Jorge Pires Correia

Implementação de um Sistema Operacional Experimental Incremental e Modular

Vitória da Conquista - BA

2020

Jorge Pires Correia

Implementação de um Sistema Operacional Experimental Incremental e Modular

Trabalho de Conclusão de Curso apresentado como requisito para a conclusão do curso de Ciência da Computação da Universidade Estadual do Sudoeste da Bahia.

Universidade Estadual do Sudoeste da Bahia

Orientador: Marlos André Marques Simões de Oliveira

Vitória da Conquista - BA

2020

Resumo

A prática dentro do contexto do estudo de sistemas operacionais é constantemente negligenciada dada a complexidade inerente a este software básico. No intuito de permitir uma melhor compreensão do seu processo de construção, este trabalho apresenta a implementação de um sistema operacional de forma incremental e modular, de modo a permitir aos estudantes um melhor acompanhamento e compreensão no estudo deste tópico. A característica da incrementalidade possibilita um processo de criação passo-a-passo, mantendo a funcionalidade do sistema em cada etapa e propiciando um aprendizado gradual. A modularidade por sua vez, permite experimentações por parte dos alunos, uma vez que os módulos podem ser substituídos por outras implementações, desde que estas sigam as interfaces definidas no projeto. Ainda que experimental, o sistema operacional aqui proposto e implementado executa em uma plataforma computacional real. E dada a sua característica puramente acadêmica, a escolha das funcionalidades implementadas foi realizada de modo a cobrir a maior parte dos requisitos de um sistema operacional, sem aprofundar em uma área específica. Dessa forma, o sistema também pode ser utilizado na experimentação em conjunto com diversos componentes curriculares dos cursos de computação, como por exemplo, redes de computadores, compiladores, entre outros.

Palavras-chaves: Experimentação; Implementação; Incremental; Modular; Prática; Sistemas Operacionais.

Abstract

The practice within the Operating Systems study context is constantly neglected given the inherent complexity of this basic software. In order to allow a better comprehension of its construction process, this work presents the implementation of an operating system in an incremental and modular way, in order to allow the students a better understand in this subject. The incremental characteristics allows a step-by-step creation process, maintaining the system working at each step and providing gradual learning. The modularity, in turn, allows students to experiment, since the modules can be replaced by other implementation, whereas they follow the project interfaces. Although experimental, the proposed and implemented Operating System runs on a real platform. Given its purely academic characteristic, the choice of implemented features was made aiming the most coverage of the operating system requirements, without delve into a specific area. Thus, the Operating System can also be used in a experimentation with other curricular components of computer science courses, such as computer networks, compilers, among others.

Key-words: Experimentation; Implementation; Incremental; Modular; Practice; Operating Systems.

Lista de ilustrações

Lista de quadros

Lista de códigos

Lista de abreviaturas e siglas

ABI	Application Binary Interface
BCD	Binary-coded Decimal
BIOS	Basic Input/Output System
CISC	Complex Instruction Set
CPU	Central Processing Unit
CSM	Compatibility Suppot Module
DMA	Direct Memory Access
ELF	Executable Linkable Format
FDC	Floppy Disk Controller
GDT	Global Descriptor Table
GPT	GUID Partition Table
HD	Hard Drive
IDT	Interrupt Descriptor Table
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
ISR	Interrupt Service Routines
KVM	Kernel-based Virtual Machine
MBR	Master Boot Record
MMU	Memory Management Unit
PCI	Peripheral Component Interconnect
PE	Portable Executable
PIC	Programmable Interrupt Controller

PIT	Programmable Interval Timer
POST	Powe-On Self-Test
RAM	Random-Access Memory
RISC	Reduced Instruction Set
TCB	Task Control Block
UEFI	Unified Extensible Firmware Interface
VFS	Virtual File System
VGA	Video Graphics Array

Sumário

1 Introdução

1.1 Contextualização do problema

A constante evolução de todos os componentes de hardware presentes em computadores, naturalmente, gera um aumento na complexidade dos sistemas computacionais em geral. Diante disso, os sistemas operacionais fornecem uma interface para os programas aplicativos, disponibilizando todas as funcionalidades existentes em baixo nível de maneira simplificada. Além de atuar como uma interface entre o hardware e os softwares de alto nível, os sistemas operacionais também atuam como gerentes de recursos: políticas de acesso a dispositivos, gerenciamento de usuários e segurança de uma maneira geral são exemplos de atividades atribuídas aos sistemas operacionais.

Tendo em vista a abrangente utilização de sistemas operacionais em computadores, seu estudo dentro dos cursos da área de computação é de extrema importância. Algoritmos utilizados em diferentes sistemas operacionais estão presentes nas ementas dos cursos de graduação em computação, contudo, a visualização desses algoritmos na prática é dificultada pela complexidade dos sistemas operacionais existentes atualmente. Um sistema operacional simples pode conter centenas de arquivos e milhares de linhas de códigos, tornando laborioso e fatigante o seu estudo.

1.2 Justificativa e motivação

Um sistema operacional composto de diversas versões incrementais e funcionais proporciona ao estudante uma visão modular do sistema, permitindo o estudo isolado dos algoritmos que o compõem. Tal cenário oferece, também, um objeto de experimentação para os estudantes: visto que o sistema é modularizado, a inserção de novas funcionalidades ou a alteração de mecanismos já existentes é realizada de maneira a não afetar diretamente funcionalidades não relacionadas.

1.3 Objetivos

1.3.1 Geral

Partindo do princípio que o tópico Sistemas Operacionais está compreendido dentro das tecnologias de computação, a experimentação é uma atividade essencial na sua aprendizagem. Desta maneira, este trabalho busca desenvolver, com base em uma

abordagem *bottom-up*, um objeto de estudo de Sistemas Operacionais, de modo a tornar o aprendizado mais prático e ágil.

1.3.2 Específicos

Para compor o sistema operacional proposto, os seguintes módulos, implementados incrementalmente, estão expostos neste trabalho: implementação de um sistema de *boot*; implementação de funções que formam uma biblioteca C do sistema; implementação de um mecanismo de comunicação com dispositivos de entrada e saída; implementação de um mecanismo de tratamento de interrupções; implementação de *drivers* de vídeo e teclado; implementação do gerenciamento de memória física através de uma área de *heap*; implementação do gerenciamento de memória virtual; implementação de um mecanismo de tratamento de dispositivos conectados ao barramento PCI;implementação de multitarefas.

1.4 Organização do trabalho

Além deste capítulo, este trabalho apresentará o referencial teórico no Capítulo 2. O Capítulo 3 mostrará o estado da arte em termos de sistemas operacionais experimentais e modulares. O Capítulo 4 descreverá o processo de desenvolvimento do sistema operacional, assim como as devidas explicações do seu código. As considerações finais, seguidas das referências bibliográficas utilizadas neste trabalho encerram o mesmo.

2 Referencial Teórico

Um sistema operacional é uma camada de software que roda acima do nível ISA (TANENBAUM, 2012), cuja função é fornecer aos programas do usuário um modelo do computador melhor, mais simples e mais limpo, assim como lidar com o gerenciamento de todos os recursos disponibilizados pelos hardware (TANENBAUM, 2016).

Tendo em vista que utilizar somente os mecanismos providos pelo processador para desenvolver alguma atividade é uma tarefa pouco eficiente, o sistema operacional proporciona abstração e gerenciamento dos recursos disponíveis para o usuário. Este conjunto de recursos normalmente se baseia na arquitetura de von Neumann, inicialmente idealizada em von Neumann (1945). Obviamente, com a evolução dos dispositivos físicos, novas subdivisões deste modelo foram adicionadas ao conjunto de elementos sobre o controle do sistema operacional, tornando-o um software essencial para os computadores pessoais modernos.

2.1 Inicialização de um Sistema Operacional

O processo de inicialização de um sistema operacional, também conhecido como *boot*, pode ser analisado como uma sequência de passos: no momento em que o computador é ligado, uma verificação de hardware, denominada POST é realizada, tendo como objetivo realizar a verificação de registradores da CPU, verificação de integridade do firmware de inicialização, verificação de componentes básicos como DMA, temporizador e controlador de interrupções, localizar e verificar tamanho e integridade da memória principal; o firmware de *boot* é carregado na memória RAM e executado, adequando o ambiente de execução, localizando o sistema operacional e o executando (WIKIPEDIA, 2020l). Atualmente, dois firmwares são comumente utilizados: BIOS tradicional e UEFI.

2.1.1 BIOS

BIOS é um firmware que foi criado com o objetivo de fornecer serviços gerais de baixo nível para os primeiros programadores (OSDEV, 2019a). Atualmente, o BIOS ainda é bastante utilizado no processo de *boot*, pois ele provê diversos serviços que são úteis e fáceis de serem utilizados.

As funcionalidades do BIOS são geralmente acessadas a partir de chamada de interrupções, com parâmetros definidos nos registradores AX ou EAX. Cada interrupção do BIOS representa um grupo de funcionalidades diferentes, como mostra o Quadro 2.1, e

possui um conjunto de registradores que armazena os valores resultantes. Todas as funções providas pelo BIOS foram descritas em Brown (2000).

Quadro 2.1 – Interrupções do BIOS (OSDEV, 2019a).

Número da interrupção	**Função**
0x10	Manipulação de vídeo
0x13	Acesso a dispositivos de armazenamento
0x15	Acesso a memória
0x16	Acesso ao teclado

Quando utilizado em processos de *boot*, o BIOS é, inicialmente, executado em *Real Mode*, por uma questão de compatibilidade com processadores antigos. Nesse modo de operação da CPU, o BIOS é capaz de instalar drivers de dispositivos e lidar com interrupções, além de oferecer uma coleção de funções de baixo nível e possuir um rápido acesso a memória, visto que o acesso é direto utilizando endereços físicos e registradores pequenos. A mudança do *Real mode* para o *Protected Mode* impede o uso de grande parte das funcionalidades do BIOS, inclusive das interrupções. Quando iniciado, o BIOS faz uma varredura em todos os dispositivos de armazenamento em busca da assinatura de *boot* no primeiro setor de cada dispositivo. Essa assinatura está presente nos bits 510 e 511, contendo respectivamente os valores 0x55 e 0xAA.

Caso o dispositivo de armazenamento que possua a assinatura seja do tipo *floppy*, todo o setor será composto por um código executável. Desta maneira, o BIOS carregará o código presente do byte 0x0 até o byte 0x1FD (0..509) para a posição de memória endereçada por 0x7C00, e esse código é executado. Caso o dispositivo seja do tipo *hard disk*, o código executável estará presente nos bytes 0x0 ao byte 0x01BD (0..445). Os bytes 0x01BE ao byte 0x01FD (446..509) conterão uma tabela que esquematiza a divisão de partições do HD, chamada de tabela MBR. Essa tabela é composta por quatro entradas chamadas partições primárias, cada uma composta por 16 bytes. Cada entrada da tabela MBR possui um layout padrão, como mostra o Quadro 2.2. Caso seja necessária a utilização de mais que quatro partições, a tabela MBR possui um tipo de partição denominada partição estendida. Esse tipo de partição pode formar uma lista encadeada de novas partições.

Após o carregamento do código executável na memória, o identificador do dispositivo de armazenamento que contém o código de *boot* é armazenado no registrador DX.

Quadro 2.2 – Entradas primárias da tabela MBR (OSDEV, 2019f).

Elemento (offset)	**Tamanho**	**Descrição**
0	byte	Flag de indicação de Boot: 0 = não, 0x80 = bootable (ou "ativo")
1	byte	Cabeçalho inicial
2	6 bits	Setor inicial (Bits 6-7 são os dois bits altos para o campo Cilindro Inicial)
3	10 bits	Cilindro Inicial
4	byte	System ID
5	byte	Cabeçalho final
6	6 bits	Setor final (Bits 6-7 são os dois bits altos para o campo cilindro final)
7	10 bits	Cilindro final
8	4 bytes	Setor relativo (para o início da partição – também igual ao valor LBA inicial da partição)
12	4 bytes	Total de setores na partição

2.1.2 UEFI

O UEFI é um firmware utilizado no processo de *boot*, que utiliza funcionalidades do BIOS. Contudo, o UEFI busca o código executável em locais diferentes e organiza a memória em um layout diferenciado. Diferentemente do BIOS, o UEFI proporciona diretamente um ambiente *protected mode*, com uma arquitetura *flat* de segmentos. Sendo um padrão, o UEFI não possui comportamentos diferenciados entre diferentes tipos de processadores.

Diferentemente do BIOS, o UEFI busca em dispositivos de armazenamentos particionados segundo o modelo GPT (OSDEV, 2020b) uma partição no formato FAT, que contém um aplicativo UEFI em formato executável PE. Esse aplicativo UEFI é carregado na memória, em um endereço definido em tempo de execução. O controle é passado para o ponto de entrada do aplicativo UEFI (OSDEV, 2020f). Por uma questão de compatibilidade, o firmware UEFI possui um mecanismo denominado CSM, que emula o BIOS tradicional.

O UEFI disponibiliza um conjunto de funções mapeadas na memória, que são referenciadas pela System Table (OSDEV, 2020f). Essa forma fornecimento de funções se difere do BIOS pois o UEFI não utiliza interrupções. Isso acaba com a limitação da quantidade de funções e permite a utilização de convenções de chamadas de funções. O formato e o comportamento das funções providas pelo UEFI são definidas pelas suas especificação, permitindo uma maior compatibilidade entre as plataformas. Por esse motivo, o UEFI é o firmware mais utilizado em sistemas atuais.

