feSO : Um Sistema Operacional Para Fins Educativos

Hermano L. S. Lustosa [1], **Frederico Luis Cabral**[1]

[1]Centro Universitário Serra dos Órgãos (UNIFESO)
Teresópolis – RJ – Brasil

Abstract. *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 operation of this kind of system. The feSO operating system, presented in this work, was developed in order to provide the students with a simplified and functional OS, which could be used as a learning tool. In this work, we describe the main aspects regarding structure and archictecture of feSO, enabling professors and students to study it and modified it as they please*

Resumo. *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 feSO foi desenvolvido no intuito de fornecer aos estudantes um SO simplificado e funcional, que possa ser usado como ferramenta de aprendizado. Neste artigo, são descritos aspectos principais da arquitetura e estrutura do sistema feSO, permitindo que professores e alunos possam estudá-lo e modificado de acordo com suas necessidades.*

1. Introdução

O entendimento de como um sistema operacional funciona é importante para se compreender como um computador funciona, uma vez que o SO (Sistema Operacional) é um componente chave de vários sistemas computacionais. As técnicas empregadas na construção dos sistemas operacionais, como a gerência de recursos, o uso de abstrações inteligentes e a programação em baixo nível podem ser aplicadas no desenvolvimento de outros tipos de sistemas [POLZE 2006]. Portanto, não é exagero dizer que um bom entendimento sobre o funcionamento de sistemas operacionais é de fundamental importância para alunos de computação e profissionais da informática.

Entretanto, dada a natureza de sua função, sistemas operacionais se tornam grandes e complexos, fazendo com que o estudo de seu funcionamento muitas vezes seja limitado à teoria. Os estudantes podem facilmente ter acesso à implementações concretas de sistemas *open-source*. Porém, SOs comerciais abertos, como as versões do UNIX e as distros do LINUX têm uma base de código muito extensa. Por exemplo, apenas o núcleo do sistema Linux possui mais de 20 milhões de linhas de código [LOHNER 2018]. Dado este tamanho considerável, a tarefa de entender o funcionamento de um sistema operacional com base no estudo do seu código fonte é bastante dificultada.

Podemos adicionar ainda outros agravantes, como o fato destes sistemas serem portáveis para diversas plataformas, compatíveis com as rígidas especificações POSIX [IEEE 2018] e altamente otimizados. É de se esperar que, com a necessidade de atender às idiossincrasias de diversas arquiteturas de hardware e aos exigentes requisitos funcionais

e de desempenho, as bases de código destes sistemas se tornem muito complexas e de difícil compreensão, sobretudo para estudantes de graduação.

Por este motivo, e pela dificuldade de se encontrar sistemas de apoio e materiais didáticos de cunho mais prático em língua portuguesa, propomos um sistema operacional para fins educativos chamado de feSO. O feSO (Figura 1) é um sistema operacional multitarefa, com suporte a processadores de arquitetura Intel x86. Ele foi desenvolvido para ser uma ferramenta de aprendizado utilizável por estudantes e professores, oferecendo funcionalidades básicas principais de um sistema operacional, sem as preocupações com requisitos de desempenho e compatibilidade com padrões existentes. Por isso, o sistema operacional pode ser mais simples, e permitir a direta aplicação dos conceitos teóricos apresentados na literatura de sistemas operacionais lecionadas nos cursos de graduação. No caso, o sistema feSO possui apenas algumas milhares de linhas código, uma ordem de magnitude a menos que os SOs comerciais atuais.

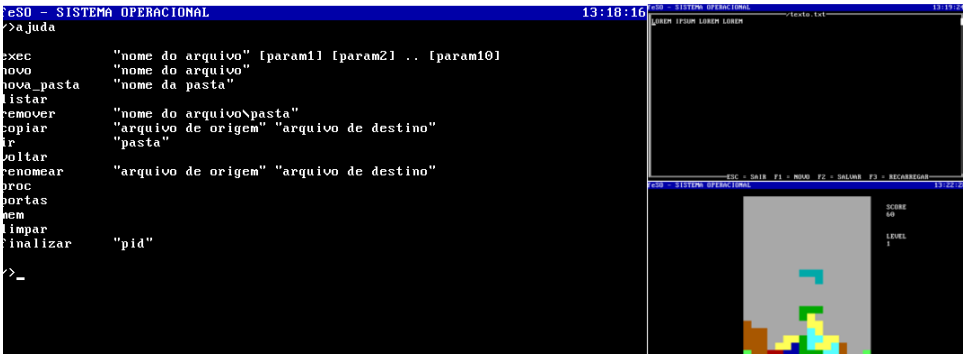

Figura 1. Telas do sistema FESO: Shell, Editor de Texto e o aplicativo do jogo Tetris.

Para apresentar o sistema feSO, organizamos este trabalho da seguinte maneira. Na seção 2 apresentamos os trabalhos relacionados. Na seção 3 descrevemos a metodologia utilizada no desenvolvimento do feSO. Na seção 4, tratamos da arquitetura do sistema feSO, dando uma visão geral sobre sua estruturação. Na seção 5 descrevemos o funcionamento interno dos principais componentes do núcleo do sistema. Na seção 6 temos uma discussão referente às possíveis aplicações do feSO em um curso de sistemas operacionais. Finalmente, na seção 7 temos as conclusões e considerações finais.

2. Trabalhos Relacionados

Atualmente, existem diversas propostas de ferramentas para o auxílio ao ensino de sistemas operacionais. Um dos sistemas mais conhecidos para este fim é o sistema MINIX [HAYS 1989]. Inicialmente, o sistema MINIX servia como uma ferramenta de ensino, mas atualmente se trata de um SO flexível e robusto, sendo o principal proponente da arquitetura microkernel em desenvolvimento. Em sua versão atual, o MINIX 3 conta, no total, considerando o núcleo e os principais servidores, com mais de 3,5 milhões de linhas de código. Portanto, é justo dizer que já se trata de um sistema de complexidade considerável e de mais difícil compreensão.

