplicações paralelas, no intuito de demonstrar o ganho de desempenho obtido. Duas aplicações foram desenvolvidas, uma para o cálculo de soma de matrizes, e outra para a multiplicação.

A tabela 3 mostra os resultados obtidos com as duas aplicações. Os testes foram realizados em um computador com o processador Intel i7 de 8 núcleos (4 núcleos reais e 4 núcleos virtuais) e sempre com matrizes de mesmo tamanho. É possível notar uma melhora de desempenho linear com a utilização de 4 núcleos. Com a utilização de mais deles, o ganho de desempenho é reduzido por causa da utilização dos núcleos virtuais de performance inferior.

Os resultados estão em microssegundos, e foram medidos usando o temporizador interno do computador. No caso do aplicativo para a soma de matrizes, as variações de tempo entre as execuções foram tão pequenas, que nem chegaram a alterar os algarismos significativos das medições. O programa para multiplicação, por sua vez, apresentou algumas variações de execução para execução.

Tabela 3 - Resultados dos Testes de Desempenho

	Soma de Matrizes				**Multiplicação de Matrizes**			
Nº de Núcleos	**1**	**2**	**4**	**8**	**1**	**2**	**4**	**8**
	28,16	15,07	7,81	5,17	4,10	2,20	1,27	1,16
	28,16	15,07	7,81	5,17	4,07	4,13	1,10	1,65
	28,16	15,07	7,81	5,17	4,07	2,04	2,04	1,32
	28,16	15,07	7,81	5,17	4,18	2,09	1,27	1,32
	28,16	15,07	7,81	5,17	4,24	2,20	1,21	1,38
Média	28,16	15,07	7,81	5,17	4,13	2,53	1,38	1,36
Speedup		1,87	3,61	5,45		1,63	3,00	3,03

As figuras 20 e 21 mostram respectivamente os gráficos com os tempos, e com o *speedup* para cada um dos testes. É possível notar a melhora de desempenho, tanto no caso da soma de matrizes, quanto na multiplicação. Entretanto, o ganho de desempenho e reduzido quando se aumento o número de *threads* de 4 para 8, pois o processador i7 possui apenas 4 núcleos reais. Os outros 4 núcleos presentes são virtuais que não proporcionam o mesmo ganho de desempenho que núcleos reais.

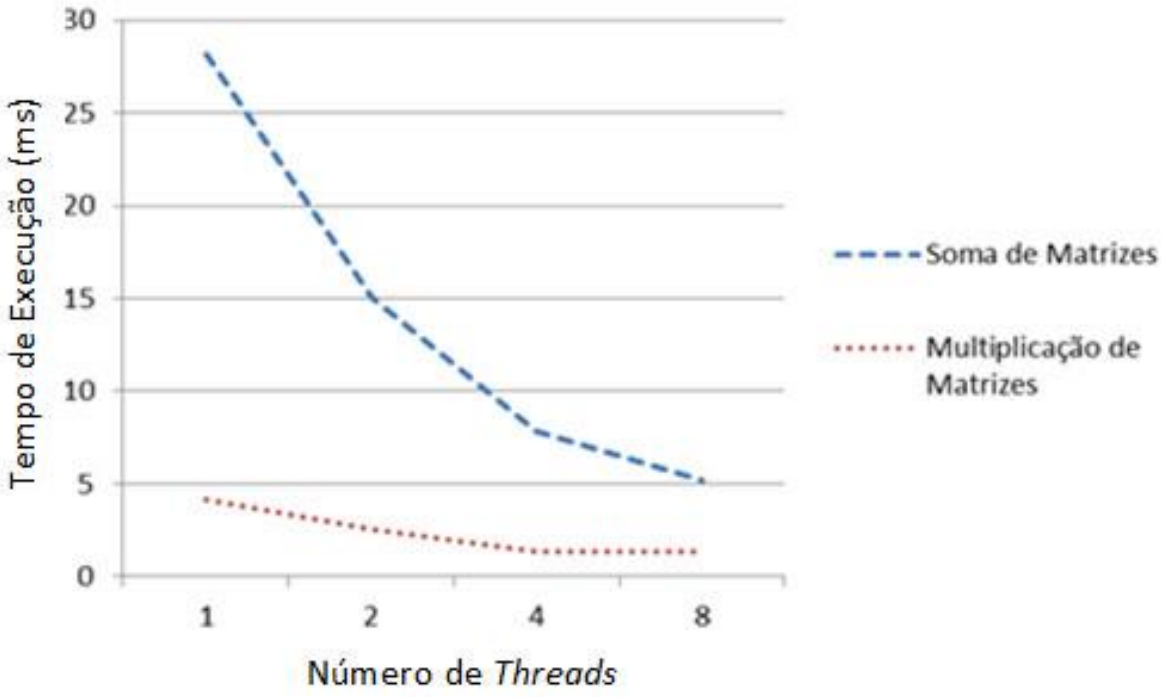

Figura 20 – Gráfico de Tempos

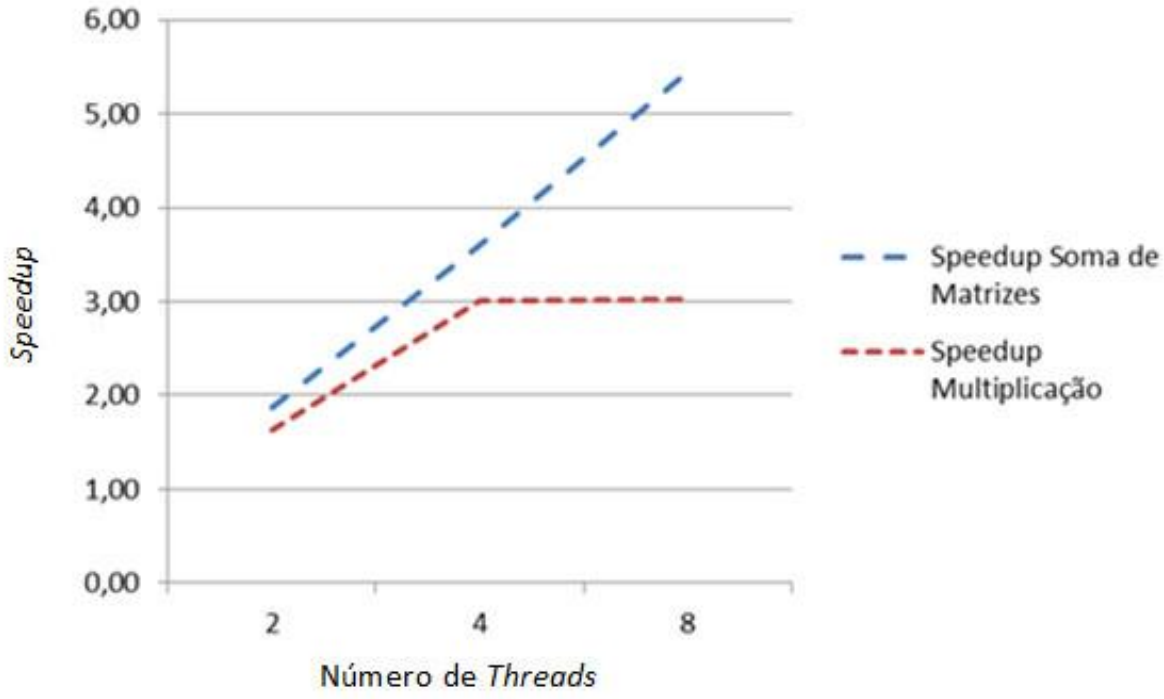

Figura 21 – Gráfico de *Speedup*

DESENVOLVIMENTO DOS DRIVERS DE DISPOSTIVOS

3.9 Driver do Console

O *driver* do console é o nome dado ao servidor responsável pelo controle do teclado e do vídeo do computador. Embora sejam dispositivos diferentes, eles foram integrados em um só servidor, pois existe uma interação constante entre eles. Isto é, aquilo que é digitado pelo usuário com o uso do teclado muitas vezes é imediatamente escrito no vídeo.