2.1.3 El-Torito

O processo de boot pode ser realizado, também, por dispositivos opticos, como CD-ROMs, DVD ou BD a partir do padrão *El-Torito*. Este padrão pode emular *floppys* e *hard disk* além de poder ser utilizado sem emulação. O sistema de arquivos utilizado pelos dispositivos que seguem o padrão *El-Torito* é o ISO 9660 (OSDEV, 2019e).

O sistema de arquivos ISO 9660 armazena no endereço de bloco 0x11 uma estrutura denominada Boot Record. Essa estrutura lista as imagens de *boot* disponíveis no sistema de arquivo (OSDEV, 2018c), que dependendo da forma que foi criado, pode ser interpretado pelo BIOS ou pelo UEFI. Diversas ferramentas são disponibilizadas para montar sistemas de arquivos ISO 9660, como a mkisofs.

2.2 Memória Virtual

Memória virtual tem como objetivo ocultar a organização complexa da memória física, simplificar os procedimentos de alocação da memória aos processos, além de garantir proteção para a memória (MAZIERO, 2019). Em processadores x86, a virtualização de memória pode ser realizada graças a um dispositivo denominado MMU. Como mostra a Figura 2.1, a MMU traduz os endereços virtuais gerados pelo processador para endereços físicos da memória. Existem três tipos principais de virtualização de memória: por partições, por segmentos e por páginas (MAZIERO, 2019).

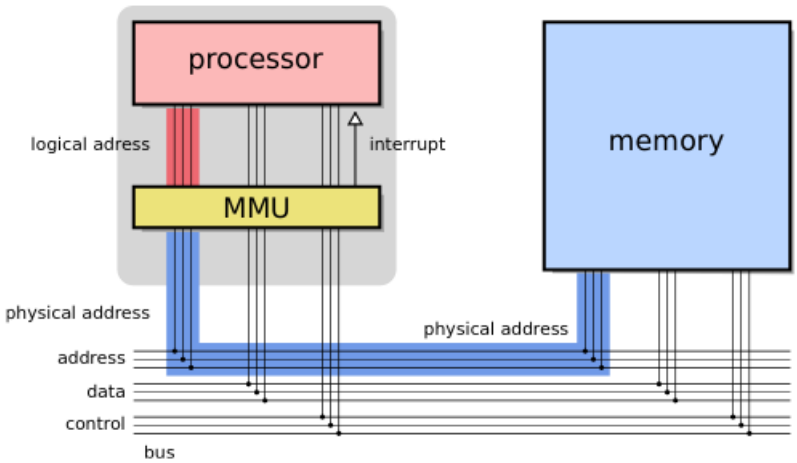


Figura 2.1 – Funcionamento básico da MMU (MAZIERO, 2019, p. 172).

A virtualização por partições proporciona ao processo um intervalo de endereçamento com o espaço necessário. Caso o processo ocupe T bytes, então ele poderá acessar endereços no intervalo [0..T-1]. O núcleo possui uma tabela que armazena a base e o limite para cada processo. Esses dois valores preenchem dois registradores na MMU, que serão usados para verificar se o processo pode acessar tal endereço, e gerar o endereço físico

correspondente ao endereço virtual. A Figura 2.2 apresenta o funcionamento da memória virtual por partições.

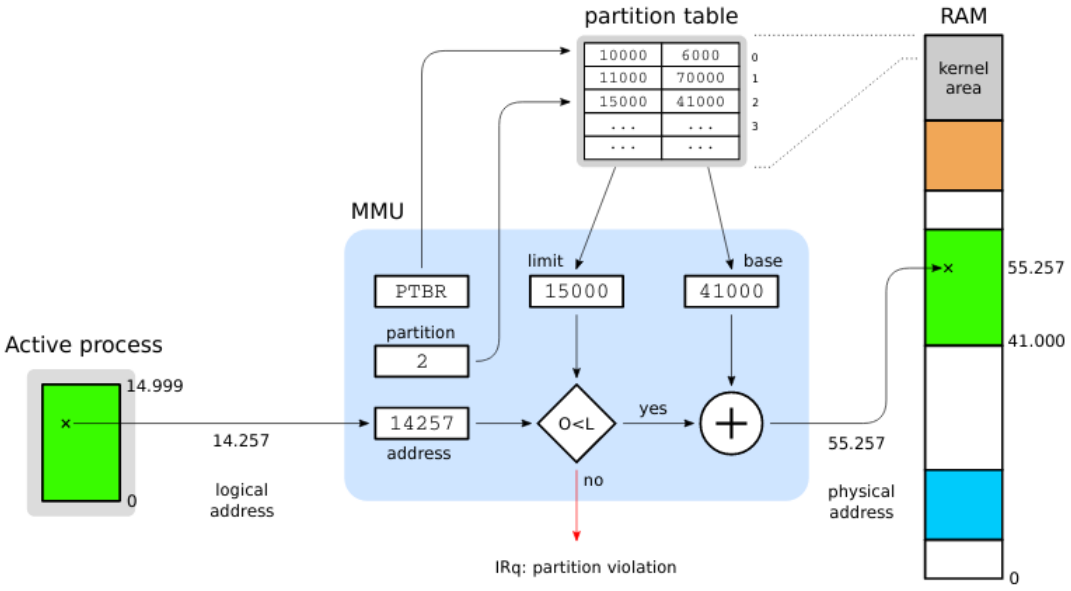


Figura 2.2 – MMU com partições (MAZIERO, 2019, p. 174).

A virtualização por segmento se baseia na virtualização por partições, contudo, possibilita a alocação fragmentada do processo. O processo é dividido em segmentos, que são definidos de acordo as sessões de memória do processo, como área de texto e área de dados. Cada segmento se comporta como uma partição, tendo endereços virtuais independentes. Os endereços são gerados a partir do par [segmento : deslocamento], onde o segmento indica a partição e o deslocamento indica o endereço virtual dentro daquela partição. Para cada processo deve existir uma tabela que armazena a base e o limite para todos os segmentos que o compõe. Esses valores são utilizados para preencher registradores específicos da MMU, responsáveis por verificar se o processo pode acessar tal endereço e gerar o endereço físico correspondente ao endereço virtual. A Figura 2.3 apresenta o funcionamento da memória virtual por segmento.

A virtualização por paginação é o modo de virtualização mais utilizado pelos sistemas operacionais atuais. A memória virtual do processo é um conjunto de endereços virtuais consecutivos, dividido em pequenos blocos de tamanho fixo denominados páginas, normalmente de tamanho 4096 bytes. A memória física também é dividida em blocos de mesmo tamanho, que são denominados quadros. O endereço virtual criado é único, mas pode ser logicamente dividido em número de página e deslocamento. Quando esse endereço chega na MMU, o número de página é substituído pelo número de quadro de acordo a tabela de páginas. Caso não exista um quadro associado a página, uma interrupção é gerada (*Page Fault*). O deslocamento é preservado durante a tradução de endereços, e referencia o deslocamento dentro da página ou quadro. A Figura 2.4 apresenta o funcionamento da

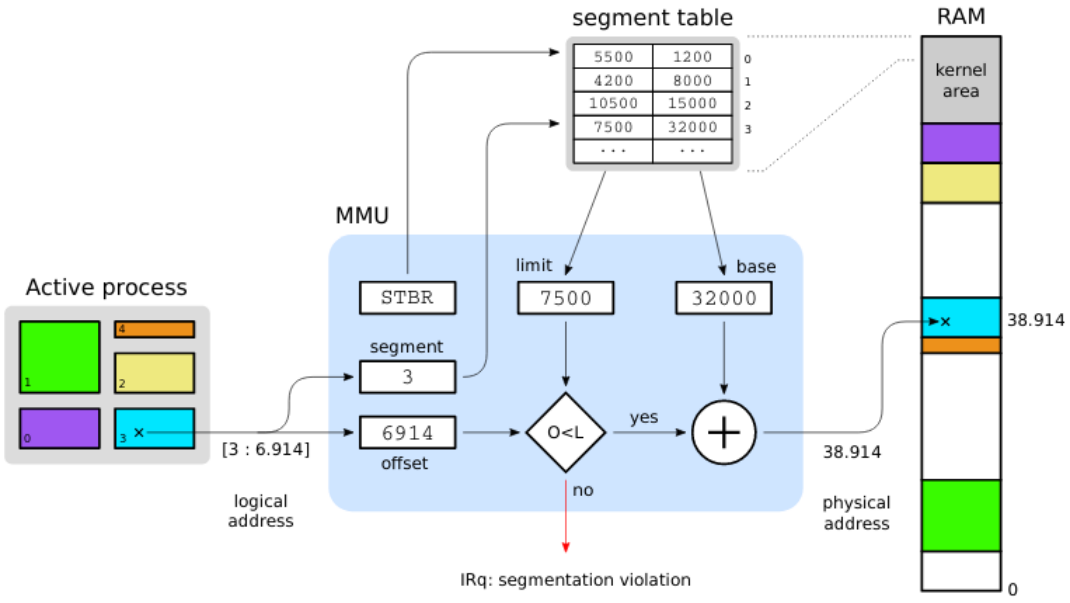


Figura 2.3 – MMU com segmentação (MAZIERO, 2019, p. 176).

memória virtual por partições.

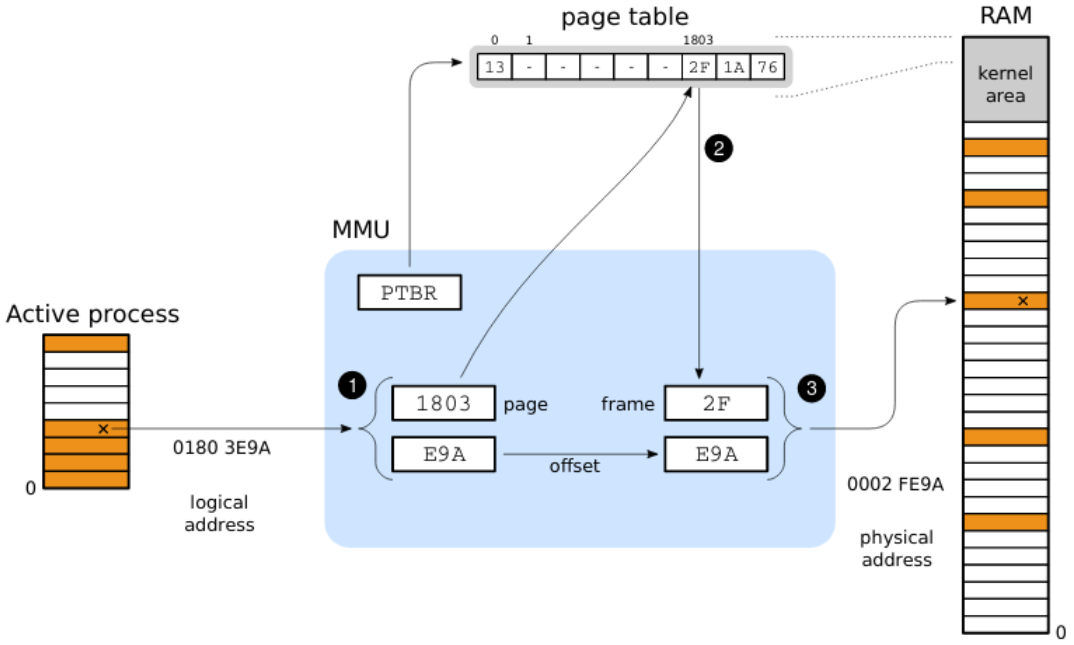


Figura 2.4 – MMU com paginação (MAZIERO, 2019, p. 179).

2.3 Tratamento de Interrupções

Uma interrupção é um sinal de entrada para o processador que indica a ocorrência de um evento que precisa de atenção imediata (WIKIPEDIA, 2020i). Como descrito

em OSDEV (2018d), as interrupções podem ser classificadas em três grupos distintos: exceções, requisições de interrupções (IRQs) e interrupções de software, como demonstrado na Figura 2.5.

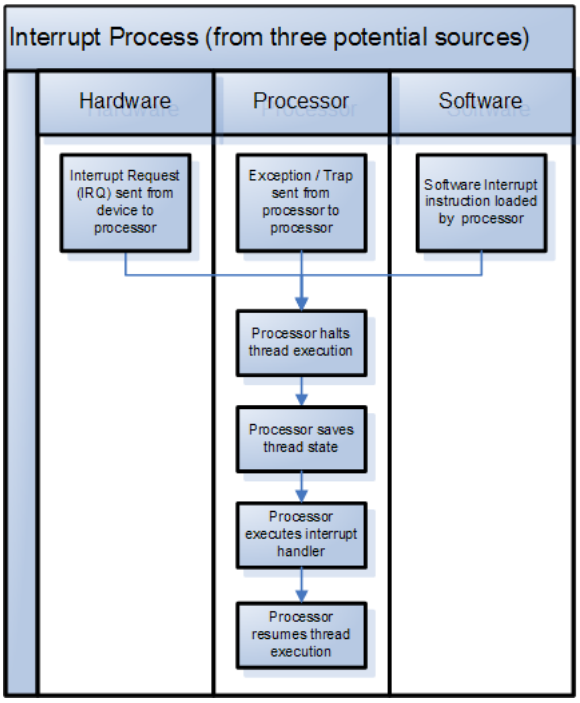


Figura 2.5 – Processo de tratamento de uma interrupção (WIKIPEDIA, 2020i).