Em meio aos diversos repositórios de código atualmente existentes na web, e também dentro das comunidades de desenvolvedores de SOs, como a OSDev Wiki

[OSDEV 2018], encontramos diversas alternativas de sistemas operacionais simplificados para o aprendizado. Exemplos disso são os sistemas IntermezzOS [WILLIAMS 2018] e MiniOS [ROMAN OTERO 2015]. Esses sistemas têm a proposta de funcionar como um tutorial de desenvolvimento, permitindo que estudantes implementem passo-a-passo as principais estruturas de um SO. São alternativas interessantes e viáveis, porém, com material de apoio disponível apenas em língua inglesa. Esses sistemas também podem exigir bastante tempo de alunos e professores durante o acompanhamento dos tutoriais e das implementações propostas. E em alguns casos, como no IntermezzOS, eles são baseados em linguagens de programação menos comuns, como o Rust. Um outro sistema voltado ao ensino de SOs é o Xv6 [COX 2018], também com material de apoio em inglês, e acessível a estudantes interessados em aprender mais sobre os meandros de sistemas do tipo UNIX.

Em língua portuguesa, temos ferramentas como o SOsim [MAIA 2018], uma aplicação desenvolvida para ilustrar diversos conceitos referentes a sistemas operacionais, e feita para ser utilizada em conjunto com a obra Arquitetura de Sistemas Operacionais [MACHADO 2013]. Entretanto, trata-se apenas de um simulador, não sendo assim, um sistema operacional real, como os apresentados anteriormente.

O sistema feSO proposto neste trabalho se difere das alternativas citadas em dois pontos principais. Além de ser um SO de fato e não um simulador, ele é uma implementação original não derivada de sistemas UNIX ou LINUX, como é o caso de grande partes das ferramentas existentes. Embora sistemas UNIX sejam amplamente utilizados, os autores deste trabalho acreditam que é interessante começar o estudo dos SOs em um nível ainda mais fundamental, explorando os conceitos básicos de forma mais geral antes de partir para o estudo de ferramentas reais como os sistemas UNIX. O segundo ponto que difere nosso trabalho dos demais está relacionado ao material de apoio associado estar em português, assim como os comentários e a base de código. Esse aspecto é um facilitador para os estudantes brasileiros que porventura ainda não dominem a língua inglesa.

3. Desenvolvimento

O sistema feSO foi desenvolvido com a utilização de técnicas já bem estabelecidas e documentadas na literatura sobre sistemas operacionais, como algoritmos de escalonamento baseado em filas de prioridades, uso de *bitmaps* para controle de alocação de memória, suporte a primitivas de comunicação entre processos *send* e *receive*, entre outras. O sistema feSO suporta apenas processadores da arquitetura Intel x86 e foi desenvolvido com a utilização das linguagens C/C++ em conjunto com alguns trechos de código em *assembly*.

Como o C e o C++ são linguagens de alto nível, que raramente levam em conta características especiais da máquina na qual seu programa irá funcionar [HYDE 2003]. Faz-se, naturalmente, necessário o uso da linguagem *assembly* para a execução de instruções específicas do hardware sendo utilizado.

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 no desenvolvimento do feSO foram o GCC (GNU compiler collection, versão 4.9.1) e o NASM (netwide assembler, versão 2.11.08). Porém, alguns cuidados especiais são utilizados na compilação dos arquivos fonte do sistema.

O sistema feSO pode ser alterado e compilado em sistemas Windows e Linux, porém, é necessário utilizar 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 [FOUNDATION 2018], no caso, o compilador GCC executa sobre um SO hospedeiro (Linux ou Windows), mas gera um código independente da plataforma atual. Isto é necessário, porque nenhum recurso ou biblioteca da linguagem C/C++ que exija suporte de sistema operacional pode ser utilizado no código fonte do feSO. Isto corre pois, este suporte não existirá nativamente para o feSO quando o mesmo estiver em execução. 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 embutida no código fonte do próprio sistema.

O desenvolvimento do sistema operacional se deu a partir de seu núcleo (*kernel*) que é composto pelas rotinas mais importantes. A ordem do desenvolvimento dos componentes se deu de acordo com suas dependências. Uma vez que o núcleo era capaz de oferecer um conjunto suficiente de funcionalidades, novos programas foram criados para funcionar a partir dele. À medida que novas funções eram implementadas no núcleo, é preciso desenvolver algum aplicativo para testá-la. Esses testes são realizados com a utilização de uma máquina virtual baseada no *hypervisor* Virtual Box (versão 5.0), e em alguns casos, em hardware real.

4. Arquitetura

Os sistemas operacionais 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. Para criar esse tipo de sistema é necessário que todos os seus módulos sejam compilados e unidos (*linkados*) em um único arquivo-objeto.

Os micronúcleos ou *microkernels*, por sua vez, são uma forma diferente de arquitetura de sistema operacional. A ideia por trás dos micronúcleos é de manter no *kernel* apenas algumas funcionalidades principais. Nesta abordagem, os *drivers* de dispositivos funcionam fora do *kernel* e são baseados em uma arquitetura cliente-servidor. O sistema operacional deve fornecer mecanismos de comunicação entre processos para permitir que os programas de usuário se comuniquem com os servidores e acessem os recursos de hardware. Como este processo de comunicação exige muitas trocas de contexto para execução de chamadas ao sistema, *microkernels* foram considerados historicamente mais lentos que *kernels* monolíticos [CHEN 1993]. Portanto, implementar sistemas operacionais com essa arquitetura que tenham desempenho comparável ao de sistemas monolíticos é um grande desafio.

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 2000]. Em seu artigo sobre a construção de *microkernels*, Liedtke [LIEDTKE 1995] lista algumas vantagens desse tipo de abordagem. Entre elas, o isolamento dos servidores e o *kernel*. E a possibilidade da coexistência de diferentes estratégias e interfaces para comunicação com os servidores.

O sistema feSO foi construído utilizando diversos conceitos presentes na arquitetura *microkernel*, uma vez que seu objetivo é mais didático e ilustrativo do que puramente obter desempenho compatível com sistemas comerciais. Desta forma, os *drivers* de dispositivos estão isolados do núcleo do sistema, e funcionam de forma similar a programas de usuário, porém com uma maior liberdade para executar instruções de entrada e saída.

Do ponto de vista do isolamento entre camadas de software, o sistema feSO se utiliza dos recursos de proteção de hardware fornecidos pelos processadores da família Intel x86. Os mecanismos de proteção do processador reconhecem 4 níveis de privilégio, numerados de 0 a 3 [INTEL 2007]. Quanto menor o nível em que estiver executando, mais privilégios o processo tem. O núcleo do sistema operacional feSO 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, porém impedidos de executar instruções privilegiadas. No nível 3 (o nível 2 não é utilizado no feSO, pois só é necessário distinguir entre três níveis de prioridade) estão os aplicativos de usuário, que não podem executar instruções privilegiadas, nem de entrada e saída de dados, devendo para tanto realizar chamadas ao sistema.