De maneira geral, o *driver* do console suporta diversos clientes ao mesmo tempo. Os clientes são *threads* que fazem solicitações ao *driver.* As solicitações são usualmente para a escrita de dados no vídeo ou para a leitura de dados do teclado. Entretanto, existem solicitações para alterar a cor do texto, ou a mesmo a posição do cursor na tela.

Para cada cliente diferente, o servidor cria um conjunto de variáveis para armazenar o estado da tela, e estas variáveis compõem o console virtual. Cada cliente tem seu próprio console virtual, com cada caractere na tela salvo em um arquivo, e com as configurações de cor, posição do *cursor*, e algumas outras armazenadas em uma estrutura. No momento em que um cliente fica em primeiro plano (*foreground*), os dados contidos em seus arquivos são carregados na memória de vídeo, e tudo é configurado de acordo com os dados salvos em sua estrutura. O usuário pode alternar entre diferentes consoles, apertando uma combinação de teclas, e então se comunicar com diversos clientes.

As operações de escrita no vídeo são realizadas diretamente na memória de vídeo apenas quando o cliente que as solicitou está em primeiro plano. Quando este está em segundo plano, os dados são gravados em um arquivo contendo o estado do vídeo. Além disso, a entrada dos dados do teclado só é redirecionada ao cliente que está em primeiro plano.

Quando um novo *thread* se comunica com o servidor, este verifica se já existe algum *thread* do mesmo processo que tenha feito alguma solicitação. Caso não exista, um novo console virtual é criado e a operação solicitada é realizada. Caso haja, o servidor executa a operação solicitada no console virtual já existente.

3.9.1 Driver do Vídeo

O *driver* do vídeo desenvolvido inicialmente para o sistema FESO suporta apenas placas de vídeo VGA no modo 0, no qual apenas texto é suportado. No modo 0, a tela do monitor de vídeo é dividida em 80 colunas por 25 linhas, como se fosse um grande reticulado (TORRES, 2001, P. 644).

Para escrever dados na tela, uma região especial da memória (a memória de vídeo) é utilizada. A memória de vídeo no modo 0 começa na posição B800h, de maneira que o caractere no canto superior direito é representado pelos primeiros 2 *bytes* a partir dessa posição de memória (MESSMER, 1999). Cada uma das 2000 posições (80 x 25) pode ser acessada por endereço correspondente a partir da posição de memória B800h, como mostra a figura 22.

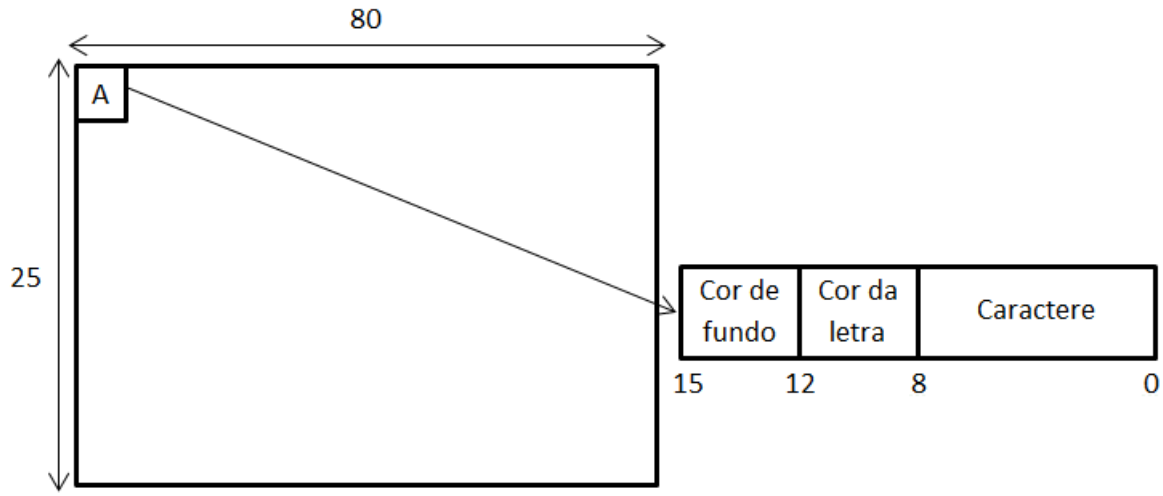

Figura 22 - Memória de Vídeo

Os caracteres são codificados por 2 *bytes*. O *byte* menos significativo armazena o código ASCII do caractere que deve ser impresso, e o *byte* mais significativo ainda se divide, de maneira que seus 4 *bits* menos significativos codificam a cor do caractere, e seus 4 *bits* mais significativos codificam a cor de fundo. Embora os caracteres fiquem dispostos visualmente como colunas e linhas, eles estão armazenados em uma região linear de memória. O *driver* trata esta região da memória como um vetor de palavras de 2 bytes. O índice no vetor para escrever um caractere na linha i e na coluna j é dado por: i*80 + j.

3.9.2 Driver do Teclado

O teclado é um dispositivo que permite que os usuários façam a entrada de dados para os programas. Sempre que uma tecla é pressionada ou liberada, a IRQ de índice 1 é gerada. O *driver* do teclado aloca a porta referente a essa IRQ durante o processo de inicialização do sistema operacional, desta maneira, sempre que uma tecla é pressionada, o *driver* do teclado muda seu estado de espera para pronto, e executa uma rotina para ler a tecla.

O *driver* do teclado acessa uma determinada porta de E/S para ler do *buffer* do teclado qual tecla foi pressionada pelo usuário. O valor lido pelo *driver* é conhecido como *scancode* ou código de varredura. A relação entre o *scancode* e a tecla depende do *layout* do teclado em si, e varia de acordo com o idioma do dispositivo. O *driver* do sistema FESO implementa um vetor que faz um mapeamento entre os *scancodes* e as teclas para um teclado brasileiro, permitindo a conversão dos valores lidos em caracteres.

TANENBAUM (2010) descreve os dois modos nos quais os *drivers* de teclado funcionam para atender as necessidades dos programas. São eles: o modo bruto (*raw mode*) e o modo preparado (*cooked mode*), também conhecido como modo canônico.

No modo bruto, o caractere representado por cada tecla é repassado diretamente ao programa, sem nenhuma interferência do *driver*. Este é o modo ideal para programas que precisam de uma maior liberdade para lidar com as teclas pressionadas pelo usuário. É o caso

de um editor de texto que se encarrega de tomar uma determinada ação para cada tecla diferente que é pressionada pelo usuário.

Porém, há programas que não tem necessidade de tanta liberdade, para os quais uma entrada corrigida dos dados é mais conveniente. Por exemplo, um interpretador de comandos que aguarda a entrada de um comando do usuário. Supondo que o usuário irá digitar o comando LISTAR. Porém, um erro ocorre, e ele digita um R a mais no final da palavra. A tecla *backspace* é pressionada, para que o último caractere digitado seja removido. Neste exemplo a sequência de teclas digitadas pelo usuário foi L-I-S-T-A-R-R-*backspace.*

Tudo que o programa precisa saber é que o resultado final obtido pelo pressionamento das teclas foi a cadeia de caracteres LISTAR. O programa não precisa da entrada completa, apenas do resultado corrigido. Neste caso, o modo preparado é o mais indicado, pois livra o programa da responsabilidade de ajustar a entrada, deixando-a a cargo do *driver*.

Outra característica necessária do *driver* é o ecoamento de caracteres. O caractere correspondente a cada tecla pressionada deve ser escrito (ecoado) na tela, de modo a permitir que o usuário visualize o resultado do pressionamento da tecla de forma imediata. O ecoamento, utilizado principalmente com o modo preparado, simplifica em muito a relação dos programas com o *driver*, fazendo com que apenas as cadeias corrigidas sejam enviadas as programas, quando o usuário termina de fazer sua entrada de dados pressionando a tecla ENTER.

3.10 Driver do Temporizador

O controle do tempo é fundamental para o sistema operacional, e muitas vezes para os próprios aplicativos de usuário. Sistemas operacionais precisam controlar a quantidade de tempo que um processo está em execução. Programas de usuário podem

precisar esperar um determinado período de tempo para continuar sua execução, ou mesmo executar um trecho de código em um momento especifico.