Exceções são interrupções geradas internamente pelo processador, comumente utilizadas para notificar o núcleo sobre alguma situação adversa. OSDEV (2019c) classifica as exceções em três grupos: *faults*, *traps* e *aborts*. *Faults* são interrupções que indicam situações que podem ser corrigidas, de maneira a permitir que o processo que a gerou prossiga sua execução como se a interrupção não tivesse sido gerada. *Traps* são interrupções lançadas no momento que uma instrução de *Trap* é executada. Essas instruções normalmente geram as interrupções do tipo *debug*, *breakpoint* e *overflow*. *Abort* são interrupções que indicam um erro irrecuperável, como a *Double fault*.

Requisições de interrupção são interrupções geradas por dispositivos fora do processador. PIC é o hardware responsável por gerenciar as requisições de interrupções no escopo deste trabalho (OSDEV, 2018a). Os dispositivos que pretendem gerar interrupções são conectados ao PIC, que é conectado ao *interrupt pin* do processador. O PIC repassa uma interrupção por vez para a CPU, de forma que quando o processador termina de tratar a interrupção, o PIC é informado, permitindo o repasse de uma nova IRQ. A arquitetura do PIC é formada por um circuito que possui 8 entradas, uma para cada dispositivo que deseja utilizar interrupções, e uma saída, que é conectada ao *interrupt pin* do processador. Com o objetivo de prover mais entradas para interrupções, dois circuitos idênticos são utilizados, de forma que um é conectado à CPU, e outro é conectado ao primeiro. A saída

do segundo circuito é conectada à entrada 2 do primeiro, de maneira que 15 entradas de interrupções são disponibilizadas. O primeiro circuito é denominado *Master PIC* e o segundo é denominado *Slave PIC*.

Interrupções de software são geradas por processos para solicitar alguma funcionalidade do núcleo. Essas interrupções são, em sua maioria, geradas a partir de chamadas de sistemas disponibilizadas pelo núcleo. As chamadas de sistemas formam a interface de comunicação entre softwares de alto nível e o núcleo, de forma a disponibilizar funcionalidades nas áreas de gestão de processos, gestão de memória, gestão de arquivos, comunicação, gestão de dispositivos e gestão do sistema (MAZIERO, 2019). Para que as interrupções possam ser utilizadas no *Protected Mode*, o processador utiliza uma tabela denominada IDT, apontada pelo registrador IDTR, que contém informações sobre o tratamento de funções. Esta tabela armazena, para cada interrupção, o endereço do código que realiza as operações necessárias para que a função de tratamento da interrupção possa executar, e que realiza também, o retorno para a execução normal. Empilhar byte de erro, caso necessário; empilhar número da interrupção; salvar contexto da CPU; preparar registradores de segmentos são operações realizadas antes chamada da função de tratamento. Desempilhar valores utilizados para o tratamento da interrupção; restaurar o contexto do processador; retornar para a execução normal são operações realizadas pelo mesmo código, após o tratamento da interrupção ser realizada. Neste trabalho, o núcleo disponibiliza uma interface na qual os *drivers* ou gerenciadores de recursos podem registrar a função que será utilizada para tratar a interrupção desejada.

2.4 Programmable Interval Timer

O *Programmable Interval Timer* (PIT) é um dispositivo que gera sinais de saída em intervalos de tempos determinados via software (WIKIPEDIA, 2018). Os sinais de saída do PIT são gerados a partir de um oscilador que funciona a uma frequência de aproximadamente 1.193182 MHz. Com o auxílio de divisores de frequência, que utilizam contadores, pode-se conseguir frequências mais baixas, permitindo a melhor manipulação deste dispositivo.

O PIT possui 3 canais de saída de sinais (OSDEV, 2020e): o canal 0 é conectado diretamente ao IRQ0 do PIC; o canal 1 costumava ser conectado à memória RAM para "atualização" dos capacitores, mas computadores atuais utilizam um dispositivo específico para realizar tal atividade, portanto este canal se tornou obsoleto; o canal 2 é conectado ao *PC speaker*. O PIT possui uma porta de entrada/saída para controle (0x43), e uma porta de dados para cada canal (0x40, 0x41 e 0x42 respectivamente).

Os divisores de frequência recebem um valor de 16 bits que é decrementado a cada ciclo de oscilação do PIT. Quando este contador atinge 0, um sinal é disparado. O valor

inicial do contador é recebido a partir das portas de dados do PIT, que recebem os bits menos significativos e os bits mais significativos de forma independente em cada operação de escrita.

O registrador de controle possui 8 bits divididos da seguinte maneira:

- **Bits 6 e 7:** Indicam o canal que as configurações desse registrador se referem;
- **Bits 4 e 5:** Indicam o modo de acesso aos registradores de dados. Os bits com valores "0, 0" indicam que a escrita nas portas de dados é bloqueada; os bits com valores "0, 1" indicam que todas as escritas nas portas de dados referenciam o byte de baixa ordem; os bits com valores "1, 0" indicam que todas as escritas nas portas de dados referenciam o byte de alta ordem; os bits com valores "1, 1" indicam que a primeira escrita referencia o byte de baixa ordem, e a segunda escrita indica o byte de alta ordem;
- **Bits 1 a 3:** Indicam o modo de operação do PIT;
- **Bit 0:** Indica a codificação da porta de dados, onde o valor 0 indica que utilizará o formato binário, e o valor 1 indica que o formato BCD (cada quatro bits representa um dígito decimal) será utilizado.

Em cada modo de operação o PIT se comporta de maneira diferente, mas todos compartilham algumas características (OSDEV, 2020e), tais como: todas as vezes que o registrador de controle é escrito, toda a lógica interna do canal especificado é reinicia; um valor inicial do contador pode ser escrito a qualquer momento; o valor atual do contador é decrementado ou reiniciado para o valor inicial na parte baixa do sinal de entrada; em modos que o valor atual do contador é decrementado quando este é carregado, este começa a ser decrementado no próximo pulso de *clock*. A lista completa dos modos de operações pode ser encontrada em OSDEV (2020e).

2.5 Multitarefas

Em um sistema computacional, é comum a necessidade da execução de mais de uma tarefa ao mesmo tempo. Isso gera a necessidade do isolamento destas, além do chaveamento do processador e de outros recursos entre as *tasks* existentes no sistema.

Uma *task* é definida como sendo a execução de um fluxo sequencial de instruções, construído para atender uma finalidade específica (MAZIERO, 2019). Do ponto de vista conceitual, o termo *task* é ambíguo, pois pode se referenciar a qualquer unidade de execução como processos, *threads*, processos leves, entre outros (WIKIPEDIA, 2020o). Nos sistemas operacionais atuais, um processo é definido como uma instancia de um programa em

execução (TANENBAUM, 2016). Um processo pode, internamente, realizar mais de uma tarefa, ou seja, pode ter mais de uma linha de execução de código. Essas linhas de execução são denominadas *threads*.

Conceitualmente, somente um processo pode ter posse da CPU por vez. Para que esta restrição não se tornasse uma limitação dos computadores, uma técnica denominada multiprogramação foi desenvolvida (TANENBAUM, 2016), a fim de permitir que vários processos executem concorrentemente em uma plataforma com somente uma CPU.

A multiprogramação se baseia no mecanismo de *time-sharing* (WIKIPEDIA, 2020p), em que cada processo tem um espaço de tempo para ter posse da CPU, e ao final deste tempo, ceder-la para outro processo. Com esse tempo pequeno, o usuário tem a percepção de que vários processos estão sendo executados ao mesmo tempo. Esse tipo de execução, denominada execução concorrente, é amplamente utilizada nos computadores pessoais atuais. A multiprogramação requer que o sistema operacional armazene o estado que o processo se encontra, pois caso este tenha que ser interrompido para ceder a CPU, ele possa retomar a sua execução normalmente. Este estado é denominado contexto, e sua composição é dependente da implementação. Normalmente, o contexto possui, entre outras informações, valores de registradores, valores de pilha e a próxima instrução a ser executada.

Quando um processo tem que ceder a CPU para outro processo, a troca de contexto é realizada. Neste procedimento, o sistema operacional salva o contexto do processo que irá liberar a CPU, restaura o contexto do processo que irá ganhar o controle da CPU e o executa. Desta forma, a troca de contexto é transparente para os processos, dando a ideia de que cada processo tem sua própria CPU e que eles executam sem interrupções. A Figura 2.6 demonstra os cenários descritos acima.

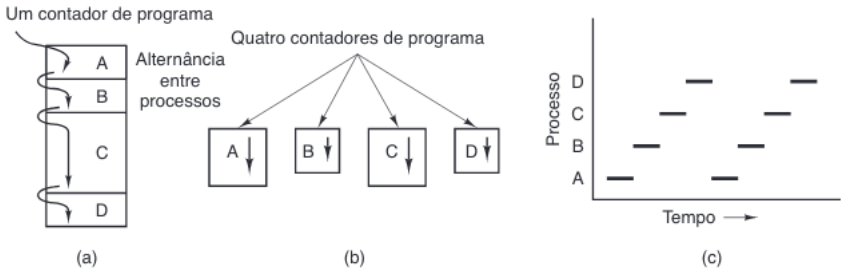


Figura 2.6 – (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Apenas um programa está ativo de cada vez (TANENBAUM, 2016).

2.6 Peripheral Component Interconnect

O PCI é um barramento de comunicação que conecta diferentes tipos de dispositivos ao processador de maneira padronizada, permitindo o controle de inicialização e configuração dos dispositivos via software (OSDEV, 2020d). De acordo com a especificação PCI (PCI Special Interest Group, 1998), cada dispositivo a ele conectado deve disponibilizar um conjunto de registradores que formam o *Configuration Space*. Existem diferentes formas de acesso a esses registradores, de maneira que este projeto utiliza as portas de entrada e saída endereçadas por 0xCF8 e 0xCFC.

O *Configuration Space* é um conjunto de registradores que formam um espaço de 256 bytes, responsável por armazenar informações que indicam o modo de operação do dispositivo, ponteiros para DMA a partir dos quais podemos acessar registradores específicos do dispositivo, entre outras informações. Os quatro primeiros registradores formam o cabeçalho do *Configuration Space*, sendo seguidos de outros registradores que dependem do tipo do cabeçalho. O Quadro 2.3 apresenta a disposição do *Configuration Space* de tipo 0x0, que será utilizado neste trabalho.

Quadro 2.3 – *Configuration Space* tipo 0x0 (OSDEV, 2020d)

register	**offset**	**bits 31-24**	**bits 23-16**	**bits 15-8**	**bits 7-0**
00	00	Device ID		Vendor ID	
01	04	Status		Command	
02	08	Class code	Subclass	Prog IF	Revision ID
03	0C	BIST	Header Type	Latency Timer	Cache Line Size
04	10	Base address #0 (BAR0)			
05	14	Base address #1 (BAR1)			
06	18	Base address #2 (BAR2)			
07	1C	Base address #3 (BAR3)			
08	20	Base address #4 (BAR4)			
09	24	Base address #5 (BAR5)			
0A	28	Cardbus CIS Pointer			
0B	2C	Subsystem ID		Subsystem Vendor ID	
0C	30	Expansion ROM base addrees			
0D	34	Reserved			Capabilities Pointer
0E	38	Reserved			
0F	3C	Max latency	Min Grant	Interrupt PIN	Interrupt Line

A porta de endereço 0xCF8, denominada CONFIG_ADDR, é um registrador de 32 bits, que possui um formato definido para identificar um grupo de registradores específicos dentro do *Configuration Space* de um dispositivo. O bits deste registrador são organizados da seguinte maneira:

- **Bit 31 (Enable bit):** Bit que indica se o conteúdo deste registrador pode ser repassado para o controlador do PCI;
- **Bits 30-24 (Reserved):** Bits reservados;
- **Bits 23-16 (Bus number):** Bits que indicam o *bus* que o dispositivo está localizado;
- **Bits 15-11 (Device number):** Bits que indicam o dispositivo no *bus* especificado;
- **Bits 10-8 (Function number):** Bits que indicam a função do dispositivo, caso o dispositivo possua mais de uma função;
- **Bits 7-0 (Register offset):** Bits que indicam o deslocamento da área do *Configuration Space* a ser acessado;

2.7 Processadores x86

Segundo Tanenbaum (2012), a programação de computadores modernos é feita utilizando linguagens de alto nível, que a partir de diversos processos de tradução, são transformadas na linguagem disponibilizada pelo processador. Quando um processador é criado, um conjunto de instruções é aceito. Esse conjunto de instruções executa a nível de microarquitetura. Processadores do tipo CISC implementam, em instruções que executam no nível de microarquitetura, um microcódigo, que executa no nível ISA. Esse microcódigo é criado para que novas instruções sejam aceitas pelo processador, sem alteração em hardware. Para esse tipo de processador, a linguagem Assembly é uma representação da linguagem do nível ISA. Em processadores RISC (onde não existe microcódigo), o Assembly é a representação da linguagem de microarquitetura. A programação Assembly é comumente utilizada para softwares de baixo nível e aplicações de tempo real.