A Figura 2 contém uma visão geral da arquitetura do sistema feSO. À direta temos os processos de usuário, que podem se comunicar entre si ou com os servidores dos *drivers* de dispositivo (também externos ao *kernel*) através do sistema de comunicação entre processos. Os processos se comunicam entre si e com o sistema operacional através da interface de chamadas ao sistema. Ao centro da figura, temos o bloco que representa o núcleo com seus principais módulos de inicialização e controle, que contém as rotinas mais críticas do sistema.

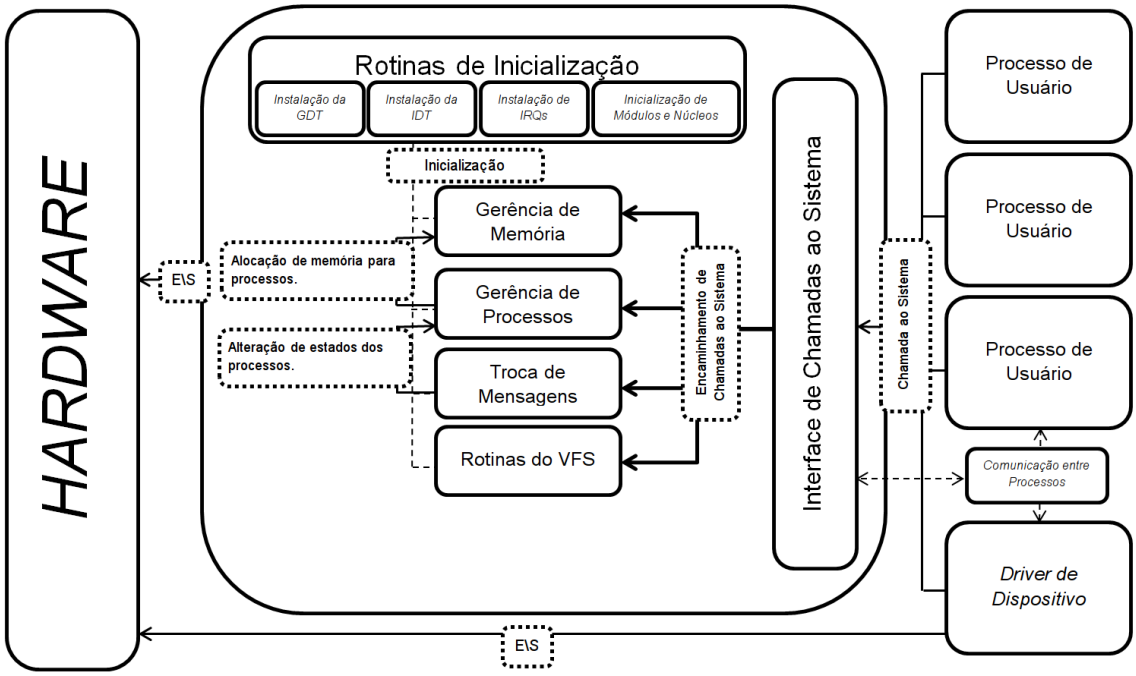

Figura 2. Arquitetura do sistema feSO.

5. O Núcleo

O núcleo ou *kernel* de um sistema operacional é composto por suas rotinas mais importantes, e que tem permissão para executar as instruções privilegiadas da CPU. No sistema feSO, o núcleo tem 4 grandes componentes principais. São eles: gerência de memória, gerência de processos, sistema de troca de mensagens e sistema de arquivos virtual. Nesta seção, são descritos os detalhes referentes a esses principais componentes.

5.1. Processo de Inicialização

O sistema feSO utiliza o GNU GRUB (GRAND UNIFIED BOOTLOADER, versão 0.97) como seu *boot loader*. O *boot loader* é executando assim que o computador é inicializado, sendo responsável pelo carregamento e transferência de controle para o *kernel* do sistema operacional [FOUNDATION 2011]. 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. Estes módulos podem ser ou *drivers* de dispositivos ou aplicações de usuário. 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.

Uma vez com o controle sobre a máquina, cabe ao sistema operacional executar todas as tarefas necessárias a sua inicialização. Primeiramente, o sistema feSO utiliza as estruturas deixadas pelo GRUB para inicializar a gerência de memória. Assim que o controle da memória física entra funcionamento, e a memória virtual esteja habilitada, o *kernel* começa a carregar as tabelas de descritores da CPU.

A GDT (*Global Descriptors Table*) é uma tabela de descritores 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 2007]. 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 determina o nível de privilegio do código, graças aos dados contidos na GDT.

Outra tabela de descritores importante criada durante o processo de inicialização é a IDT (*Interrupt Descriptors Table*). A IDT armazena os descritores de porta (*gate descriptors*) que fornecem o acesso aos manipuladores de exceções e interrupções [INTEL 2007]. 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 trata-la.

Em seguida, o SO analisa a estrutura deixada pelo GRUB na memória a fim de colocar os módulos em execução. Uma vez que todo esse processo esteja concluído, o sistema operacional habilita as interrupções, e coloca a CPU em estado de espera. Quando uma interrupção ocorrer, uma rotina do SO será chamada pelo processador, e o algoritmo de escalonamento será posto automaticamente em execução. A partir deste momento, todos os programas, tanto servidores quando aplicativos de usuário (carregados como módulos do GRUB) estarão em funcionamento.

5.2. Processo de Inicialização com Múliplos Núcleos

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, ou mesmo, que um programa seja dividido em partes e executado por diversas CPUs em simultâneo. O sistema feSO tem suporte a processadores da família x86 com múltiplos núcleos. Embora não exista um limite estabelecido para o número de núcleos suportados, o sistema feSO foi testado em hardware real com máquinas contendo processadores Intel de até 4 núcleos reais e 4 núcleos virtuais.

Durante o processo de inicialização, o feSO faz a detecção da existência de múltiplos processadores. Após a detecção, o sistema pode inicializar os processadores através do envio de IPIs (Inter-processor interrupts) ou interrupções entre processadores, que funcionam como um sistema de troca de mensagens ou sinalização entre núcleos em um mesmo processador. Sempre que um computador com múltiplos processadores é iniciado, um dos processadores é selecionado pela BIOS (ou pelo *firmware* responsável pelo *boot*) 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 na máquina.