Por causa desta necessidade, os projetistas dos PCs implementaram o PIT, *programmable interval timer*. O PIT é um *chip* que gera uma interrupção em intervalos de tempo programáveis com base em um oscilador de cristal independente da CPU (MESSMER, 1999, p. 681). Isto quer dizer que as frequências de funcionamento do PIT independem da CPU presente no computador.

O *driver* do temporizador do sistema FESO tem algumas funções importantes. A primeira delas é a configuração do PIT, fazendo com que este gere uma interrupção (IRQ 0) aproximadamente a cada 50 ms. Essa interrupção faz com que a CPU interrompa o programa em execução, coloque o *driver* do temporizador em execução, e quando este entra em espera, um novo *thread* é escalonado na CPU.

Além de auxiliar no compartilhamento da CPU, este *driver* também pode receber solicitações de *threads* que desejam ficar em espera por um determinado período de tempo. Supondo que um *thread* queira ficar aguardando durante 1000 ms, basta enviar uma mensagem ao servidor, e este irá criar um novo contador para o *thread*. O *thread* fica em estado de espera, aguardando por uma resposta do servidor. Enquanto isso, o servidor atualiza o contador a cada nova interrupção, e quando o contador chega ao limite, uma mensagem é enviada para o *thread* cliente, a fim de acordá-lo.

Outra função importante deste *driver* é o controle da hora do sistema. No momento de sua inicialização, além da configuração do PIT, o *driver* do temporizador também obtém a hora atual do sistema se comunicando com a BIOS do computador. Uma vez que a hora é obtida, o próprio *driver* se encarrega de manter um contador de horas, minutos, segundos e milissegundos. Assim sendo, os programas de usuário podem obter a hora do sistema fazendo uma simples solicitação do *driver* do temporizador.

3.11 Driver IDE

O sistema operacional FESO possui um suporte básico a dispositivos IDE. Este suporte foi desenvolvido para permitir a utilização de sistemas de arquivos externos ao *kernel*, que necessitam de acesso a algum dispositivo de armazenamento. O *driver* desenvolvido para o sistema FESO suporta a execução de operações simples nas interfaces ATA e ATAPI, e permite a comunicação com *drivers de* discos rígidos e de CD-ROM que funcionem neste formato.

O *driver* desenvolvido utiliza entrada e saída programada. Neste modo de E/S, o processador precisa executar uma sequência explicita de instruções para cada caractere lido ou escrito (TANENBAUM, 2006, p. 225). O problema com essa abordagem, é que todos os dados lidos passam obrigatoriamente pela CPU, como mostra figura 23, fazendo com que ela fique ocupada.

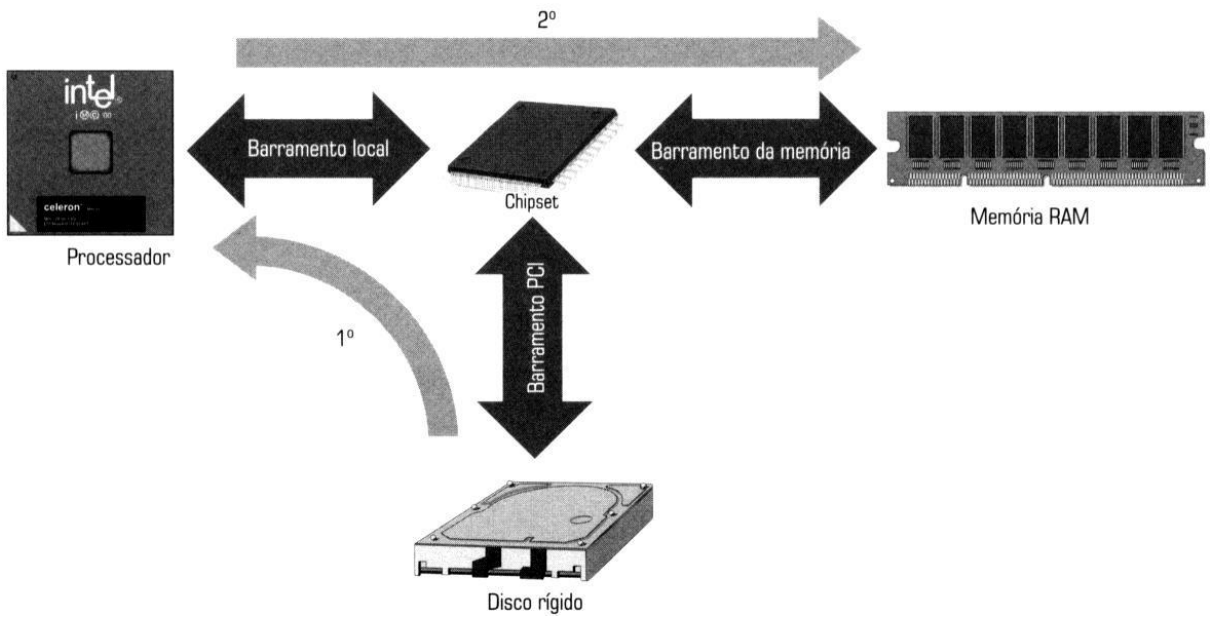

Figura 23 - Utilização de E/S programada (TORRES, 2001, p. 815)

As operações de leitura e escrita em discos rígidos e em outros dispositivos podem ser consideradas lentas em comparação com a velocidade na qual as CPUs podem

executar instruções. Por este motivo, em alguns momentos a CPU simplesmente fica em espera ocupada, até que uma operação seja concluída. Este problema não ocorre com o uso de outras técnicas para a realização de entrada e saída, como o acesso direto a memória, no qual os dados não precisam passar pela CPU.

De qualquer forma, o *driver* para dispositivos IDE foi desenvolvido utilizando entrada e saída programada, pois esta é uma forma de comunicação simples, e suportada pela grande maioria dos dispositivos. Entretanto, futuramente, melhorias no *driver* atual poderão permitir formas mais rápidas de E/S com esses dispositivos.

Durante o processo de inicialização do *driver*, ocorre a detecção de dispositivos ATA e ATAPI conectados ao computador. Os primeiros dispositivos ATA e ATAPI encontrados tornam-se imediatamente os dispositivos padrão. Todas as solicitações de leitura e escrita enviadas ao *driver* são repassadas automaticamente aos dispositivos detectados.

O *driver* oferece aos clientes algumas funções básicas. Ele permite a leitura e escrita de dados em um determinado bloco de um dispositivo ATA, usualmente, um disco rígido. Também permite a leitura de blocos em dispositivos ATAPI, por exemplo, um *drive de* CDROM. Essas funções são utilizadas pelos servidores dos sistemas de arquivos, que precisam acessar diretamente os dispositivos, e obter a estrutura do sistema de arquivo contido nele.

3.12 Sistemas de Arquivo ISO 9660

O padrão mais comum entre os sistemas de arquivos para CD-ROMs foi adotado como um Padrão Internacional em 1988 sob o nome ISO 9660 (TANENBAUM, 2010, p. 193). Por este motivo, junto ao sistema FESO, foi desenvolvido um programa servidor capaz de suportar este sistema de arquivos.

Os CD-ROMs possuem blocos lógicos de 2048 *bytes* cada. Esses blocos são indexados, sendo acessados através de um número. Os primeiros blocos de um CD são reservados para algumas estruturas utilizadas pelo sistema de arquivos, para que seja possível identificar a posição na qual cada arquivo está.

Ao ser iniciado, o servidor lê o bloco 16 do disco, onde está localizado o descritor de volume primário. Este descritor de volume primário é uma estrutura que armazena algumas informações sobre o volume. O registro do diretório raiz, que é a estrutura que armazena as informações necessárias para se encontrar o diretório raiz do volume, se encontra também dentro do descritor de volume primário.

Existe ainda, uma outra estrutura que é referenciada pelo descritor de volume primário, chamada de *path table*. A *path table* é uma estrutura que fornece uma maneira alternativa de realizar a leitura dos dados em CDROMs com sistemas de arquivos ISO 9660. Ela contém uma lista com todos os diretórios existentes no volume, como é possível ver na figura 24. Entretanto, está estrutura não é utilizada pelo servidor, que utiliza apenas os registros de diretório.