O processador Intel 80386 (também conhecido como i386), introduzido no ano de 1985, foi utilizado por vários anos em *workstations* e computadores pessoais (WIKIPEDIA, 2020h), e introduziu um novo modelo de arquitetura de processadores, baseado em 32 bits, que ainda é utilizado atualmente. Tendo em vista sua importância histórica, a utilização atual da arquitetura introduzida por esse processador e sua vasta documentação, este foi utilizado como processador que executará o sistema operacional proposto.

2.7.1 Sintaxes do Assembly x86

Como apresentado em Cloutier (2019), processadores x86 possuem uma vasta quantidade de instruções disponíveis. Existem duas principais sintaxes para o assembly de processadores x86: Intel e AT&T. O Quadro 2.4 apresenta algumas das principais características.

Quadro 2.4 – Principais diferenças entre a sintaxe Intel e a sintaxe AT&T (WIKIPEDIA, 2020q).

	AT&T	**Intel**
Ordem dos parâmetros	Origem antes do destino `mov $5, %eax`	Destino antes da origem `mov eax, 5`
Tamanho dos parâmetros	Mnemônicos são sufixados pela letra que identifica o tamanho dos operandos:*q* para qword, *l* para long (dword), *w* para word e *b* para byte `addl $4, %esp`	Tamanho definido de acordo com o nome do registrador que é usado (Ex.: rax, eax, ax, al implica em q, l, w, b respectivamente) `add esp, 4`
Símbolos	Operandos imediatos são prefixados por "$", registradores são prefixados por "%"	Assembler automaticamente detecta o tipo de simbolo
Endereços efetivos	Sintaxe geral: *DISP(BASE,INDEX,SCALE)* `movl mem_location (%ebx, %ecx, 4), %eax`	Expressões aritméticas entre colchetes; palavras chave de tamanho como *byte*, *word* ou *dword* devem ser adicionadas caso o tamanho não possa ser definido pelos operandos. `mov eax, [ebx + ecx*4 + mem_location]`

2.7.2 Registradores

Registradores são unidades de armazenamento de dados existente dentro do *chipset* do processador, utilizados para armazenamento de informações que necessitam de rápido acesso ou para informações de controle da CPU e do sistema em geral. Em processadores da família x86, existem diferentes grupos de registradores, cada grupo com funções determinadas: registradores de propósitos gerais; registradores ponteiros; registradores de segmento; registrador FLAGS; registradores de controle. Os grupos apresentados, além de alguns complementares, são apresentados em OSDEV (2019b).

Registradores podem ter diferentes tamanhos: 8 bits, 16 bits, 32 bits e 64 bits, de maneira que o nome do registrador indica o seu tamanho. Existem extensões em novos

processadores que permitem registradores de 128 bits, 256 bits e 512 bits, demonstrados em Wikipedia (2020n) e Wikipedia (2020a). Os registradores de tamanhos menores referenciam alguma parte dos registradores maiores. O registrador AX, por exemplo, possui 16 bits. O registrador AL, que possui 8 bits, corresponde aos 8 bits de baixa ordem do registrador AX. O registrador AH, que possui 8 bits, corresponde aos 8 bits de alta ordem do registrador AX. Da mesma maneira, o registrador AX corresponde aos 16 bits de baixa ordem do registrador EAX, que possui 32 bits.

O conjunto de registradores AX, BX, CX, DX, SI, DI, BP e SP, além de seus respectivos registradores de tamanhos distintos, formam o grupo de registradores de propósito geral. Cada um desses registradores possui uma função em determinadas instruções, como mostra o Quadro 2.5

Quadro 2.5 – Registradores de propósitos gerais (OSDEV, 2019b).

32 bits	**16 bits**	**8 bits altos**	**8 bits baixos**	**Descrição**
eax	ax	ah	al	acumulador
ebx	bx	bh	bl	base
ecx	cx	ch	cl	contador
edx	dx	dh	dl	dados
esi	si	N/A	N/A	índice de origem
edi	di	N/A	N/A	índice de destino
ebp	bp	N/A	N/A	ponteiro para base da pilha
esp	sp	N/A	N/A	ponteiro para o topo da pilha

Os registradores ponteiros armazenam o endereço de memória que contem a próxima instrução que deve ser executada pelo processador. Esse registrador não pode ser lido ou escrito diretamente por um programa. O Quadro 2.6 apresenta os registradores ponteiros disponíveis.

Quadro 2.6 – Registradores ponteiros (OSDEV, 2019b).

32 bits	**16 bits**	**Descrição**
eip	ip	pointeiro de instrução

Os registradores de segmento são utilizados para formar endereços de memória. Dependendo do modo de operação da CPU, a utilização desses registradores pode ser diferente. O Quadro 2.7 apresenta os registradores disponíveis para esse fim, além da sua aplicação.

Quadro 2.7 – Registradores de segmentos (OSDEV, 2019b).

16 bits	**Descrição**
cs	segmento de código
ds	segmento de dado
es	segmento extra
ss	segmento da pilha
fs	segmento de propósito geral
gs	segmento de propósito geral

O registrador FLAGS é um registrador que armazena o estado atual do processador. Os registradores EFLAGS e RFLAGS são seus sucessores, tendo 32 bits e 64 bits respectivamente. Algumas instruções fazem uso de algumas *flags* desse registrador, como as *jz* (jump if zero), *jc* (jump if carry) e *jo* (jump if overflow)(WIKIPEDIA, 2020g). As *flags* existentes nesse registrador são apresentadas no Quadro 2.8. Vale ressaltar que os bits que não estão citados são reservados.

Quadro 2.8 – Registrador FLAGS (OSDEV, 2019b).

Bit	**Abreviação**	**Descrição**
0	cf	*Flag* Carry
2	pf	*Flag* Parity
4	af	*Flag* Auxiliary
6	zf	*Flag* Zero
7	sf	*Flag* Sign
8	tf	*Flag* Trap
9	if	*Flag* Interrupt
10	df	*Flag* Direction
11	of	*Flag* Overflow
12-13	iopl	Nível de privilégio de entrada e saída
14	nt	*Flag* Nested task
16	rf	*Flag* Resume
17	vm	*Flag* Virtual 8086 mode
18	ac	*Flag* Alignment check
19	vif	*Flag* Virtual interrupt
20	vip	Interrupções virtuais pendentes
21	id	*Flag* Id

Os registradores de controle são registradores responsáveis por modificar ou controlar o comportamento geral da CPU ou de outro dispositivo (WIKIPEDIA, 2020d). O registrador de controle CR0 possui várias *flags* que modificam as operações básicas da CPU.

Para realizar operações de leitura e escrita nesse registrador, é necessário um processo de duas vias (utilização de registradores auxiliares), diferentemente dos registradores comuns que podem ser acessados diretamente. As instruções lmsw e smsw também podem ser utilizadas para esse fim. O Quadro 2.9 descreve as *flags* desse registrador.

O registrador de controle CR1 é reservado. O registrador de controle CR2 contém um valor chamado *Page Fault Linear Address*. Quando uma interrupção *Page Fault* é gerada, o endereço que o programa tentou acessar é armazenado no registrador CR2. O registrador CR3 é utilizado quando o último bit do registrador CR0 (*Paging bit*) é ativado, sendo utilizado durante a tradução de endereços virtuais para endereços físicos.

O registrador de controle CR4 é utilizado para controlar operações como suporte a virtualização 8086, *I/O breakpoints*, extensões de tamanho de páginas e *machine-check*. Os registradores de controle CR5, CR6 e CR7 são reservados.

Quadro 2.9 – Registrador de controle CR0 (OSDEV, 2019b).

Bit	**Abreviação**	**Descrição**
0	pe	protected mode habilitado
1	mp	Monitor co-processor
2	em	Emulação
3	ts	Troca de Task
4	et	Tipo de extensão
5	ne	Erro numérico
16	wp	Proteção de escrita
18	am	Máscara de alinhamento
29	nw	Não escrever
30	cd	cache desabilitada
31	pg	Paginação

2.7.3 Real Mode

Real Mode é um modo de operação de 16 bits de uma CPU x86. Dentro desse modo, existe menos que 1MB de RAM disponível, não possui qualquer proteção de segurança, os operandos da CPU são de 16 bits (os registradores de 32 bits estão disponíveis para utilização, caso existam) e o acesso a endereços de memória maiores que 64KB só pode ser realizado com o auxílio de registradores de segmentos.

O acesso à memória no *Real Mode* pode ser realizado diretamente a partir de endereços físicos ou utilizando registradores de segmentos. Os registradores de segmentos disponibilizados são CS, DS, ES, FS, GS e SS. Para utilizá-los, basta descrever um endereço de memória a partir da associação do segmento com um deslocamento, como mostrado a seguir:

```
segmento : deslocamento
```

A transcrição para o endereço físico é realizada da seguinte maneira:

```
endereço físico = (segmento * 16) + deslocamento
```

Esse tipo de endereçamento permite o acesso à chamada área alta de memória, aumentando a capacidade de acesso no *Real Mode*

A pilha no *Real Mode* é delimitada por dois registradores de 16 bits: BP e SP. BP aponta para a base da pilha e SP para o topo da pilha. Vale ressaltar que a pilha cresce do maior endereço para o menor endereço, portanto, o valor de BP deve ser sempre maior ou igual a SP. O registrador SS pode ser utilizado como registrador de segmento para definir endereços para a pilha.

2.7.4 Protected Mode

Protected Mode é o modo de operação utilizado comumente pelos processadores de 32 bits. Diferentemente do *Real Mode*, os operandos da CPU são de 32 bits, o que permite um endereçamento de memória composto por 32 bits, possibilitando o controle de até 4GB de memória RAM; o sistema de acesso a memória é mais sofisticado pois suporta paginação; a segurança é efetuada a partir da utilização de anéis de proteção. A mudança do *Real mode* para o *Protected Mode* se dá pela mudança do bit menos significativo do registrador CR0 (*protected mode eneabled*), seguido de um pulo distante. Esse pulo tem como objetivo limpar o *Pipe Line*, para que não existam instruções de um modo executando em outro.

2.7.5 Anéis de privilégio

Anéis de privilégio são mecanismos de segurança que tem como objetivo proteger dados e funcionalidades de falhas e comportamentos maliciosos (WIKIPEDIA, 2020m). Nos processadores x86, esse mecanismo é composto por quatro níveis de privilégio, em que quanto menor o nível, mais privilégios. Quando o processador está no *ring 0* por exemplo, o acesso à qualquer posição de memória está disponível para o código que tem a posse da CPU, assim como a capacidade de executar instruções privilegiadas. Caso o processador esteja em *ring 3* por exemplo, o código que está em execução no momento está restrito ao seu espaço de memória virtual e a um conjunto restrito de instruções que este pode executar.

Posto isso, é comum que sistemas operacionais executem seus códigos com o processador no *ring 0*, já que estes códigos necessitam, em alguma instancia, de acessos

privilegiados. Os processos de usuário são comumente executados em *ring 3*, impedindo que eles acessem diretamente recursos sensíveis, sendo necessária a comunicação desses com o sistema operacional.

O *ring 1* e o *ring 2* não são utilizados por sistemas operacionais modernos. Contudo, tecnologias de virtualização podem usufruir desses níveis para executar sistemas operacionais convidados e *hypervisors*, assim como sistemas operacionais *microkernel* (TANENBAUM; HERDER; BOS, 2006) podem reorganizar partes do seu código entre estes níveis de proteção.

2.7.6 Barramentos

Um barramento é um caminho elétrico dentro de um dispositivo ou comum entre vários componentes (TANENBAUM, 2012), tendo como objetivo transferir dados (WIKIPEDIA, 2020b). Barramentos entre dispositivos exercem uma influência significativa no desempenho do sistema, além de serem responsáveis por permitir que o processador acesse a memória e outros dispositivos. Barramentos antigos como o EISA (WIKIPEDIA, 2020f) e o PCI (WIKIPEDIA, 2020k) transmitem dados de maneira paralela, o que pode parecer uma estrutura que aumenta o desempenho, visto que tem uma largura de banda maior. Contudo, quando os dados são transmitidos em altas taxas de transmissão, o tempo de propagação dos bits em paralelo difere, limitando a velocidade dos barramentos. Esse fenômeno é conhecido como *skew*. Isso levou os projetistas a desenvolverem barramentos seriais como o PCI Express (WIKIPEDIA, 2020j), que mesmo transferindo um bit por vez, conseguem alcançar velocidades muito superiores aos barramentos paralelos, superando a taxa de transmissão de barramentos paralelos.

Barramentos responsáveis por interligar os componentes de hardware são, comumente, compartilhados entre vários dispositivos. Isso implica na necessidade de componentes especializados em gerenciar os barramentos, que chaveiam o acesso entre os dispositivos e o processador. Alguns dispositivos de entrada e saída funcionam de maneira analógica, e o bloqueio do acesso ao barramento pode levar a perca de dados. Por isso, estes devem possuir prioridades maior que o processador. O barramento PCI buscou melhorar esse esquema, adicionando um barramento específico entre o processador e a memória, facilitando a comunicação entre estes. A Figura 2.7 demonstra de forma simples, o esquema lógico de um barramento PCI.