O sistema feSO carrega previamente em um espaço no inicio da memória, um pequeno programa responsável por configurar corretamente os processadores. O endereço do ponto de entrada desse programa é enviando junto com as IPIs para cada um dos APs. Ao receber a IPI o *application processor* de destino executa a partir da posição recebida junto com a interrupção, ficando corretamente configurado após a execução do código trampolim. Cada AP possui seu próprio circuito temporizador que gera interrupções em um intervalo de tempo configurável. Sempre que esse temporizador gera uma interrupção, o AP executa a rotina de escalonamento de processos do sistema feSO, fazendo com que algum programa pronto na fila de escalonamento seja alocado ao AP e então executado.

5.3. Gerência de Memória

Segundo Stallings [STALLINGS 2010], alguns dos requisitos que devem ser atendidos por um sistema de gerência de memória de um sistema operacional são: Proteção, Compartilhamento e Organização física e lógica.

A gerência da memória do feSO funciona em 3 níveis básicos. 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 deste, existem os algoritmos de alocação de memória, que funcionam em espaço de usuário, e se utilizam de uma chamada ao sistema para aumentar a quantidade de memória utilizada pelos processos.

Começando pelo nível mais básico, a gerência de memória física funciona da seguinte maneira. Após o processo de inicialização do sistema, uma área de memória, posterior à imagem do sistema operacional é reservada. Está área, cujo tamanho depende do tamanho da memória real disponível, é utilizada para armazenar um mapa de bits utilizado no controle da memória física. Cada bit no mapa representa um *frame* de memória

de 4KB como ilustra a Figura 3. 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 livre.

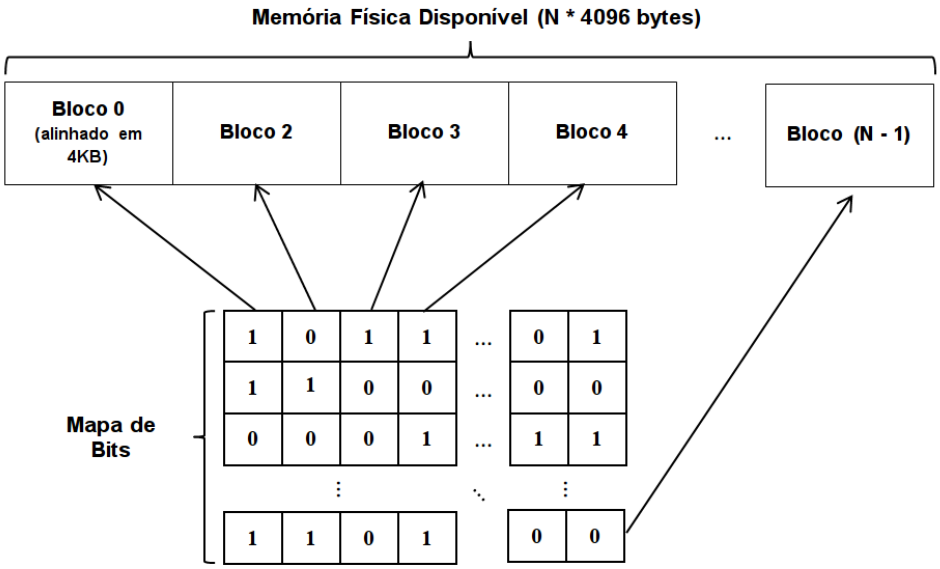

Figura 3. Mapa de bits para gerência de memória física.

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 do endereço de memória iguais a 0. Como cada bloco tem 4KB, todos eles começaram 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 retornando. É possível saber o endereço inicial do *frame*, com base na posição do bit dentro do mapa.

No segundo 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 ainda 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 x86 é de 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 contíguas 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.

Nos processadores da família Intel x86, o mapeamento de endereços virtuais em físicos é feito através do uso de dois níveis de tabelas de páginas, como demonstra a Figura 4. 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 *frame* de 4KB de memória física. O mapeamento de endereços virtuais em físicos é feito da seguinte forma. Os 10 bits mais significativos dos endereços virtuais de 32 bits representam um índice na *Page Directory* e 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.

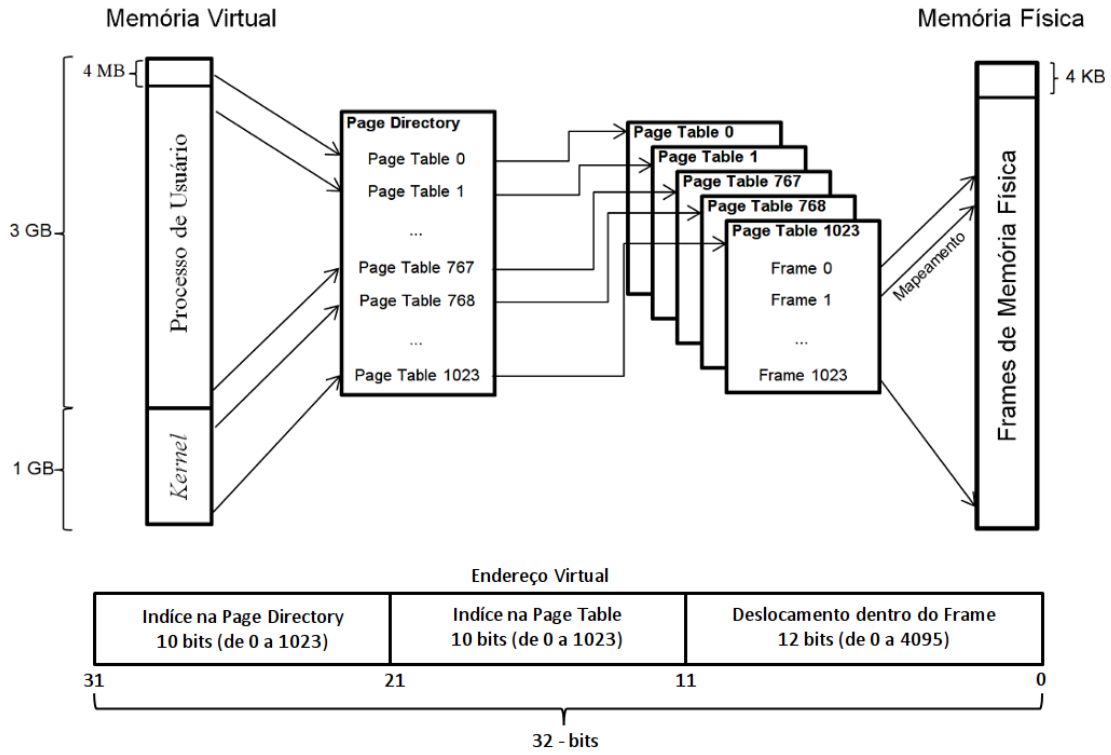

Figura 4. Estrutura de páginas para a gerência de memória virtual.