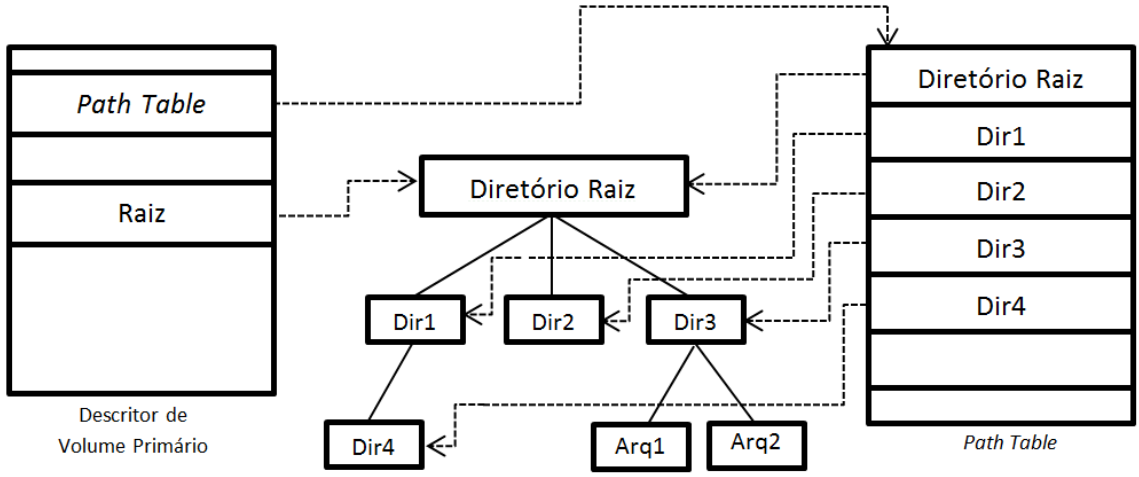

Figura 24 - Estrutura do sistema de arquivos ISO 9660

Os diretórios no sistema de arquivos ISO 9660 devem ser armazenados como um arquivo contendo um conjunto de registros que servem para identificar um arquivo ou outro diretório (ECMA, 1987). Isto é, cada diretório é tratado como um arquivo cujo conteúdo é uma lista de registros que identificam outros arquivos e diretórios dentro do volume.

Os arquivos no sistema ISO 9660 (e isto inclui os diretórios) ficam armazenados em blocos lógicos contínuos. Por este motivo, para ler um arquivo por completo, é necessário apenas conhecer seu bloco inicial e seu tamanho. Com estas duas informações, basta ler um conjunto de blocos lógicos a partir daquele que foi identificado como o inicial que seja grande o suficiente para armazenar todo o arquivo. É desta forma que, não só os dados dos arquivos convencionais são lidos, como também os dados de arquivos que representam diretórios.

O servidor realiza uma varredura por todo o sistema de arquivos, começando pelo diretório raiz. O registro do diretório raiz é lido, sua posição no disco é encontrada e seus dados carregados. Esses dados contém uma lista de registros de diretório, que representam todos os arquivos ou diretórios contidos dentro do diretório raiz. Todos os registros que representam diretórios são adicionados a uma fila, que é utilizada em uma busca em largura. Enquanto essa fila não estiver vazia, o primeiro registro dela é carregado e analisado. Caso outros registros que representem diretórios sejam encontrados, estes são também adicionados na lista.

O servidor se encarrega de criar um nó no VFS do sistema FESO para cada arquivo e diretório encontrado no CD-ROM. Desta forma, qualquer processo pode verificar, utilizando as chamadas do próprio VFS, quais arquivos estão contidos dentro do CD-ROM. Além disso, o servidor armazena uma tabela com o identificador de cada arquivo no VFS e sua posição e tamanho dentro do CD-ROM, para que seja possível carregá-lo na memória quando solicitado por outros processos.

4 DESENVOLVIMENTO DE APLICATIVOS

4.1 Aplicativos em Espaço de Usuário

O sistema operacional FESO possui suporte a novos programas escritos em C/C++ e linguagem *assembly*. Com a utilização de um compilador cruzado, e de outras ferramentas é possível desenvolver novos programas que funcionem sobre o FESO.

Os programas para o FESO devem obedecer a algumas restrições. Os binários devem estar no formato ELF, e todos os trechos de código utilizados pelo programa devem estar ligados em um único arquivo. Apenas é possível utilizar recursos da linguagem e bibliotecas suportadas pelo sistema. Isto é, alguns recursos da linguagem C++ que necessitem de suporte em tempo de execução não funcionarão corretamente até que o sistema FESO seja adaptado para tal.

Foi desenvolvido um conjunto de funções para realizar todas as chamadas de sistema, e para realizar a comunicação básica com os *drivers* de dispositivos desenvolvidos. Essa biblioteca pode ser utilizada para a criação de programas simples, e pode também ser expandida à medida que novos recursos sejam adicionados.

4.2 Interpretador de Comandos

O primeiro programa de usuário iniciado junto ao sistema FESO é o interpretador de comandos. Este programa tem a função de ler a entrada de usuários, e executar comandos relativos a essas entradas. O interpretador de comandos funciona totalmente em espaço de usuário, e faz o uso de uma biblioteca para a comunicação com *drivers* de dispositivos e das chamadas ao sistema FESO para conseguir realizar suas funções.

O interpretador tem comandos para a navegação pelo sistema de arquivos, para a criação e finalização de outros programas, entre outras funções. A tabela 4 mostra uma lista

de comandos suportados, e a função de cada um deles. A figura 25 mostra a tela do sistema operacional executando o interpretador de comandos.

Tabela 4 - Comandos do interpretador

Comando	Função
Exec	Executa um programa.
Novo	Cria um novo nó de arquivo no VFS.
Nova_pasta	Cria um novo nó de diretório no VFS.
Listar	Exibe uma lista com os arquivos e diretórios presentes no diretório atual.
Remover	Remove um nó do VFS.
Copiar	Copia um arquivo,
Ir	Altera o diretório atual do programa.
Voltar	Volta um nível na hierarquia de diretórios a partir do diretório atual.
Renomear	Altera o nome de um arquivo. Pode ser utilizado para remover o arquivo de um diretório para outro.
Proc	Exibe uma lista dos processos em execução.
Portas	Exibe uma lista com as portas alocadas.
Mem	Exibe a quantidade de memória física total e disponível.
Limpar	Limpa a tela do editor de textos.
Finalizar	Finaliza um programa em execução.
Ajuda	Exibe uma lista com o nome dos comandos.