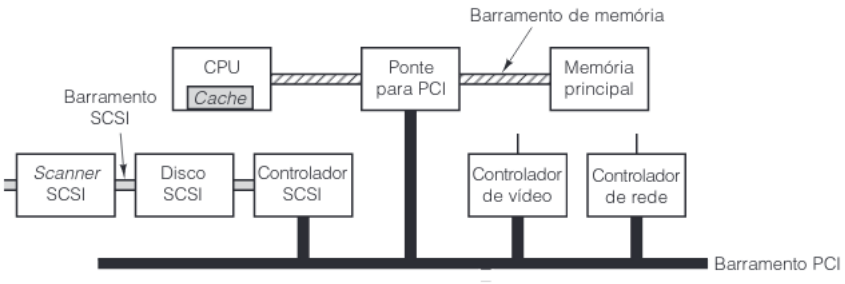


Figura 2.7 – Arquitetura típica montada em torno do barramento PCI. O controlador SCSI é um dispositivo PCI (TANENBAUM, 2012).

3 Estado da Arte

Dentro do contexto acadêmico, existem alguns sistemas operacionais voltados para o aprendizado de seus algoritmos e a execução destes, sendo o MINIX (HERDER et al., 2006), uma das principais referencias no tema. Sistemas operacionais voltados à prática da implementação são mais comuns dentro do contexto da comunidade de desenvolvimento de sistemas operacionais, onde existem sites focados no desenvolvimento destes, tutoriais, e projetos que auxiliam no aprendizado de estudantes.

A visualização e execução de algoritmos na prática é importante para o processo de entendimento dos componentes dos sistemas operacionais, mas o modo de interação destas partes está relacionado à forma que o sistema foi desenvolvido. A partir dessas constatações, estão descritos abaixo o sistema operacional MINIX e outros sistemas operacionais que tem como foco principal proporcionar um objeto de estudo voltado à implementação de sistemas operacionais.

3.1 MINIX

O MINIX (HERDER et al., 2006) é um sistema operacional livre que foi implementado com o objetivo de ser utilizado em universidades para o ensino e pesquisa. Este sistema operacional, assim como suas versões anteriores, serviram como base para a implementação do núcleo Linux, além de ser objeto de pesquisa no campo de arquitetura de sistemas operacionais, principalmente para a arquitetura *microkernel* (TANENBAUM; HERDER; BOS, 2006). A partir do MINIX, diversos projetos de implementação de sistemas operacionais foram iniciados, expandindo a variedade entre os sistemas operacionais e o número de pesquisas na área.

O MINIX é um sistema operacional baseado na arquitetura *microkernel*, que tem como objetivo principal a confiabilidade e segurança. A Figura 3.1 demonstra que os processos do sistema são divididos em camadas, e a maioria delas são executadas com o processador no modo usuário. Desta maneira, o núcleo do MINIX é responsável somente pelo tratamento de interrupções, programação da CPU e da MMU, escalonamento, comunicação entre processos e chamadas do kernel, mantendo o as políticas no modo usuário. A utilização de servidores executando em modo usuário permite que atividades de gerenciamento saiam do kernel, diminuindo a possibilidade de uma falha local gerar uma falha de todo o sistema.

A comunicação entre processos é realizada através de mensagens síncronas de tamanho fixo, ou através de notificações. Isso permite que os processos possam trocar

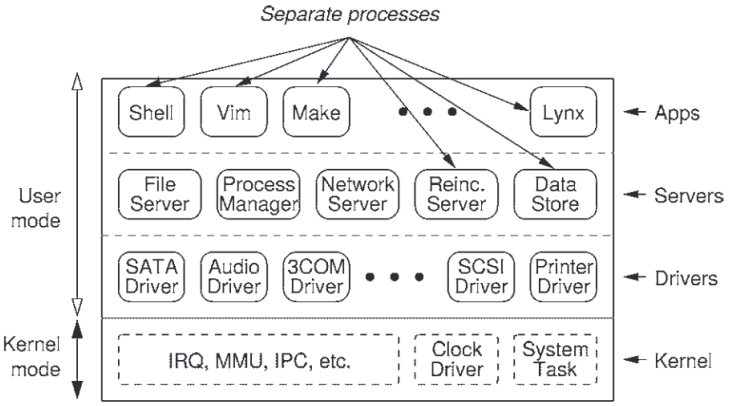


Figura 3.1 – Arquitetura do sistema multiserver do MINIX 3 (HERDER et al., 2006).

mensagens com o intermédio do kernel de maneira segura. Os processos que executam em modo kernel, *clock driver* e *system task*, são responsáveis por lidar com escalonamento e chamadas do kernel, respectivamente. Mesmo implementados no kernel, estes processos são escalonados como processos comuns, em que seus espaços de memória coincidem com o espaço de memória do kernel.

Os *drivers* (com exceção do *clock driver*) estão implementados no espaço de usuário, fazendo com que o acesso à endereços de memória fora do escopo dos seus processos seja negado. Contudo, estes processos são autorizados a realizar chamadas de kernel, que são tratadas pelo *system task*. Essa abordagem foi utilizada pois normalmente, *drivers* são produzidos por terceiros, aumentando a possibilidade de existência de erro em seus códigos. Dessa forma, uma falha em um *driver* não afetará outros componentes do sistema. Os processos servidores, também implementados no nível de usuário, são responsáveis por receber requisições de processos aplicativos do usuário, comunicar os *drivers* caso necessário, e responder o processo chamador, utilizando chamadas do kernel.

Com o objetivo de manter o sistema mais confiável e seguro, o MINIX implementa dois processos servidores especiais: *reincarnation server* e *data store*. O processo *reincarnation server* é o processo pai de todos os *drivers*. Desta maneira, assim como em sistemas UNIX, o processo pai é notificado quando um processo filho termina sua execução, e sabe status de término. Quando um *driver* apresenta problemas, o *reincarnation server* se encarrega de reinicializá-lo, e com a ajuda do *data store*, recuperar o seu estado, antes da ocorrência do erro.

3.2 Sistema Operacional do grupo BrokenThorn

O grupo BrokenThorn (2008) desenvolveu um documento que tem como objetivo orientar o leitor a implementar um sistema operacional completo. Este documento, composto por 34 tópicos organizados em 8 estágios, possui uma forte base teórica associada à

códigos apresentados quando necessário.

O primeiro estágio é composto 7 tópicos: o primeiro e o segundo estão relacionados aos conhecimentos prévios necessários para o leitor do documento, e às ferramentas necessárias. Dentro do conjunto de ferramentas está o compilador GCC, Assembler NASM para a linguagem Assembly x86 (Intel syntax) e o BOCHS para execução o projeto. O terceiro tópico aborda aspectos históricos de sistemas operacionais, atrelados à alguns conceitos básicos como gerencia de memória, gerencia de tarefas, proteção de memória, núcleo, sistema de arquivos, shell, interfaces gráficas, e *bootloaders*. Os seguintes tópicos abordam a construção do *boot sector* em si.

O segundo estágio, dividido em 5 tópicos é responsável por apresentar a arquitetura atual do sistema e modificar o modo de operação do processador para *Protected Mode*, a fim de preparar o ambiente para a execução do núcleo do sistema. Esse ambiente é explorado no terceiro estágio, que é totalmente composto por conceitos teóricos. Nele são apresentados diversas arquiteturas de sistemas operacionais, assim como suas funcionalidades e aplicações.

O quarto estágio, composto por 12 tópicos, abrange as partes principais do documento. Inicialmente, questões de padronização são expostas, acompanhadas da explanação dos métodos de tratamento de interrupções, gerenciamento de memória física, gerenciamento de memória virtual, programação do teclado (com foco em hardware), base teórica sobre DMA e FDC, sistema de arquivos, espaço de usuário e gerência de processos.

O quinto estágio aborda a utilização de binários PE na implementação, enquanto que o sexto é responsável por descrever, a nível de hardware, os dispositivos utilizados para receber IRQs (PIC) e o dispositivo utilizado como *timer* (PIT). O sétimo estágio aborda o processo de implementação de um ambiente gráfico, seguido do ultimo estágio, que apresenta padrões de outras plataformas computacionais.

3.3 Sistema Operacional de James Molloy

Molloy (2008) apresenta um documento que tem como objetivo guiar o leitor durante o desenvolvimento de um sistema operacional estilo UNIX. Mesmo apresentando somente parte de códigos, Molloy mescla teoria e prática dentro dos dez tópicos que dividem o material.

No primeiro momento, Molloy apresenta a preparação das ferramentas que ele utiliza durante durante o documento. Nenhum tipo de compilador cruzado é apresentado, indicando somente que a linguagem C e Assembly x86 (sintaxe AT&T) são utilizadas, ao lado do compilador GCC, linker LD e Assembler GAS. Para demonstrar a execução dos módulos apresentados, Molloy utiliza o BOCHS como plataforma de execução, assim

como suas configurações. Em seguida, Molloy apresenta a construção do processo de *boot* utilizando o GRUB (GNU, 2012). Já que o GRUB é um sistema já implementado, o processo de *boot* consiste apenas na configuração desta ferramenta, dispensando a implementação de um *boot sector* próprio.

O *driver* de vídeo é apresentado a partir da utilização do mapeamento do VGA na memória, seguido da explicação do funcionamento das tabelas GDT e IDT. Com essas estruturas de dados abordadas, Molloy exibe a forma com que as interrupções devem ser tratadas. A paginação neste sistema é realizada a partir da utilização de uma tabela de páginas de dois níveis (tabela de páginas e tabela de diretórios), e o método de habilitação da memória virtual segue o padrão definido pelo processador i386. Uma área de *heap* é definida, utilizando uma lista encadeada para mapear os endereços livres, e funções para alocar e liberar endereços. Com o objetivo de acessar os dispositivos de armazenamento de maneira padronizada e criar um nível de abstração para a existência de arquivos, Molloy reserva um espaço em seu documento para a apresentação de um VFS, assim como os componentes que o compõe. Um sistema de multitarefas é idealizado em conjunto com a habilitação do modo usuário, demonstrando o compartilhamento de áreas de memória do kernel para as *tasks* de usuário, troca do diretório de páginas, mudança de pilha e inicio da nova *task*.

Existem diversos projetos na comunidade que implementam sistemas operacionais seguindo o documento de Molloy. A implementação de Fenollosa (2018), que também é influenciada pelo trabalho de Blundell descrito na seção seguinte, se destaca por dois principais motivos: com o objetivo de fazer com que seu trabalho fosse incremental, Fenollosa dividiu seu projeto em 24 partes, seguindo a sequência de implementação do Molloy até a implementação da área da *heap*; Fenollosa buscou resolver a série de problemas encontrados no material de Molloy, definidas em OSDEV (2020c).

Dentre os 21 grupos de erros relatados se destacam os seguintes: não utilização de um compilador cruzado, levando à utilização errada das *flags* de compilação; erro na definição do tipo de convenção de chamadas de funções; não definição prévia do local da pilha; quebra do padrão ABI ao criar a estrutura de tratamento de interrupções localmente; erro ao empurrar códigos de erro durante o tratamento das interrupções 17 e 21; erro de alinhamento na função de alocação de memória na área *heap*; definições de *inline Assembly* diferem do Assembly puro.

3.4 Sistema Operacional de Nick Blundell

Blundell (2010) apresenta sistema operacional com poucas funcionalidades implementadas, mas com uma forte base teórica, exposta de maneira didática e incremental. Conceitos teóricos básicos são expostos para melhor entendimento do leitor, como: fun-

cionamento do sistema de numeração hexadecimal; conceitos da linguagem Assembly; conceito de registradores; processo de compilação; arquitetura de um disco rígido; acesso à memória principal. O sistema de numeração hexadecimal é elucidado a partir de conversões de números binários para suas representações decimais e hexadecimais. A linguagem Assembly é destrinchada a medida que os códigos são escritos, assim como a utilização dos registradores disponíveis.

O processo de desenvolvimento se inicia com a implementação de um *boot sector* e a explanação do funcionamento de processo de *boot*, desde o ambiente inicial *16-bit Real Mode* até o ambiente *32-bit Protected Mode*. Blundell expõe a implementação do setor de *boot* a partir de um dispositivo *floppy*, escrevendo-o em bytes puros inicialmente, seguido da implementação Assembly. Blundell apresenta, por meio da Figura 3.3, a disposição dos endereços de memória após o carregamento do código executável. Em seguida, Blundell finaliza seu trabalho implementando um driver para o vídeo, utilizando o mapeamento do VGA no *Protected Mode*.

O processo de compilação é descrito a partir da análise dos códigos gerados pelo compilador, além da descrição das *flags* utilizadas. A explicação da arquitetura de discos rígidos foi realizada em torno da Figura 3.2, cobrindo a forma de armazenamento, busca por dados, leitura e escrita. O acesso à memória é realizado a partir de instruções Assembly e através de ponteiros na linguagem C.

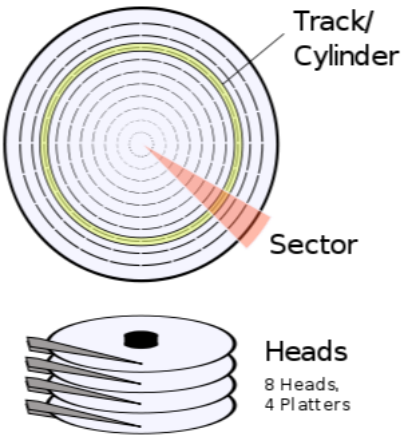


Figura 3.2 – Estrutura de cilindro, cabeça de leitura e setores de um disco (BLUNDELL, 2010, p. 26).