Como boa parte dos programas de computador são criados de maneira que a quantidade total de memória necessária em sua execução não é definida durante seu desenvolvimento, é necessário que existam mecanismos que permitam que um processo aloque

mais memória. 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* necessita de mais memória, é preciso aumentar o *heap*, e retornar o endereço do primeiro byte de memória da área recém-expandida. A função *malloc* utiliza uma lista encadeada com blocos livres de memória como é possível ver na Figura 5. 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. A função *free* faz o oposto e libera um bloco de memória previamente alocado pelo programa.

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 nenhum bloco livre de tamanho adequado é encontrado. Neste caso, o programa solicita um novo bloco de memória ao sistema operacional. Quando isto ocorre, potencialmente novos blocos de memória física são alocados e mapeados para o espaço de endereçamento do processo solicitante, e um ponteiro para essa nova região de memória é retornado pelo sistema. Este novo bloco de memória é então adicionado à lista de blocos livres, gerenciada pela função *malloc* e utilizado pelo programa em execução.

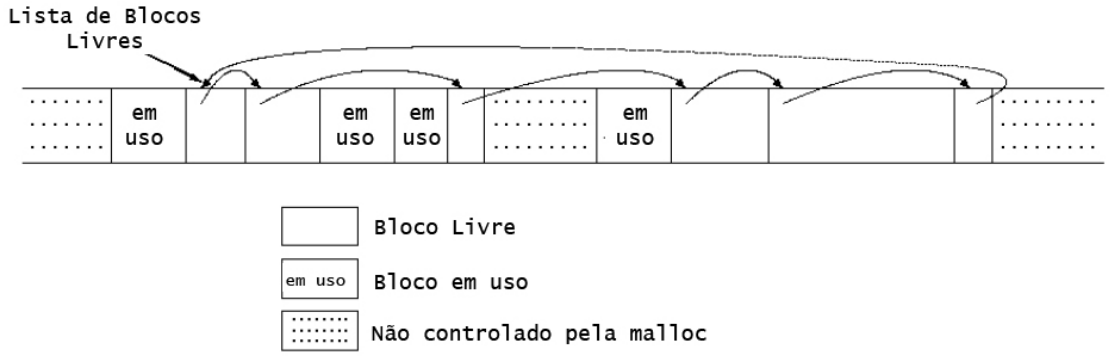

Figura 5. Estrutura mantida em espaço de usuário pela função malloc [KERNIGHAN 1988]

5.4. Gerência de Processos

Um processo pode ser considerado um programa em execução [SILBERSCHATZ 2000], muito embora, o programa em si seja apenas o conjunto de instruções, e o processo seja uma instância daquele programa na memória do computador. 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 utilizados e outras informações utilizadas no controle do processo.

O sistema feSO suporta *threads* em modo *kernel*. Neste caso, o SO tem conhecimento da existência das *threads* e as utiliza como unidade de alocação da CPU. Cada

processo possui uma lista de *threads* associados, cada *thread* possui sua própria estrutura com conjunto de estados dos registradores. As *threads* em um processo podem estar executando trechos de código diferentes entre si, porém todos compartilham o mesmo espaço de endereçamento, e as mesmas variáveis globais. Entretanto, existe uma pilha associada a cada *thread*, na qual são armazenadas as variáveis locais e estruturas de controle para chamadas de sub-rotinas.

A Figura 6 mostra como está dividido o espaço de endereçamento virtual de um processo. No começo do espaço de endereçamento, ficam o código e os dados do programa. Imediatamente após os dados, começa o *heap*, posição onde ficam mapeados os *frames* de memória alocados dinamicamente pelo programa e mantidos pela função *malloc*. 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.

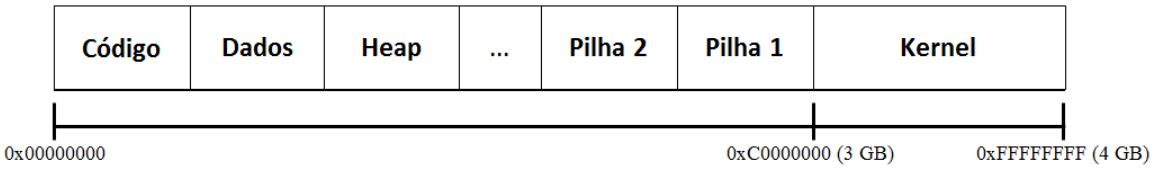

Figura 6. Espaço de endereçamento de um processo no sistema feSO.

O sistema feSO fornece um conjunto de chamadas ao sistema que permitem a realização de operações de criação e remoção de processos e *threads*. A criação de processos é feita através do carregamento de arquivos executáveis no formato ELF (*Executable and Linkable Format*)[TIS 1995]. Como o carregamento de bibliotecas compartilhadas ainda não é suportado, é preciso que o arquivo executável esteja com todas as suas bibliotecas estaticamente *linkadas* em um único arquivo binário. Quando um processo é criado, ele tem uma única *thread* principal associada. Novas *threads* podem ser criadas dentro de um processo através de uma chamada ao sistema.

Como o número de total de *threads* no sistema na maioria dos casos é maior do que a quantidade de CPUs disponíveis, 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, também conhecido como *round-robin*, para escalonar as CPUs para os diversos *threads* presentes no sistema. A Figura 7 mostra um exemplo de execução do algoritmo de escalonamento, bem como um diagrama das possíveis trocas de estado entre *threads*.