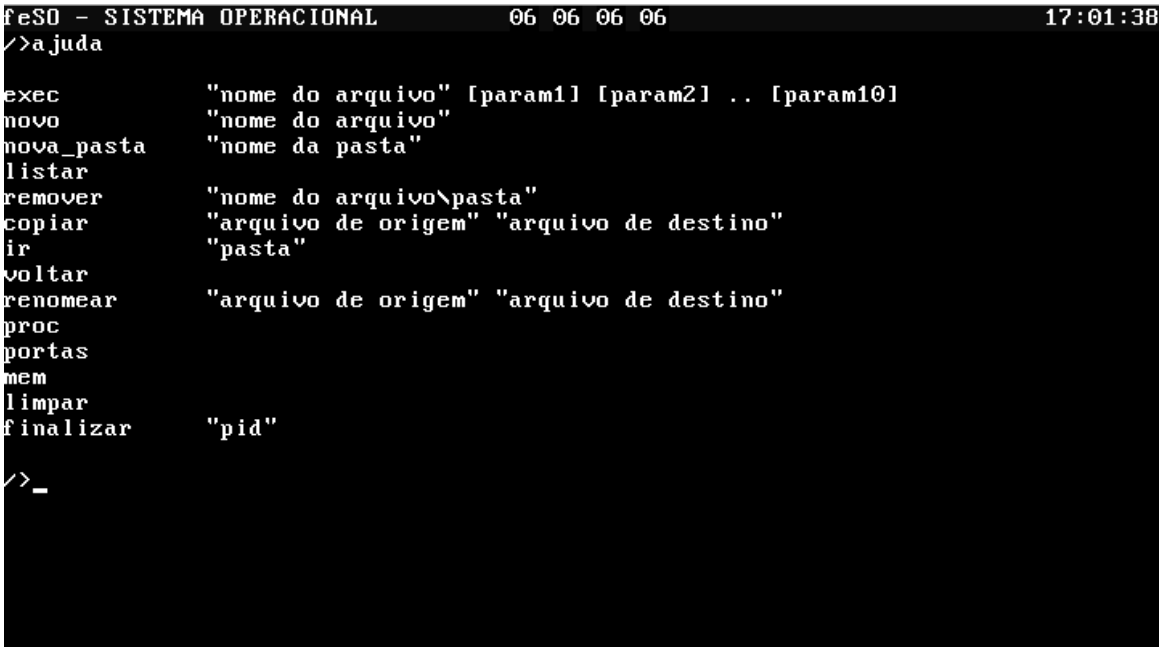

Figura 25 - Tela do Interpretador de Comandos

4.3 Editor de Textos

Um dos primeiros programas de usuário desenvolvidos junto ao sistema foi o editor de textos. O editor de textos padrão do sistema FESO é muito simples, e é capaz apenas de lidar com arquivos de texto puro, sem nenhum tipo de formatação. A figura 26 mostra o a tela do editor de textos.

Figura 26 - Tela do Editor de Textos

Para abrir um arquivo com editor de texto é preciso executá-lo no interpretador de comandos, e passar por parâmetros o nome do arquivo que se quer abrir. O caminho completo do arquivo fica em exibição na barra superior do editor. Caso nenhum arquivo seja informado, o editor de textos irá criar um arquivo chamado texto, no diretório raiz do sistema de arquivos virtual.

Na barra inferior existe a indicação das funções existentes no editor. A primeira função, sair, é executada quando a tecla ESC é pressionada. Ela fecha o editor sem salvar o arquivo aberto. A opção NOVO, ligada a tecla F1 simplesmente limpa todo o conteúdo do arquivo, e a tecla F2 salva o arquivo atual. A tecla F3 recarrega o conteúdo do arquivo na tela, desfazendo qualquer alteração.

O editor pode ser utilizado para a visualização, criação e edição de arquivos de texto simples. Ele é pode ser bastante útil, sobretudo para a visualização das saídas geradas por outros programas em formato de arquivo. O editor, em conjunto com o interpretador de comandos oferecem as ferramentas mais básicas para a utilização do sistema operacional FESO.

5 CONCLUSÕES E SUGESTÕES PARA TRABALHOS FUTUROS

O objetivo inicial do projeto foi atingido, e ao final foi desenvolvido um sistema operacional simplificado, porém funcional. Não só o núcleo e suas funções foram implementados e testados, como também uma série de *drivers* de dispositivos e alguns programas de usuário.

Neste trabalho procurou-se desenvolver um SO simplificado, que pudesse ser utilizado como uma ferramenta de estudo para os interessados no funcionamento de sistemas operacionais. No entanto, este sistema ainda possui diversas limitações, que podem ser superadas através de desenvolvimentos futuros.

Além disso, nenhum esforço foi realizado até o momento no sentido de se utilizar o sistema FESO dentro de um curso de sistemas operacionais. Nem mesmo foram feitas avaliações para determinar a eficácia do uso do FESO como ferramenta de aprendizado. Ficando assim, esta avalição para projetos futuros.

Com base nas limitações atuais do sistema operacional FESO, e com o atual método de sincronização entre processadores, que é funcional, porém não é o mais eficiente, as seguintes propostas para elaboração de trabalhos futuros são sugeridas:

- Desenvolvimento de uma estrutura para suporte de uma Interface Gráfica;
- Desenvolvimento do suporte a novos sistemas de arquivo;
- Criação de novos *drivers* de dispositivos para adaptadores de rede;
- Desenvolvimento de *drivers* de dispositivo para discos mais modernos, utilizando acesso direto à memória;
- Suporte para troca de páginas entre a memória e o disco usando paginação;
- Melhoria no método de sincronização entre processadores;

- Desenvolvimento de *drivers* de dispositivos diversos.
- Porte para o sistema FESO da biblioteca padrão C e de outras ferramentas, tais como compiladores e *linkers*.

6 REFERÊNCIAS

ACPI. Advanced configuration & power interface. 2011. Disponível em: <http://www.acpi.info/>. Acesso em: 07 jan. 2013.

CPLUSPLUS. 2013. Disponível em: <http://www.cplusplus.com/>. Acesso em: 30 jun. 2013.

ECMA. Volume and file structure of cdrom for information interchange. 1987. 59 p. Disponível em: <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-168.pdf>. Acesso em: 07 jan. 2013.

FREE SOFTWARE FOUNDATION. Cross-compilation. 2012. Disponível em: <http://www.gnu.org/savannah-checkouts/gnu/automake/manual/html_node/Cross_002dCompilation.html>. Acesso em: 07 jan. 2013.

FREE SOFTWARE FOUNDATION. GCC, the GNU Compiler Collection. 2013. Disponível em: < http://gcc.gnu.org/>. Acesso em: 01 jun. 2013.

FREE SOFTWARE FOUNDATION. Gnu grub. 2010. Disponível em: <http://www.gnu.org/software/grub/>. Acesso em: 07 jan. 2013.

HYDE, Randall. The Art of Assembly Language. 1. ed. Ed. No Starch, 2003.

INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual.2011. 4161 p. Disponível em: <http://www.intel.com/content/www/us/em/processors/architectures-software -developer-manuals.html>. Acesso em: 07 jan. 2013.

INTEL. Multiprocessor Specification. 1997. 97 p. Disponível em: <http://www.intel.com/design/archives/processors/pro/docs/242016.htm>. Acesso em: 07 jan. 2013.

KERNIGHAN, Brian W; RITCHIE, Dennis M. The C Programming Language. 2. ed. Prentice Hall, 1988.

LIEDTKE, J.. On micro-kernel construction. Acm sigops operating systems review. Nova Iorque, p. 237. 03 dez. 1995.

MACHADO, Francis B; MAIA, Luiz Paulo. Arquitetura de Sistemas Operacionais. 4. ed. Rio de Janeiro: LTC, 2007.

MESSMER, Hans-Peter. The Indispensable PC Hardware Book. 3. ed. Addison-Wesley Professional, 1999.

OSDEV WIKI. Global Descriptor Table. 2012. Disponível em: <http://wiki.osdev.org/GDT>. Acesso em: 07 jan. 2013.

SILBERSCHATZ, Abraham; GALVIN, Peter; GAGNE, Greg. Sistemas Operacionais: Conceitos e Aplicações. 1. ed. Rio de Janeiro: Campus, 2000.

SHAY, William. Sistemas Operacionais. 1. ed. São Paulo: Ed. Makron Books, 1996.

STALLINGS, William. Arquitetura e Organização de Computadores. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

STALLINGS, William. Operating Systems: Internals and Design Principles. 7. ed. Prentice Hall, 2011.

TANEBAUM, Andrew S. Organização Estrutura de Computadores. 6. ed. São Paulo: Prentice Hall, 2006.

TANEBAUM, Andrew S. Sistemas Operacionais Modernos. 3. ed. São Paulo: Pearson Education, 2010.

TANEBAUM, Andrew S; WOODHULL, Albert S. Sistemas Operacionais: Projeto e Implementação. 2. ed. Porto Alegre: Bookman, 2000.

THE NASM TEAM. The netwide assembler. 2012. Disponível em: < http://www.nasm.us/>. Acesso em: 30 jun. 2013.

TOOL INTEFACE STANDARD (TIS). Portable format specification. 1993. 262 p. Disponível em: <http://www.acm.uiuc.edu/sigops/rsrc/pfmt11.pdf>. Acesso em: 07 jan. 2013.

TORRES, Gabriel. Hardware: Curso Completo. 4. ed. Rio de Janeiro: Ed. Axcel Books, 2001.