3.5 Sistema operacional de Samy Pessé

Pessé (2015) apresenta uma implementação de um sistema operacional em C/C++, parcialmente modularizado e produzido de maneira incremental. O sistema operacional

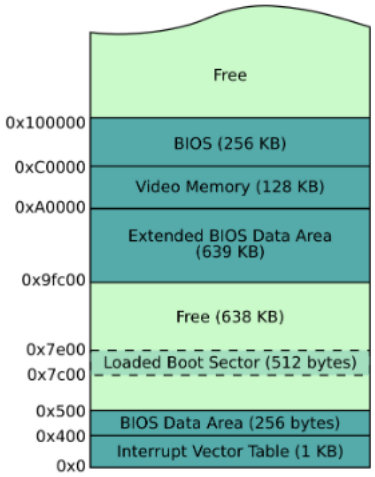


Figura 3.3 – Layout típico da baixa memória após o *boot* (BLUNDELL, 2010, p. 14).

em si é completo, mas as seções modularizadas, divididas em 9 capítulos, abrangem somente a preparação de ferramentas, processo de *boot* com GRUB, criação de runtime C++, criação de interfaces para chamadas de funções Assembly, criação da tabela GDT e IDT, base teórica para gerenciamento de memória física e virtual e a implementação do gerenciamento de memória virtual. Para cada capítulo de implementação, Pessé escreveu pequenos textos, com figuras e tabelas, que introduzem conceitos necessários para a implementação do módulo em questão. Fora da parte mapeada em módulos, o sistema conta com gerenciamento de processos e multitarefas, execução de binários ELF, espaço de usuário e chamadas de sistema, *drivers* modulares, *driver* para discos rígidos IDE, partições DOS, sistema de arquivos EXT2 somente leitura, biblioteca padrão C, ferramentas básicas UNIX (sh e cat) e um interpretador de Lua.

Tendo em vista que este trabalho foi escrito em C++, a organização deste difere um pouco do padrão de construção dos anteriores, pois a utilização de funcionalidades presentes auxilia na criação de classes, sobrecarga de operadores, entre outros.

Uma classe foi criada para implementação de funções de tratamento de vídeo, com as funções putc e print, que imprimem um caractere e strings formatadas, respectivamente. A string formatada implementada suporta impressão de ponteiros, inteiros expressos em decimal e hexadecimal e inserção de strings recebidas por parâmetro.

A GDT e a IDT foram escritas a partir de estruturas escritas em C++, seguindo a formatação padrão, sendo carregadas a partir de código *inline Assembly*. A atribuição dos registradores de segmento foi realizada através do mesmo mecanismo, enquanto o tratamento de interrupções foi feito em Assembly puro.

A descrição do funcionamento da memória virtual foi realizada a partir da Figura 3.4 e Figura 3.5, onde são expostas a arquitetura de dois níveis de tabela de páginas, a tradução, com auxílio do registrador CR3, de endereços virtuais para endereços físicos e o mapeamento

inicial da memória virtual, onde os primeiro 4MB de memória virtual coincidem com os 4MB da memória física. Pessé organizou os 9 capítulos em um livro (PESSé, 2018) contendo somente o conteúdo teórico referente a cada parte do sistema operacional que foi modularizada no seu trabalho principal;

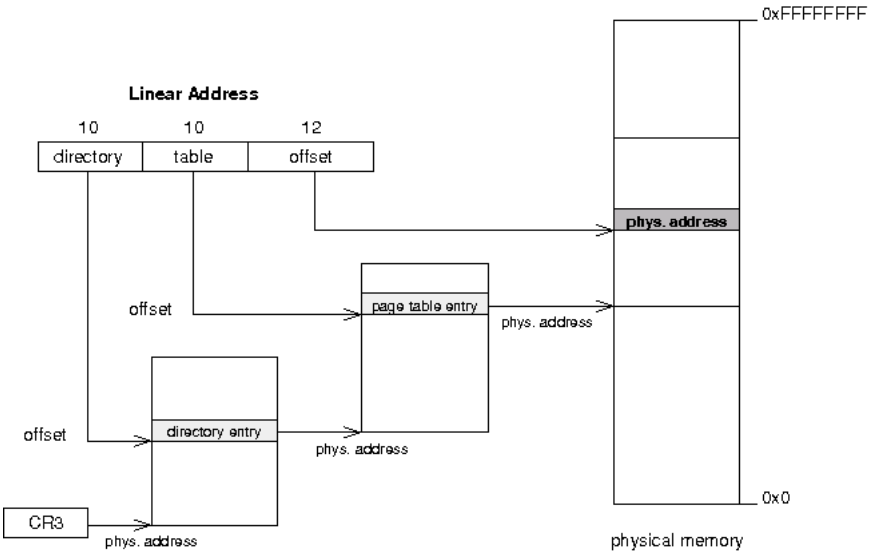


Figura 3.4 – Arquitetura de dois níveis de tabelas de páginas do processador i386 (PESSé, 2015).

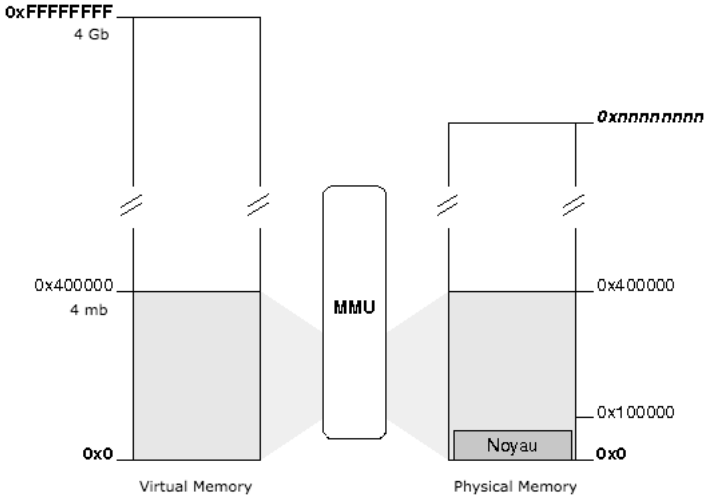


Figura 3.5 – Mapeamento inicial da memória virtual (PESSé, 2015).

4 Implementação de um Sistema Operacional Experimental Incremental e Modular

A implementação de um sistema operacional requer, além da base teórica que aborda todos os conceitos necessários para sua implementação, uma descrição robusta dos componentes de hardware que compõem o conjunto de elementos sob controle do sistema operacional. Tecnologias novas, além de serem complexas, ainda não dispõem de vastas documentações disponíveis, diferentemente de tecnologias que já possuem um tempo considerável de utilização e contam com diversos instrumentos de estudo que podem auxiliar durante o entendimento destas tecnologias. Por este motivo, o processador Intel 80386 (i386) foi escolhido como plataforma alvo para o sistema operacional implementado.

Neste trabalho, nome de funções, variáveis, comandos *shell*, *flags* de comandos e instruções são apresentadas na cor vermelha para maior destaque. Registradores são escritos em letras maiúsculas. Nomes de arquivos são expressos com o caminho de trabalho completo, partido do diretório raiz do projeto. Ao se iniciar o texto de cada módulo, será apresentada uma breve explicação deste, seguido da sua interface. O texto subsequente é referente à explicação da construção desta interface. A dependência entre os módulos é apresentada através do grafo direcionado contido na Figura 4.1, em que um módulo A depende de um módulo B quando existe uma aresta de B para A, ou uma aresta de algum módulo dependente do módulo B para A.

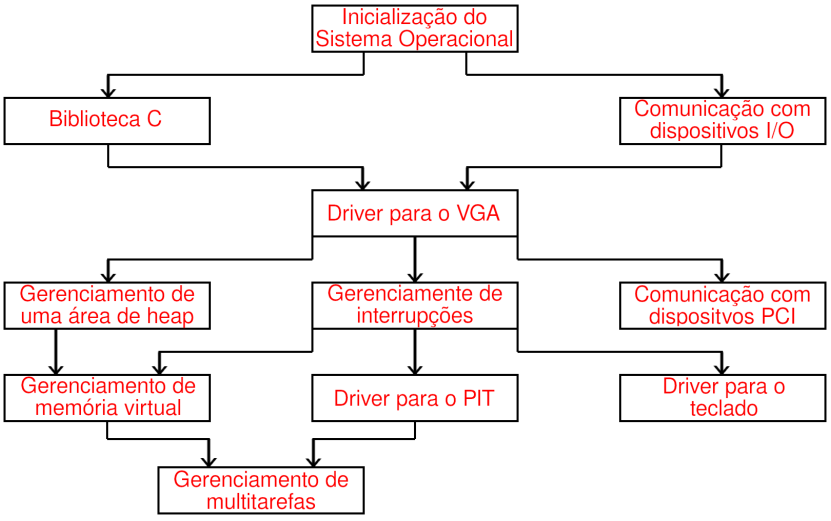


Figura 4.1 – Grafo de dependência entre os módulos.

4.1 Preparação de ferramentas e ambiente de desenvolvimento

4.1.1 Linguagem C

C é uma linguagem de programação minimalista, independente de plataforma e não requer *runtime* (OSDEV, 2018b). Diante da simplicidade proposta, a linguagem de programação C se tornou comum no meio de desenvolvimento de sistemas operacionais. A linguagem C disponibiliza ao programador um controle de mecanismos de baixo nível maior que outras linguagens de programação de alto nível. Ultimamente, a linguagem de programação Rust também tem se mostrado uma boa opção para desenvolvimentos de sistemas de baixo nível, pois presa por características essenciais para esse tipo de software, como performance e confiabilidade (BALASUBRAMANIAN et al., 2017). Mesmo desfrutando da simplicidade da linguagem C, a sua independência de plataforma faz com que procedimentos altamente dependentes da plataforma precisem ser implementados diretamente em Assembly.

4.1.2 Compilador

Um compilador de uma linguagem de alto nível geralmente produz código para para uma plataforma específica (hardware e sistema operacional). Tendo em vista que este trabalho foi realizado utilizando um processador Intel(R) Core(TM) i7-3537U e um sistema operacional com núcleo Linux, utilizar o mesmo compilador que executa nessa plataforma para gerar código para o novo sistema operacional causaria diversos problemas. Portanto, se faz necessária a utilização de um compilador cruzado (*Cross-Compile*). Um compilador cruzado gera código para uma plataforma diferente da que ele está sendo executado. Desta maneira, o compilador utilizado executa na plataforma Intel(R) Core(TM) i7-3537U com núcleo Linux e gera código para a plataforma Intel 80386. Vale ressaltar que não é especificado o sistema operacional alvo que o compilador cruzado gerará código, pois o objetivo é construir um. Dessa maneira, *flags* passadas para o compilador durante a sua compilação e durante a compilação do código do sistema operacional indicam esse cenário.

Quando o código do compilador GCC é compilado diversas *flags* são definidas, de forma que se faz necessário um novo processo de compilação para alterá-las. A criação de um compilador cruzado requer o download do código fonte do compilador, definição das *flags* e compilação. Andrewchen e Thatmadhacker (2020b) e Andrewchen e Thatmadhacker (2020a) disponibilizam uma forma de obtenção do compilador cruzado para usuários do sistema operacional Arch Linux e outros sistemas derivados. Com o objetivo de disponibilizar para outras plataformas, um script escrito em *shell*, denominado *i686-elf/i686-elf-install.sh* foi criado, para que seja possível a construção do compilador cruzado. Este script recebe como parâmetro a versão do GCC desejável, e a versão do *binutils* (conjunto de ferramentas auxiliares utilizado no processo de compilação e *linking*) desejável. A lista de versões

do GCC são encontradas em GNU (2020b), e a lista de versões do *binutils* podem ser encontradas em GNU (2020a).

Implementado com base em OSDEV (2020a), o script baixa os códigos necessários e compila com as seguintes *flags*: –disable-nls é opcional, mas reduz dependências e tempo de compilação pois ela indica ao GCC e ao *binutils* não incluírem suporte à língua nativa; –with-sysroot habilita o suporte à *sysroot*, permitindo a mudança do diretório raiz utilizado para localizar arquivos; –without-headers informa o GCC que as bibliotecas C podem não estar presentes para a compilação; –enable-languages=c,c++ indica que o GCC construído suportará somente linguagens C e C++; –target=i686-elf indica que o resultado da compilação padrão utilizando o compilador cruzado será binários ELF para processadores i386; –prefix=$PWD/i686-elf/opt/cross indica que todos os arquivos do compilador cruzado estarão nesse diretório.

Além das *flags* definidas durante o processo de compilação do compilador, se faz necessária a passagem de outras informações no momento da compilação do código do sistema operacional. A *flag* -g foi utilizada para geração de informações de *debug*. A *flag* -c indica que o processo de *linking* não seja realizado juntamente com o processo de compilação. Existem dois tipos de ambiente de compilação no GCC: *Hosted*, que já é utilizada por padrão, e *Freestanding*, habilitada através da *flag* -ffreestanding. No ambiente *Hosted*, toda a biblioteca padrão C está disponível para o código que será compilado, enquanto que no ambiente *Freestanding*, a biblioteca padrão C pode não está presente, com exceção de poucos arquivos de cabeçalho contendo alguns *defines* e tipos, e o ponto de entrada de execução não necessariamente precisa ser a função denominada main. Em programas de espaço de usuário, algumas funções são executadas antes e depois da função main, com o objetivo de preparar o ambiente de execução, e realizar o retorno do programa. Tendo em vista que não se faz necessária a existência dessas funções para o código de um sistema operacional, a *flag* nostdlib foi utilizada para a não inclusão destas. Esta *flag* também desabilita a inclusão da biblioteca padrão C. A desassociação da biblioteca C está ligada à desassociação do sistema operacional alvo do compilador, fazendo com que somente o processador seja utilizado para este objetivo. Quando a opção nostdlib é utilizada, a *libgcc*, que é uma biblioteca utilizada no processo de compilação pelo GCC, também é desabilitada. Algumas operações que o compilador realizada podem requerer funções dessa biblioteca, portanto, a *flag* lgcc deve ser utilizada para que esta biblioteca esteja disponível. Para a inclusão do diretório que contem os arquivos de cabeçalho escritos neste trabalho na lista de diretórios que são utilizados pelo GCC, a opção -I include foi utilizada, adicionando o diretório *include* à esta lista.