Existem basicamente 2 tipos de *threads* presentes no sistema feSO, as *threads* de aplicativos de usuário (com prioridade 1), e as *threads* dos processos servidores (prioridade 0). O algoritmo de escalonamento do feSO utiliza 2 filas de *threads* com prioridades diferentes. Um esquema de escalonamento circular 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, 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

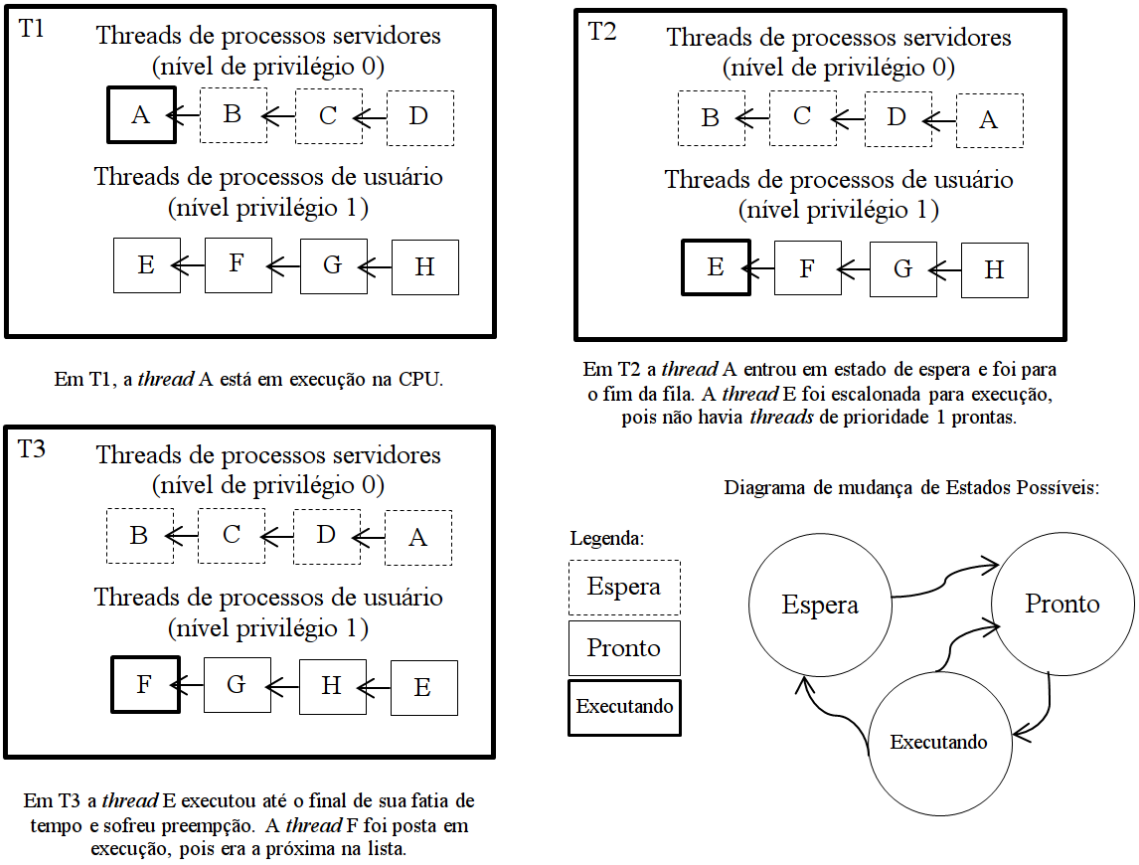

Figura 7. Algoritmo de escalonamento do sistema feSO.

interrupção causada pelos temporizadores do sistema, ou porque alguma *thread* executou uma chamada bloqueante. Em ambos os casos, a *thread* tem sua execução interrompida, e o estado de seus registradores salvo. A diferença é que *threads* cuja fatia de tempo se encerra, ainda estão prontas para execução, e só não estão alocados, pois o sistema decidiu permitir que outra *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 ela está esperando.

O algoritmo de escalonamento coloca a *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, todas as *threads* têm a chance de obter uma fatia de tempo para execução na CPU. Entretanto, como as *threads* de processo servidores têm uma prioridade maior que *threads* de processos de usuário, caso uma *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.

5.5. Comunicação entre Processos

O método de comunicação entre processos primário do sistema feSO é baseado em troca de mensagens. Este tipo de sistema de comunicação utiliza duas primitivas, *send* e *receive*

(enviar e receber) [TANENBAUM 2010]. 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. A Figura 8 mostra um exemplo de comunicação entre um cliente e um servidor, utilizando as chamadas disponibilizadas pelo sistema operacional.

O sistema de troca de mensagens utiliza o conceito de portas. Cada porta está associada a um número, que funciona de forma análoga a uma caixa postal. Uma *thread* pode alocar uma porta, e desta maneira, sempre que uma mensagem for enviada para a porta alocada, será redirecionada para a *thread* que a alocou. Com isso, uma *thread* de um processo servidor pode alocar sempre as mesmas portas, que são previamente conhecidas pelos seus clientes.

As *threads* também utilizam portas para serem avisadas no caso da ocorrência de algum evento no sistema. Por exemplo, as 15 primeiras portas estão relacionadas às IRQs (*interrupt requests*), 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 àquele IRQ está alocada a alguma *thread*. Se uma IRQ for disparada pelo hardware, uma mensagem é enviada à *thread*, informando a ocorrência da interrupção. Está é a maneira pela qual os processos servidores para *drivers* de dispositivos são sinalizados da ocorrência de uma interrupção.

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 uma *thread* especifique qual porta ele pretende alocar. Algumas portas são reservadas apenas para *threads* de processos servidores, para evitar que processos de usuário se apossem delas indevidamente.

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, a *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, a *thread* retoma sua execução quando mensagem chega, independente de sua origem.

5.6. Sistema de Arquivos Virtual

O núcleo do sistema feSO possui suporte 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]. 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.