7 APÊNDICE A - COMPILANDO O KERNEL E APLICATIVOS PARA O FESO

Requisitos

Para fazer alterações no *kernel* e criar novos aplicativos, os seguintes programas são necessários:

-Cygwin (Disponível em: http://www.cygwin.com/)

-Cross-compiler (Disponível no repositório: http://code.google.com/p/hermano-os/downloads/list, sobre o nome: CrossCompiler-Win32(i586-elf).part01.exe e CrossCompiler-Win32(i586-elf).part02.exe).

- Algum editor de texto.

Além dos programas, é necessário baixar o arquivo com todo o código fonte do *kernel*, mais o GRUB. Esses arquivos também podem ser obtidos nos repositórios (http://code.google.com/p/hermano-os/downloads/). É aconselhável baixar a última versão disponível.

Cygwin

O *Cygwin* é composto por um conjunto de ferramentas que permitem que o sistema operacional Windows possa ser utilizado de forma parecida com um sistema UNIX. Depois de baixar o instalador e executá-lo, é preciso selecionar a partir de qual repositório o *download* das ferramentas será feito. Após isso, será aberta uma janela para seleção de quais ferramentas devem ser baixadas.

Algumas ferramentas precisam ser selecionadas na tela de instalação, indicada na figura 27, para permitir o desenvolvimento do SO. As ferramentas necessárias são:

-NASM (*Netwide assembler*)

-Make (programa para criação de builds)

-Mkisofs (programa para criar imagens de CD-ROM bootável)

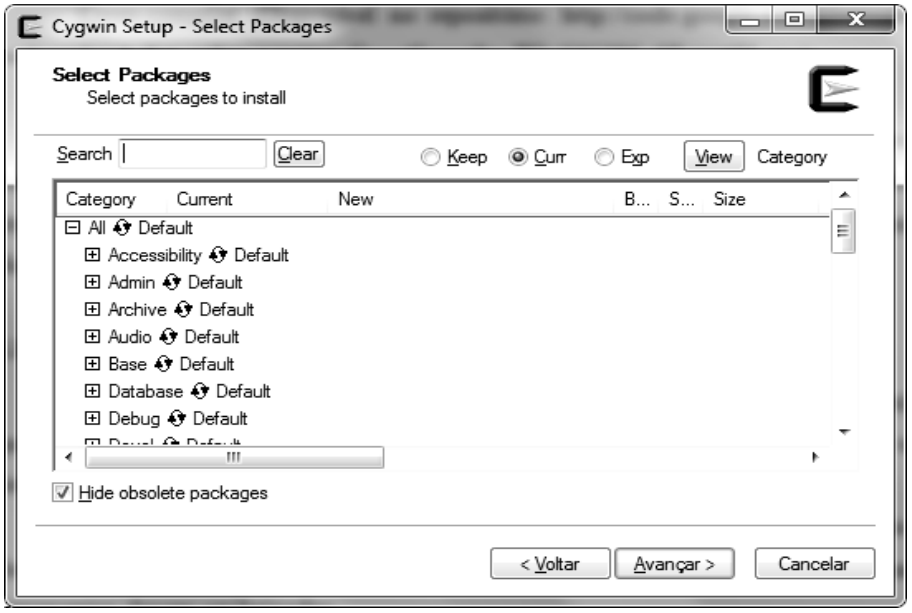

Figura 27 – Instalador do *Cygwin*

Cada uma dessas ferramentas deve ser buscada no repositório usando a caixa de pesquisas *search*, como mostra a figura 28. Dependendo da ferramenta selecionada, aparecerão diversas opções, é preciso encontrar a opção correta verificando o nome da ferramenta em *package*.

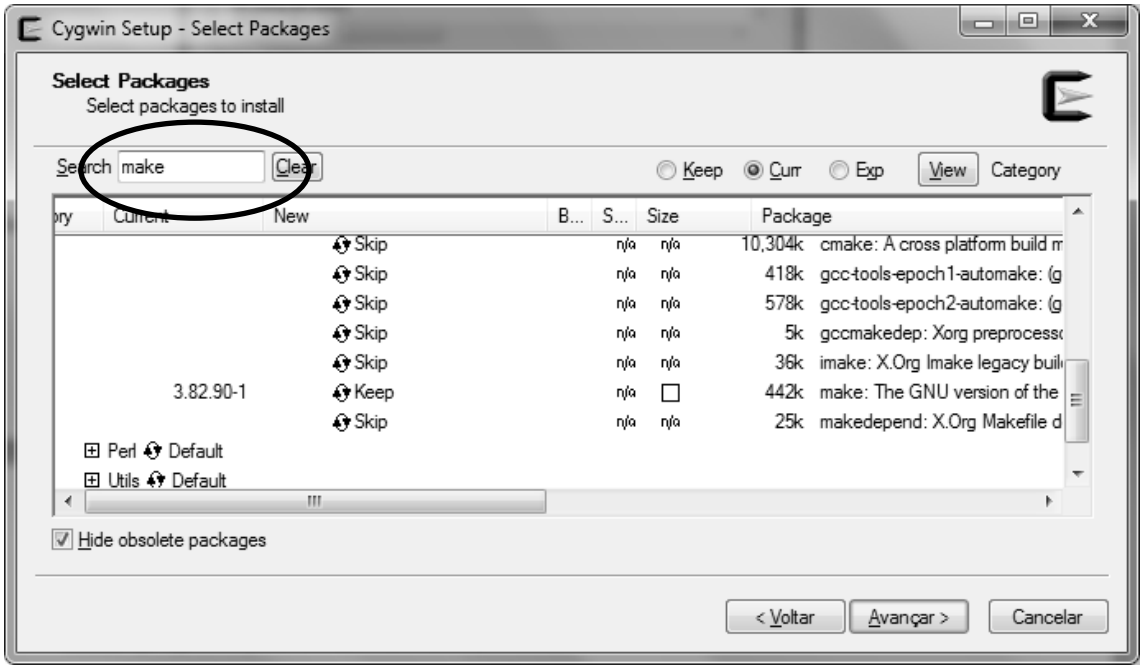

Figura 28 – Instalador do *Cygwin* (Tela 2)

Para instalar a ferramenta, é preciso clicar na opção *skip* até que ela se torne o número da versão do programa, indicando que ele não será ignorado durante a instalação do *cygwin*. O mesmo procedimento de busca e seleção deve ser repetido para todas as ferramentas. É importante verificar o conteúdo em *package* para saber qual ferramenta será instalada. O conteúdo na coluna *package* para cada ferramenta é:

-make (Que está sob a guia Devel): *The GNU version of the make utility*

-nasm: *The Netwide Assembler*

-mkisofs (Que está sob a guia Utils): *Create ISO filesystem images*

Após isso, basta clicar em avançar e aguardar o fim da instalação do *cygwin*.

Cross-Compiler

Um *cross-compiler* (ou compilador cruzado) é um compilador que funciona em uma plataforma e gera código executável para outra plataforma. As plataformas podem ou não diferir em arquitetura de CPU ou formato de arquivo executável (ELF, PE, A.OUT, etc..). Os arquivos executáveis devem obrigatoriamente estar no formato ELF, pois atualmente este é o único formato para o qual o FESO possui suporte. O sistema operacional Windows não utiliza este formato, então é natural que os compiladores baseados em Windows não deem origem a programas-objeto ELF. Por isso, se faz necessário o uso de um *cross-compiler* que crie arquivos no formato suportado pelo SO.

O sistema operacional Linux, suporta nativamente o formato ELF. De forma que o compilador GCC, o *linker* ld e NASM podem trabalhar com arquivos ELF. Porém, versões do SO e dos programas criadas com ferramentas e versões do GCC presentes no UBUNTU apresentaram um funcionamento inconsistente. Possivelmente, os compiladores nativos criam código que possui alguma dependência ao sistema operacional no qual ele está funcionando. Por isso, é mais aconselhável o uso de uma versão *cross-compiler* do GCC ao invés da utilização de uma versão nativa em alguma distribuição do Linux.