4.1.3 QEMU

Como um hardware físico i386 não estava disponível durante o processo de produção deste trabalho, a utilização de um emulador de hardware se fez necessária para que o processador e todos os dispositivos agregados (PIC, PCI, PIT, *floppy*, memória e hardwares auxiliares) fossem emulados, permitindo a execução do sistema operacional.

QEMU é um software livre que pode desempenhar o papel de emulador de hardware ou de um virtualizador (QEMU, 2020). Quando executado como emulador de hardware, o QEMU permite, a partir de tradução dinâmica, emular diferentes máquinas. Como virtualizador, o QEMU pode ser executado juntamente com o *xen hypervisor* (XEN, 2018), ou com o KVM, que permite a utilização dos recursos de virtualização do processador, de maneira que as instruções do ambiente virtualizado sejam executadas diretamente pelo processador.

A versão 5.1.0 da implementação do QEMU, que disponibiliza uma arquitetura i386, foi utilizada para prover os elementos de hardware necessários da seguinte maneira: qemu-system-i386 -fda os-image.bin, em que a *flag* -fda os-image.bin indica a inserção de um *floppy*, que tem o mesmo conteúdo do arquivo *os-image.bin*.

4.1.4 GNU Make

O GNU Make é uma ferramenta que controla a geração de códigos executáveis e outros códigos não fontes de um programa, a partir dos seus códigos fontes (GNU, 2020c). Considerando que a quantidade de comandos a se executar para compilar e executar o sistema operacional cresce na proporção que as funcionalidades são adicionadas, o GNU Make se torna bastante útil em organizar este processo. O GNU Make necessita de um arquivo que define suas regras e suas ações, chamado *Makefile*. É proporcionado também a criação de macros que auxiliam no processo de escrita deste arquivo, que tem como objetivo deixa-lo mais organizado, permitindo previamente a definição do compilador que será utilizado, as *flags* passadas para o compilador e regras utilizando expressões regulares. A definição das regras e das ações do *Makefile* serão apresentadas ao decorrer da implementação dos módulos.

4.2 Módulo 0: Implementação de um mecanismo de inicialização do Sistema Operacional

Tendo em vista que o objetivo deste módulo é a realização de todo procedimento que antecede a execução do núcleo em si, sua interface é definida através da seguinte demanda: a função de entrada do núcleo deve ser executada com o processador no modo *Protected Mode*.

O primeiro código implementado, de acordo com o modelo de implementação *bottom-up*, foi o *boot/boot_sector.bin*. Este arquivo contém o código executável presente no setor de *boot* do dispositivo de armazenamento, que no escopo deste projeto é um *floppy*. Dessa maneira, os bytes 0x0 até 0x1FD são preenchidos com código executável, e os bytes 0x1FE e 0x1FF são preenchidos com os valores 0x55 e 0xAA respectivamente, formando a assinatura de *boot*.

O ambiente deixado pelo BIOS (firmware de inicialização utilizado neste trabalho) após o repasse do controle para o código do setor de *boot* consiste no modo de operação *Real Mode* de 16 bits do processador, além da disponibilidade de uma série de funções chamadas a partir de interrupções. Não se pode assumir valores de registradores deixados pelo BIOS, exceto pelo registrador DX que armazena o identificador do dispositivo utilizado para obtenção do setor de *boot*.

4.2.1 Teste da assinatura de *boot*

Inicialmente, este módulo tem como objetivo verificar se a assinatura de *boot* está sendo reconhecida pelo BIOS, de forma que o código *boot/boot_sector.bin* contém apenas um *loop* infinito. O Código 4.1 apresenta a representação ASCII deste código. Com o objetivo de otimizar espaço do texto, 29 linhas contendo bytes 0 foram substituídas por uma linha contendo "***".

Código 4.1 – Primeira versão do setor de *boot* em bytes puros

```
00000000: ebfe 0000 0000 0000 0000 0000 0000 0000
00000010: 0000 0000 0000 0000 0000 0000 0000 0000
***
000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa
```

O código do setor de *boot* contém apenas um *loop* infinito que é executado pelos dois primeiros bytes. O processador i386 disponibiliza a instrução jmp, que modifica o fluxo normal de execução de um programa. Esta modificação pode ser executada de diferentes maneiras, fazendo com que esta instrução possua vários *opcodes* distintos. O *opcode* 0xEB indica que um *short jump* será executado. Essa instrução não recebe como parâmetro o endereço absoluto para onde o registrador IP deve apontar, mas sim um deslocamento de 8 bits que leva em consideração o sinal, que terá como base a instrução atual. Tendo em vista que a instrução jmp do tipo *short jump* ocupa 2 bytes (um para o *opcode* e um para o deslocamento), o *loop* infinito será executado realizando um deslocamento negativo de 2 bytes em relação ao valor do registrador IP, que estará apontando para a instrução subsequente. Como o processador i386 utiliza complemento de 2 para representar números negativos, esse deslocamento é indicado pelo valor 0xFE.

Para a execução deste código, o seguinte comando foi executado: qemu-system-i386 -fda boot/boot_sector.bin. A Figura 4.2 apresenta o resultado da execução.

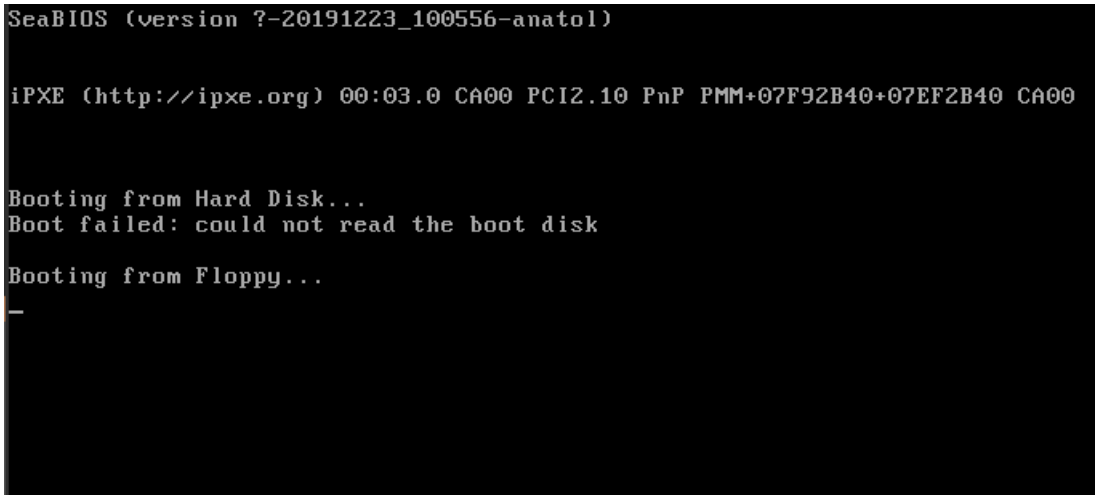


Figura 4.2 – Teste da primeira versão do setor de *boot*

A Figura 4.2 demonstra que o BIOS, inicialmente, faz a busca pela assinatura de *boot* no HD. Por não existir um HD, o BIOS passa para o próximo dispositivo, o *floppy*. Ao encontrar a assinatura de *boot*, o BIOS carrega e executa o código executável na memória, que contém o *loop* infinito.

É possível escrever o setor de *boot* utilizando Assembly. A linguagem de montagem é preferível pois, por ser uma representação das instruções oferecidas pelo processador, a programação é facilitada, sem perder o controle das funcionalidades de baixo nível. A primeira versão do setor de *boot* escrita em Assembly é demonstrada no Código 4.2, escrito no arquivo *boot/boot_sector.asm*.

Código 4.2 – Primeira versão do setor de *boot* em Assembly

```
1 jmp $
2 times 510 - ($-$$) db 0
3 dw 0xaa55
```

O comando nasm boot/boot_sector.asm -o boot/boot_sector.bin compila o código presente em *boot/boot_sector.asm* e escreve o resultado no arquivo *boot/boot_sector.bin*. O código gerado é idêntico ao apresentado no Código 4.1, visto que a instrução jmp $ é responsável por executar a instrução em hexadecimal 0xEBFE, a instrução times 510 - ($-$$) db 0 é responsável por preencher o arquivo com bytes de valor 0 até o byte 509, sobrando dois bytes que serão escritos com o comando dw aa55, formando a assinatura de *boot*.

4.2.2 Impressão utilizando o BIOS

O BIOS disponibiliza funções que permitem ao código do setor de *boot* a manipulação de diversos dispositivos a partir de interrupções. A forma de utilização das funções

providas pelo BIOS se assemelha às chamadas do sistemas de sistemas operacionais, onde parâmetros que definem o modo de operação das funções são definidos em registradores, assim como o retorno destas. A utilização dessas funções para a manipulação do VGA é de extrema utilidade para debug, pois estas permitem a impressão de valores na tela.

A interrupção 0x10 proporciona uma vasta quantidade de operações que podem ser utilizadas para manipulação de vídeo. A definição do valor 0x0E no registrador AH indica ao BIOS que a operação é de impressão de um caractere na posição atual do cursor, seguida do avanço em uma posição do mesmo. O caractere a ser impresso é esperado no registrador AL. A partir de uma iteração, pode-se utilizar esta interrupção do BIOS para imprimir strings, como mostra o Código 4.3.

Código 4.3 – Impressão de *string* utilizando o BIOS

```
1  print_string_rm:
2    pusha
3    mov ah, 0x0e
4    print_string_rm_loop:
5      mov al, [bx]
6      cmp al, 0
7      je end
8      int 0x10
9      add bx, 1
10     jmp print_string_rm_loop
11   end:
12     popa
13     ret
```

A função do Código 4.3, escrita no arquivo *boot/print_string_rm.asm*, espera que o endereço de início da string a ser impressa esteja armazenado no registrador BX, que é iterada até que o valor 0x0 seja encontrado. É importante frisar a utilização as instruções pusha e popa, que podem ser utilizadas para salvar o contexto dos registradores de propósito geral. A instrução pusha é responsável por colocar na pilha os valores dos registradores AX, CX, DX, BX, SP original, BP, SI, e DI. A instrução popa escreve os valores que estão na pilha nos registradores de propósito geral nesta sequência: DI, SI, BP, BX, DX, CX, e AX. O valor de SP não é restaurado, sendo ignorado nesta sequência. Vale ressaltar também a utilização da instrução cmp, que realiza a subtração dos valores passados como parâmetro, de forma que se o resultado da operação for zero, ou seja, os valores forem iguais, a *flag Zero Flag* do registrador FLAGS é ativa. A instrução je (*jump if equal*) funciona de maneira análoga a instrução jz (*jump if zero*). Ambas analisam a *flag Zero Flag* do registrador FLAGS, e realizam o salto caso esta esteja ativa.

É comum em processos de debug a análise de valores na memória e nos registradores, sendo imprescindível a criação de uma função com este objetivo. Com o auxílio da função de impressão de strings, o Código 4.4, escrito no arquivo *boot/print_hex.asm*, imprime em

hexadecimal o valor presente no registrador DX.

Código 4.4 – Impressão de valores em hexadecimal

```
print_hex:
  pusha
  mov ax, 2
  mov cl, 12
  print_hex_loop:
    push dx
    shr dx, cl
    and dx, 0x000f
    mov bx,dx
    mov dx, [MAP+bx]
    mov bx,ax
    mov [MSG+bx], dl
    pop dx
    add ax, 1
    cmp ax, 6
    je end_hex
    sub cl, 4
    jmp print_hex_loop
  end_hex:
    mov bx, MSG
    call print_string_rm
    popa
    ret
  MAP:
    db '0123456789abcdef'
  MSG:
    db '0x????', 0
```

Este código analisa o valor recebido por parâmetro a cada 4 bits, pois é a quantidade de bits que um algarismo hexadecimal representa. Uma string denominada MAP armazena em cada índice a representação hexadecimal deste, de forma que o valor dos quatro bits analisados no momento é utilizado como índice de acesso à string MAP, e o valor neste índice é colocado na string MSG na posição correta. Ao final da análise do parâmetro, a string MSG é passada para função de impressão de string.

Estas duas novas funções, que estão escritas em arquivos isolados, devem ser utilizadas em conjunto com o código do setor de *boot*. Para isso, algumas mudanças no arquivo *boot/boot_sector.asm* se fazem necessárias, como mostra o Código 4.5.