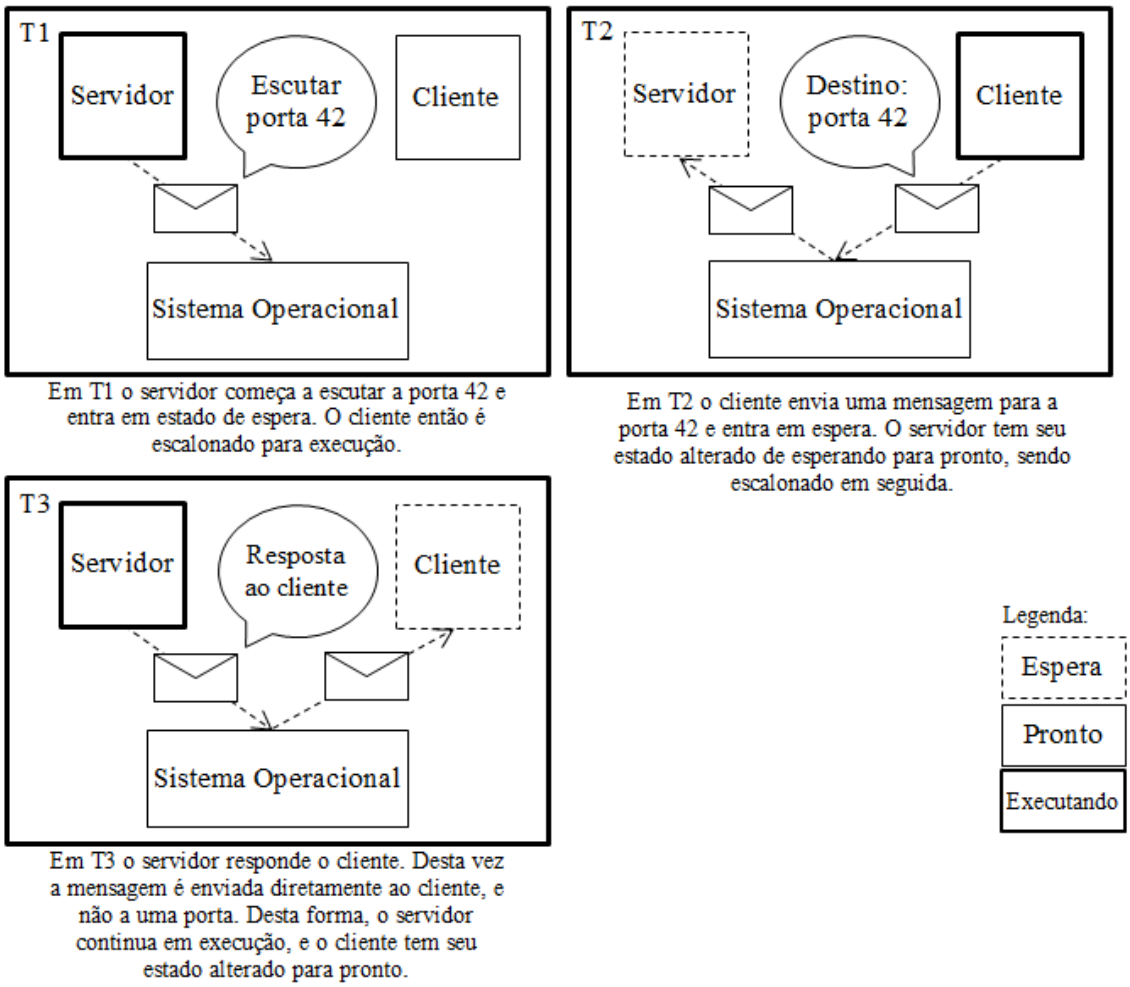

Figura 8. Comunicação entre processos por troca de mensagens.

Cada arquivo do sistema está em uma mesma estrutura hierárquica, formada por arquivos e diretórios. Os diretórios são arquivos especiais, que agrupam um conjunto de arquivos e outros diretórios, e formam a árvore de diretórios. A árvore 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.

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, o local (memória ou dispositivo de armazenamento), e uma lista blocos com os dados armazenados. Essas estruturas ficam na memória, em uma árvore balanceada, ordenadas pelo descritor do nó que representam. O descritor de um nó é um número sequencial utilizado para sua identificação e precisa ser informado durantes as operações realizadas no VFS.

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 ar-

quivos 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.

O VFS do feSO oferece uma estrutura para que arquivos externos sejam manipulados. Esses arquivos externos ficam salvos primeiramente em algum dispositivo de armazenado, como uma mídia óptica 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 9 mostra um exemplo 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 princípio, estes nós contém apenas uma lista de blocos 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.

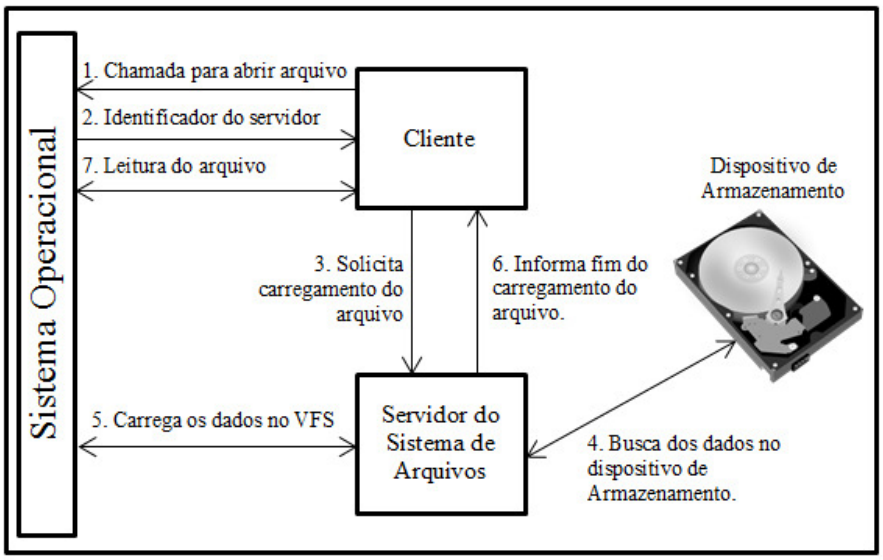

Figura 9. Protocolo para operações sobre aquivos externos do VFS.

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 ativam o 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.

5.7. Sincronização de Processadores

O sistema feSO utiliza-se da técnica da espera ocupada com uma variável de trava, *spinlock*, para controlar o acesso de múltiplos processadores às rotinas do núcleo. Os processadores da família Intel x86 suportam operações atômicas em locais da memória do sistema [INTEL 2007]. 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.

Todas as rotinas do núcleo do sistema ficam bloqueadas por uma única trava. Com isto, apenas um processador pode estar executando rotinas no núcleo ao mesmo tempo. A vantagem dessa técnica é a facilidade de sua implementação e a garantia de que não as estruturas e variáveis globais do núcleo não ficaram em um estado inconsistente por causa do acesso simultâneo de dois ou mais processadores. A desvantagem é a perda de desempenho, pois os processadores eventualmente ficam ociosos esperando por uma chance de acessar as rotinas do *kernel*.

6. Aplicação em Sala de Aula

Os autores deste trabalho vislumbram diversas maneiras através das quais um SO simplificado como o feSO pode ser utilizado durante um curso de sistemas operacionais. São elas:

- **Ensino baseado em exemplos**: Está é a maneira mais imediata de se utilizar o sistema feSO em sala de aula. Cada um dos aspectos teóricos pertinentes ao funcionamento interno de um sistema operacional encontra uma contrapartida prática dentro do código do sistema feSO. Se um professor ou aluno quiser exemplos práticos e simples de: rotinas de tratamento de interrupções; chamadas ao sistema; blocos de controle de processos; algoritmos de escalonamento; algoritmos com políticas de alocação de páginas; primitivas de comunicação entre processos e código de controle de dispositivos de E\S, basta navegar pela relativamente pequena base de código do feSO, e será possível encontrar facilmente algo que ilustre o conceito sendo estudado. Os principais componentes do feSO estão encapsulados em classes C++ com nomes significativos, o que facilita ainda mais esta tarefa.