É possível baixar o código fonte do GCC (GNU *compiler collection*) e compilar uma versão dele que gere código para uma nova plataforma (Há um tutorial disponível neste link: http://wiki.osdev.org/GCC_Cross-Compiler). Outra solução mais simples é baixar e executar o instalador do *cross-compiler* disponível no repositório indicado no começo deste apêndice.

O instalador simplesmente descompacta uma série de diretórios (no local indicado pelo usuário durante a instalação). Um desses diretórios é o *bin*, que contém os arquivos binários das ferramentas necessárias para compilar e realizar a *linkedição* do sistema operacional. Essas ferramentas possuem nomes no seguinte formato: i586-elf-(nome da ferramenta). De forma que o GCC *cross-compiler* é o i586-elf-gcc.exe.

Trabalhando com o *Cygwin* e o *Cross-compiler*

Dentro dos arquivos de código fonte do SO, existem scripts de compilação e *makefiles* utilizados para compilar o SO. Esses arquivos procuram por padrão o compilador no seguinte diretório: "/usr/local/cross/bin/". Esse é o caminho de diretório no formato utilizado pelo sistema operacional Linux. Isso é necessário, pois o *cygwin* simula o sistema de arquivos do Linux em sua interface textual. Para tanto, ele cria um diretório em c:\cygwin, no sistema de arquivos do Windows. Dentro dele, existem diversos diretórios comuns aos sistemas Linux (como bin, usr, var, lib).

Esses diretórios são acessados internamente pelo *cygwin*, utilizando a barra (/) como separador de diretório (ao invés da contra barra (\) padrão do Windows), e tendo como diretório raiz a própria pasta do *cygwin* (c:\cygwin). Desta forma, se um arquivo, por exemplo, chamado de "foo.txt" é criado em: c:\cygwin\usr. Ele pode ser acessado de dentro do *cygwin* pelo caminho /usr/foo.txt. O *cygwin* oferece outra forma para acessar o sistema de arquivos padrão do Windows, através do *cygdrive*. Por exemplo, caso seja necessário acessar

a partir do *cygwin* um arquivo no disco local do Windows, que esteja fora da pasta c:\cygwin, isto pode ser feito através do caminho: /cygdrive/c. Este caminho acessa o disco local do Windows, e permite acesso aos arquivos armazenados nele.

Para evitar a necessidade de alterar os arquivos *makefile* presentes nos repositório, é necessário que todos os arquivos que são criados pelo instalador do *cross-compiler* estejam na pasta: c:\cygwin\usr\local\cross. Caso a pasta *cross* não exista, ela deve ser criada. Basta descompactar os arquivos nesta pasta, ou copia-los posteriormente, de forma com que fiquem acessíveis por esse caminho.

Compilando o kernel e criando um arquivo ISO

Alguns detalhes sobre a estrutura dos arquivos do SO são necessários neste ponto. O arquivo RAR que é obtido no repositório contém os seguintes diretórios: Desenvolvimento (com o código fonte do SO, Drivers e Aplicativos de usuário), *isofiles* (que é a pasta transformada em imagem inicializável pelo *mkisofs*) e Grub (com alguns arquivos do GRUB).

Dentro da pasta Desenvolvimento, existem mais 4 pastas: *kernel* (com o código fonte do kernel), *dev* (com os arquivos fonte dos drivers de dispositivo), *app* (com alguns aplicativos de usuário) e *lib* (com a biblioteca básica para desenvolvimento e as funções para chamadas ao sistema). A pasta desenvolvimento contém também dois arquivos chamados: "compilar_os", um com a extensão BAT, e outro com a extensão BS. O arquivo BAT (arquivo em lotes) pode ser executado diretamente no sistema operacional Windows, e sua função é executar o *cygwin* passando por parâmetro o conteúdo do arquivo com a extensão .BS. Esses arquivos contêm scripts que navegam através do sistema de arquivos oferecido pelo *cygwin*, abrindo cada uma das pastas e chamando o comando *make*, responsável por compilar os arquivos fonte. Ao final, os scripts copiam todos os arquivos objeto gerados para a pasta *isofiles*. Depois, o *mkisofs* é chamado para transformar o conteúdo da pasta em um

arquivo ISO. Esse arquivo ISO pode ser utilizado em uma máquina virtual para testes, ou mesmo pode ser gravado em um CD, e utilizando para inicializar o computador. Dentro do ambiente do *cygwin*, a sintaxe para navegar entre diretórios, criar, remover e executar arquivos é igual à sintaxe utilizada no *shell* dos sistemas Linux. De forma que os scripts utilizam esse tipo de comando.

Os *scripts* contidos no repositório funcionam a partir da pasta: c\tcc\desenvolvimento, que é o local original onde esses arquivos foram gerados pela primeira vez. Caso esse caminho seja alterado, o arquivo compilar_os.bs deve ser alterado de acordo com o novo local.

Para compilar o SO, é preciso que o arquivo compilar_os.bs esteja configurado com os caminhos corretos, assim como, com todas as ferramentas instaladas. Os comandos presentes no arquivo compilar_os.bs se resumem a navegar até a pasta onde estão os arquivos fonte do *kernel*, e de cada driver de dispositivo e aplicativo de usuário. Após isso, o comando *make* é executado, utilizando as instruções contidas no arquivo *makefile* para compilar e gerar os arquivos binários. Em seguida, os arquivos binários são copiados para a pasta *isofiles*, e então o programa *mkiso* é chamado, criando uma ISO inicialzável a partir desta pasta. Caso os caminhos estejam corretamente especificados, basta executar o arquivo compilar_os.bat para compilar o *kernel* do SO e gerar uma ISO em c:\bootcd.iso.

Compilando um novo aplicativo e adicionado à lista de módulos do GRUB

Dentro da pasta *default*, e consequentemente na nova pasta *default*, que é copiada para criar um novo projeto, existe um arquivo *makefile*, que deve ser editado para a geração correta do programa. Primeiramente, a linha iniciada por PATH_LIB :=, contém o caminho para a pasta *lib*. Esse caminho é colocado em /cygdrive/c/tcc/Desenvolvimento/lib por padrão. Caso a pasta *lib* esteja em outro diretório, é necessário alterar esse caminho para que este

aponte para o local correto, como é possível verificar através da figura 29. Como o arquivo *makefile* é utilizado internamente pelo *cygwin*, os caminhos devem estar no formato do sistema operacional Linux.

```
#########################
PATH_ROOT    := $(shell PWD)
PATH_INC     := $(PATH_ROOT)/include
PATH_BIN     := $(PATH_ROOT)/bin
PATH_LIB     := /cygdrive/c/tcc/Desenvolvimento/lib
PROJDIRS     := $(PATH_ROOT) $(PATH_INC) $(PATH_BIN) $(PATH_LIB)
CSRCFILES    := $(shell find $(PROJDIRS) -type f -name "*.cpp")
ASMSRCFILES  := $(shell find $(PROJDIRS) -type f -name "*.asm")
HDRFILES     := $(shell find $(PROJDIRS) -type f -name "*.h")
COBJFILES    := $(patsubst %.cpp, %.o, $(CSRCFILES))
ASMOBJFILES  := $(patsubst %.asm, %.o, $(ASMSRCFILES))

#alterar nome do executável aqui
EXECUTABLE   := $(PATH_BIN)/

# FERRAMENTAS
#########################
```

Figura 29 – Makefile (Parte1)

Outra alteração necessária é a definição do nome do arquivo executável que será criado. Para isto, é preciso editar a linha iniciada por EXECUTABLE :=, adicionando após a barra o nome do arquivo binário que deve ser gerado na pasta bin. Por padrão, o *makefile* referencia as ferramentas que serão utilizadas para a compilação da forma apresentada na figura 30.

```
# FERRAMENTAS
#########################
NASM    =nasm -f elf
C++     =/usr/local/cross/bin/i586-elf-c++
LD      =/usr/local/cross/bin/i586-elf-ld
RM      =rm
```

Figura 30 – Makefile (Parte2)

Tanto C++ (que aponta para o compilador c++), quando LD (que aponta para o *linker*) fazem referencia aos arquivos do *cross-compiler* que por padrão ficam em /usr/loca/cross/bin. Caso o local seja alterado, todos os arquivos *makefile* do projeto devem ser alterados. Vale notar que tanto o *kernel*, como cada driver de dispositivo e programa de usuário possui seu próprio arquivo *makefile*.