- **Ensino através de projetos de implementação**: O aprendizado ativo, no qual o aluno se torna protagonista e não um mero espectador durantes as aulas é uma característica desejável também em cursos de SO. O sistema feSO pode

ajudar neste aspecto, servido de base para diversas tarefas de implementação. O feSO foi construído com base em tipos abstratos de dados comuns (listas e árvores balanceadas) que podem ser manipulados pelos alunos de maneira simples. Por isso, é possível organizar tarefas que envolvam substituir ou recriar trechos de códigos do sistema. Por exemplo, uma tarefa interessante seria de substituir o algoritmo de escalonamento e medir o impacto desta substituição no *throughput* da execução de processos. Outro exemplo pode envolver a alteração das políticas de alocação de páginas e medir o grau de fragmentação da memória. Também é possível alterar as primitivas de comunicação e medir impacto no desempenho, bem como aferir a diferença de tempo entre as trocas de contexto e comunicação entre processos. Os alunos também podem criar novas chamadas ao sistema junto com aplicações clientes dessas mesmas chamadas, o que aprofundará o entendimento de como os códigos de usuário e o núcleo de um SO interagem. Essas são apenas algumas alternativas dentre uma miríade de possibilidades que a utilização de um sistema real, porém simplificado, pode oferecer.

- **Ensino de fundamentos de sincronização e programação paralela**: O estudo da programação paralela e das técnicas de sincronização que são tão necessárias aos sistemas operacionais tem aplicações incontáveis, mesmo em outros contextos. Com o suporte a processadores de múltiplos núcleos e a criação de *threads* em espaço de usuário, os alunos podem ter os primeiros contatos com aplicações paralelas baseadas na interface simplificada de programação oferecida pelo sistema feSO. Os estudantes podem utilizar as primitivas de comunicação fornecida pelo sistema para sincronizar aplicações diversas, ou mesmo criar seus próprios protocolos com base nos já implementados para a comunicação entre servidores e aplicativos de usuário.

- **Ensino de hardware e programação de baixo nível**: O conhecimento sobre sistemas operacionais e *drivers* de dispositivos pode ser a porta de entrada para a criação de aplicações de baixo nível, que necessitem de conhecimento em *assembly* ou mesmo controle direto do hardware do computador. Programar diretamente no núcleo de um sistema operacional permite que estudantes experienciem total controle do hardware e da memória de um microcomputador.

Além de todas essas aplicações, o sistema feSO foi criado com o objetivo de fomentar o estudo de sistemas operacionais dentro de um curso de graduação e também de criar uma comunidade de desenvolvimento. Espera-se que novos projetos e trabalhos de conclusão de curso possam surgir com base no sistema futuramente.

7. Conclusão

Neste trabalho, apresentou-se os principais aspectos de funcionamento de um sistema operacional simplificado para fins educativos chamado feSO. Foram discutidos detalhes sobre os módulos mais importantes do núcleo, como a gerência de memória, escalonamento, comunicação entre processos, sistema de arquivos virtual.

Diversas melhorias e expansões no sistema podem ser propostas, como o suporte a paginação, bibliotecas compartilhadas, novos *drivers*, interfaces gráficas e o porte de novas ferramentas já existentes.

Atualmente, o sistema feSO está disponível no GitHub (https://github.com/ hllustosa/feso.git). No momento, o sistema ainda possui poucas aplicações nativas de exemplo, como um editor de textos, uma ferramenta de linha de comando, aplicações lúdicas simples, entre outras. Um dos autores vem trabalhando em sua adaptação para a utilização em um curso prático sobre o funcionamento de sistemas operacionais.

Referências

CHEN, J. Bradley e BERSHAD, B. N. (1993). The impact of operating system structure on memory system performance. *SIGOPS Oper. Syst. Rev.*, 27(5):120–133.

COX, R. (2018). Xv6, a simple unix-like teaching os. Accessed: 2018-02-22.

FOUNDATION, F. S. (2011). Grub - grand unified bootloader. https://www.gnu.org/software/grub/. Accessed: 2018-01-23.

FOUNDATION, F. S. (2018). Cross-compilation. https://www.gnu.org/software/automake/manual/html_node/Cross_002dCompilation.html. Accessed: 2018-01-23.

HAYS, J. H. (1989). An operating systems course using minix. *SIGCSE.*, 21(4):11–12.

HYDE, R. (2003). *The Art of Assembly Language*. Number v. 1. No Starch Press.

IEEE (2018). Posix.1-2008. Accessed: 2018-02-22.

INTEL (2007). *Intel 64 and IA-32 Software Developer's Manual*. Intel Corporation.

KERNIGHAN, B.W. e RITCHIE, D. (1988). *The C Programming Language*. Prentice-Hall software series. Prentice Hall.

LIEDTKE, J. (1995). On micro-kernel construction. *SIGOPS*, 29(5):237–250.

LOHNER, C. (2018). Funny statistics for the linux kernel. Accessed: 2018-02-22.

MACHADO, F.B. e MAIA, L. (2013). *Arquitetura De Sistemas Operacionais*. LTC.

MAIA, L. (2018). Sosim: Simulador para o ensino de sistemas operacionais. Accessed: 2018-02-22.

OSDEV (2018). Osdev wiki. Accessed: 2018-02-22.

POLZE, P. (2006). Teaching os: The windows case. *SIGCSE*, 38(1):298–302.

ROMAN OTERO, R. E. A. A. (2015). Minios: An instructional platform for teaching operating systems projects. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 430–435, New York, NY, USA. ACM.

SILBERSCHATZ, A. (2000). *SOs: conceitos*. Prentice Hall.

STALLINGS, W. (2010). *Arquitetura e organização de computadores*. PRENTICE HALL BRASIL.

TANENBAUM, A. (2010). *Sistemas operacionais modernos*. Prentice-Hall do Brasil.

TIS (1995). Executable and linking format (elf) specification. http://refspecs.linuxbase.org/elf/elf.pdf. Accessed: 2018-01-23.

WILLIAMS, A. (2018). Intermezzos. a little os. Accessed: 2018-02-22.