Existem duas formas de acessar o programa a partir do FESO. Caso o SO esteja executando em uma máquina com suporte a drivers IDE, basta que os arquivos estejam na imagem do disco, desta forma, eles podem ser acessados através da pasta /cdrom, dentro do sistema de arquivos do FESO. Caso não haja driver IDE, é necessário configurar o GRUB para carregar o arquivo executável, editando o arquivo de configuração localizado em: (caminho para *isofiles*)\boot\grub\menu.lst. A figura 31 mostra um exemplo de arquivo de configuração do GRUB.

```
default=0
timeout=30
title feSO
foreground bbbbbb
background 000000
kernel /boot/kernel.bin root=/dev/hda2
module /drivers/timer.o -DRIVER
module /drivers/ide.o -DRIVER
module /drivers/console.o -DRIVER
module /drivers/iso9660.o -DRIVER
module /app/teste.o -APP
module /app/editor.o -APP
module /app/doc.txt -APP
module /app/shell.o -APP_INI
module /app/idle.o -IDLE
module /drivers/tramp.o -TRAMP
```

Figura 31 – Arquivo menu.lst

Este arquivo possui diversas linhas que começam com a palavra module. Cada uma delas informa qual módulo deve ser carregado pelo GRUB durante a inicialização do sistema operacional. Para cada programa criado, deve existir uma nova linha dentro desse arquivo com o seguinte formato: module (caminho do arquivo dentro do CD, que é igual ao caminho do arquivo dentro da pasta *isofiles*) -(*flag*).

O sistema operacional reconhece alguns *flags* básicos para o carregamento do arquivo. O *flag* -DRIVER, indica que o módulo é um driver de dispositivo que deve ser iniciado junto com o SO com um nível de privilégio maior. O *flag* -APP, indica que é um aplicativo normal, que deve apenas ser criado e não executado. O *flag* -APP_INI é utilizada para indicar um aplicativo de usuário normal, que deve ser iniciado junto com o SO. Outros *flags* como -IDLE e -TRAMP são para indicar programas com funções específicas para o sistema operacional.

Após copiar a pasta, alterar o arquivo *makefile* e *menu.lst*, desenvolver e copiar todos os arquivos fonte e cabeçalhos em suas devidas pastas, o novo programa estará pronto para ser compilado. Para tal, os seguintes comandos devem ser executados no *cygwin*.

-cd (caminho para pasta criada para o novo programa)

-cp (caminho para o arquivo binário recém compilado) (caminho para pasta isofiles)

-mkisofs -R -b boot/grub/stage2_eltorito -no-emul-boot -boot-load-size 4 -boot-info-table -o (caminho para criação da iso)\nome_da_iso.iso (caminho para pasta isofiles)

O primeiro comando serve para navegar até pasta criada para o programa. O segundo comando (*make*), executa o arquivo *makefile* com as especificações para criar o arquivo binário. O terceiro comando copia o novo arquivo binário para a pasta *isofiles*. É necessário especificar um subdiretório dentro da pasta *isofiles*, porque o conteúdo dela será o mesmo da imagem do sistema.

O caminho para os módulos no GRUB deve estar de acordo com o caminho indicado para a cópia do arquivo binário dentro dessa pasta. Por exemplo, se um arquivo foo.o é gerado

na pasta foo/bin de um programa executável, esse arquivo pode ser copiado para a pasta isofiles/app/foo.o. De forma que o arquivo menu.lst deve conter uma linha do tipo:

Module /app/foo.o -APP

O último comando transforma o conteúdo da pasta isofiles em uma imagem inicializavel. Dentro dessa pasta, devem conter todos os arquivos dos drivers de dispositivos, assim como todos os programas de usuário. Além disso, essa pasta contém os arquivos do GRUB, que são utilizados para inicializar o computador e carregar o *kernel* e os módulos.

Após isso, um novo arquivo ISO será criado e poderá ser utilizado para o boot em máquina real ou virtual. Após o boot, é possível acessar a pasta /módulos dentro do SO e chamar o comando exec (nome do programa) para ver o aplicativo funcionando no SO.

8 APÊNDICE B – DIAGRAMA GLOBAL

Este apêndice contém um diagrama global do funcionamento do sistema FESO mostrado na figura 32.

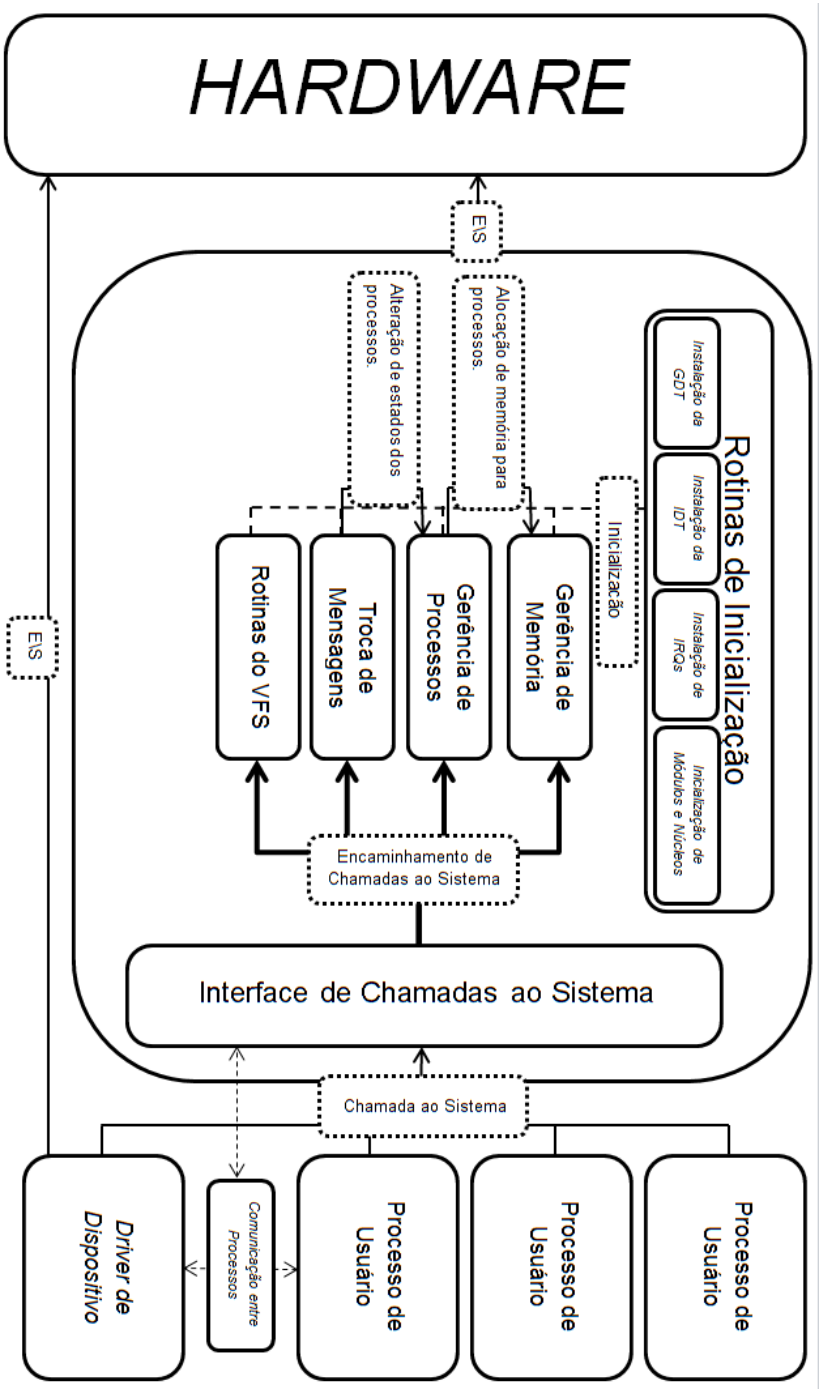

Figura 32 - Diagrama Global do sistema FESO