Código 4.5 – Segunda versão do setor de *boot*

```
[org 0x7c00]
mov bp, 0x9000
mov sp, bp
```

```
4  mov bx, DEBUG_MSG
5  call print_string_rm
6  mov dx, 0xabcd
7  call print_hex
8  jmp $
9  %include "print_string_rm.asm"
10 %include "print_hex.asm"
11 DEBUG_MSG db "Printing a hex value: ", 0
12 times 510-($-$$) db 0
13 dw 0xaa55
```

Este código introduz alguns conceitos importantes, como a diretiva [org 0x7c00]. O BIOS carrega o código de setor de *boot* no endereço 0x7C00, e como o Assembler trata o código como se ele começasse do endereço 0x0, a diretiva org indica que deve-se esperar que o código seja carregado no endereço passado por parâmetro. Como os novos códigos utilizam a pilha, se faz necessária a inicialização desta, definindo os ponteiros de base e topo para algum valor livre da memória. A diretiva %include pode ser vista como se o código presente no arquivo passado como parâmetro fosse copiado para o local da definição, integrando novos códigos. A compilação é realizada a partir do comando nasm boot/boot_sector.asm -o boot/boot_sector.bin, e o comando qemu-system-i386 -fda boot/boot_sector.bin executa o QEMU com o novo setor de *boot*, como mostra a Figura 4.3.

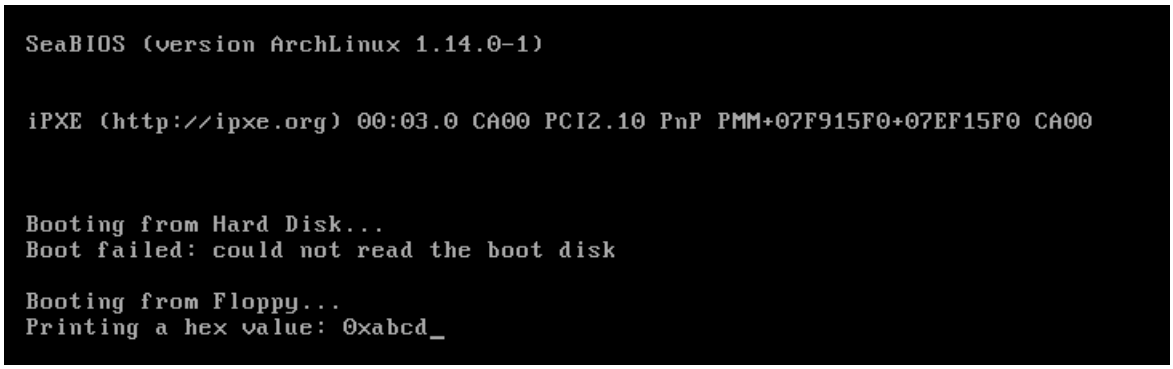


Figura 4.3 – Teste da segunda versão do setor de *boot*

4.2.3 Carregamento de código a partir de um dispositivo de armazenamento

Considerando a restrição do setor de *boot* possuir somente 512 bytes, o sistema operacional não pode estar contido nele. Portanto, se faz necessária a utilização de mais espaço do dispositivo de armazenamento para manter este código. O BIOS disponibiliza funções de tratamento de dispositivos de armazenamento por meio da interrupção 0x13. A organização do parâmetros nos registradores segue o seguinte modelo:

- Registrador AH com valor 0x2, indicando que a chamada da interrupção 0x13 executará uma leitura no dispositivo identificado pelo registrador DL (escrito pelo BIOS durante a sua execução);
- Registrador AL com a quantidade de setores a serem lidos. Esse valor varia de acordo ao crescimento do sistema operacional;
- Registrador CH com os 8 bits de baixa ordem do número do cilindro inicial a ser lido;
- Bits 6 e 7 do registrador CL com os 2 bits de alta ordem do número do cilindro inicial a ser lido;
- Bits 0 a 5 do registrador CL com o número do primeiro setor a ser lido;
- Registrador DH com o número da superfície a ser lida;
- Registrador ES com o segmento de memória em que os dados serão carregador;
- Registrador BX com o deslocamento dentro do segmento especificado em ES, onde os dados serão carregados.

Após a efetuação da leitura, o BIOS escreverá no registrador AL a quantidade de setores que foram lidos. A *flag Carry Flag* será definida como 1, caso algum erro ocorra.

Para implementar esta funcionalidade, o arquivo *boot/disk_load.asm*, contendo o Código 4.6 foi criado, e uma diretiva %include adicionada referenciando esse arquivo no *boot/boot_sector.asm*. Esta função espera como parâmetro o número de setores no registrador DH, o segmento de memória onde os dados serão escritos em ES, e o deslocamento dentro do segmento em BX. A ocorrência de erros é verificada através da *flag Carry Flag* e através da comparação da quantidade de setores lidos com a quantidade de setores requisitados.

Código 4.6 – Leitura de setores utilizando o BIOS

```
1  disk_load:
2    push dx
3    mov ah, 0x02
4    mov al, dh
5    mov cl, 2
6    mov ch, 0
7    mov dh, 0
8    int 0x13
9    jc disk_error_1
10   pop dx
11   cmp al, dh
12   jne disk_error_2
```

```
13   ret
14 disk_error_1:
15   mov bx, DISK_ERROR_MSG_1
16   call print_string_rm
17   jmp $
18 disk_error_2:
19   mov bx, DISK_ERROR_MSG_2
20   call print_string_rm
21   jmp $
```

4.2.4 Habilitação do *Protected Mode*

A execução do sistema operacional em um ambiente *Protected Mode* é mais segura e eficiente, tendo em vista as funcionalidades disponíveis neste modo. A sua habilitação implica na perca de funcionalidades do BIOS, de modo que todas as operações que dependam delas devem ser executadas antes da ativação do *Protected Mode*. A habilitação do novo modo de operação consiste nos seguintes passos: desativação das interrupções; carregamento da GDT; alteração do bit *protected mode eneable* do registrador CR0; execução de um *far jump* para o código que executa no novo modo de operação do processador.

As funções de tratamento de interrupções são inicialmente alocadas pelo BIOS, no endereço de memória 0x0 e ocupam 1024 bytes. Ao executar a instrução cli, a *Interrupt Flag* do registrador FLAGS é definida como 0, desabilitando todas as interrupções externas. A instrução sti habilita o tratamento de interrupções externas pelo processador. A desabilitação das interrupções se faz necessária pois o modo de tratamento de interrupções no *Protected Mode* necessita do carregamento da IDT. Caso as interrupções não sejam desabilitadas, a geração de uma interrupção pelo PIT, por exemplo, gerará uma *Triple Fault* (resultando na reinicialização do processador), pelo fato de o processador não saber onde as rotinas de tratamento de interrupções estão definidas.

A GDT é uma tabela que contém descritores dos segmentos de memória que existem no sistema operacional. Diferentemente do *Real Mode*, os segmentos no *Protected Mode* não possuem somente o endereço de início do segmento, mas também diversos atributos que definem sua forma de tratamento. Cada entrada da tabela GDT é composta por 8 bytes, como mostra a Figura 4.4.

O endereço base do segmento é composto por 32 bits, que estão espalhados em diferentes campos *base* na entrada da tabela. De maneira análoga, o tamanho do segmento é composto por 20 bits, também fragmentado em diferentes campos denominados *limit*. O campo *flag* contém as seguintes *flags*:

- **Gr:** Granularity. Caso seja 0, o campo *limit* é medido em unidades de 1 byte

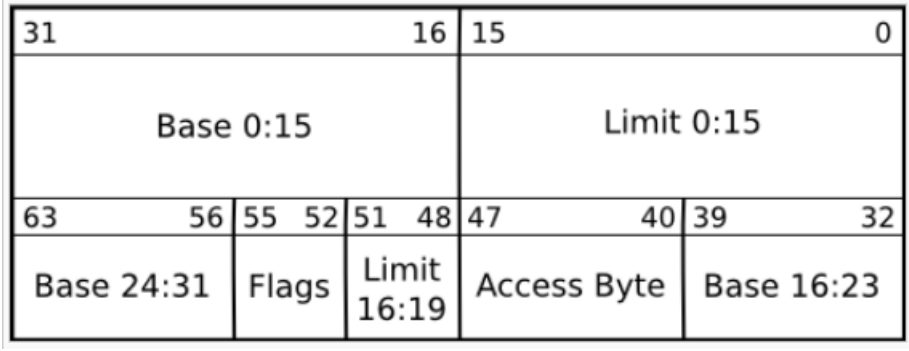


Figura 4.4 – Arquitetura da entrada da GDT (OSDEV, 2019d).

(*block granularity*). Caso seja 1, o campo *limit* é medido em unidades de 4kB (*page granularity*);

- **Sz:** Size. Caso seja 0, indica 16 bit *real mode*. Caso seja 1, indica 32 bit *protected mode*.

O campo *Access Byte* contém as seguintes *flags*:

- **Pr:** Present. Caso seja 1, esse descritor está relacionado a um segmento válido;
- **Privl:** Privilege. Indica o anel de privilégio do processador que o segmento deve executar;
- **S:** Descriptor type. Deve ser 1 para segmentos de código e dados. Deve ser 0 para segmentos do sistema;
- **Ex:** Executable. Deve ser um para segmento de código. Deve ser 0 para segmento de dados;
- **DC:** Direction/Conforming. Caso seja um segmento de código, esse bit é do tipo conforming. Caso 1, códigos executando em um nível de privilégio igual ou menor podem acessá-lo, porem, sem modificar seu nível de execução. Caso seja 0, somente códigos com o mesmo nível de privilégio podem acessá-lo. Caso seja um segmento de dados, esse bit é do tipo Direction. Caso seja 0, o segmento cresce a medida que os endereços de memória crescem. Caso seja 1, o segmento cresce a medida que os endereços de memória diminuem.
- **RW:** Readable/Writable. Caso seja um segmento de código, esse bit é do tipo Readable. Caso 1, a operação de leitura é permitida. Caso 0, a operação de leitura é proibida. A operação de escrita é sempre permitida. Caso seja um segmento de dados, esse bit é do tipo Writable. Caso 1, a operação de escrita é permitida. Caso 0, a operação de escrita é proibida. A operação de leitura é sempre permitida;

- **AC:** Access. Definida como 0. O processador modificará esse bit para 1 quando o segmento é acessado.

Em Intel (2019), é apresentado um modelo de disposição de segmentos denominado *flat model*. Este é o modelo mais simples de organização dos segmentos, que consiste em dois segmentos sobrepostos, que abrangem toda a memória, e um segmento nulo. A linguagem Assembly foi utilizada para definir os bytes que devem formar os três segmentos, além de um descritor da tabela contendo o seu tamanho e o endereço de início.

O preenchimento da GDT foi realizado da seguinte maneira:

- **Base:** Todos os campos foram preenchidos com o valor 0x0, indicando o início da memória;
- **Limit:** Definido como 0xfffff. Como a granularidade foi definida como *page granularity*, o tamanho da memória coberta por esses segmentos possui 4GB;
- **Flag present:** Definido como 1, indicando um segmento válido;
- **Privilege:** Definido como 0, indicando o maior nível de privilégio;
- **Type:** Definido como 1, indicando que o segmento é de dados ou de código;
- **Direction/Conforming:** Definido como 0, indicando que o segmento de dados cresce a medida que os endereços de memória crescem, e indicando que o código presente no segmento só pode ser executado no anel de privilégio definido no bit *privilege*;
- **Readale/Writable:** Definido como 1, indicando que o segmento de código pode ser lido, e que o segmento de dados pode ser escrito;
- **Access:** Definido como 0. Esse bit é alterado pelo processador;
- **Granularity:** Definido como 1, indicando blocos de 4096 bytes;
- **Size:** Definido como 1, indicando modo de operação de 32 bits.

O arquivo *boot/gdt_struct*, contendo o Código 4.7 implementa o esquema apresentado acima. Com a GDT definida, é necessário carregá-la, para que o processador saiba em que local da memória ela está. A instrução lgdt tem como operando o descritor da tabela GDT, de forma a carrega-lo em um registrador específico, denominado GDTR.

Código 4.7 – GDT utilizando *flat model*

```
1 gdt_start:
2 gdt_null:
```

```
3   dd 0x0
4   dd 0x0
5 gdt_code:
6   dw 0xffff      ; limit(0-15)
7   dw 0x0         ; base (0-15)
8   db 0x0         ; base (16-23)
9   db 10011010b   ; set 8 flags
10  db 11001111b   ; limit (16-19) and set 4 flags
11  db 0x0         ; base (24-31)
12 gdt_data:
13  dw 0xffff      ; limit(0-15)
14  dw 0x0         ; base (0-15)
15  db 0x0         ; base (16-23)
16  db 10010010b   ; set 8 flags
17  db 11001111b   ; limit (16-19) and set 4 flags
18  db 0x0         ; base (24-31)
19 gdt_end:
20 gdt_descriptor:
21  dw gdt_end - gdt_start - 1 ; gdt size
22  dd gdt_start               ; gdt address
23 CODE_SEG equ gdt_code - gdt_start
24 DATA_SEG equ gdt_data - gdt_start
```

O próximo passo é modificar o registrador CR0, para que seu primeiro bit seja ativado. Como o registrador CR0 não pode ser alterado diretamente, seu valor deve ser copiado para um registrador de propósito geral, submetido à uma operação OR com valor 0x1 (para preservar os valores dos outros bits), e copiado novamente para o registrador CR0.

Um *far jump* implica na execução de uma instrução jmp para um segmento diferente do que executado no momento da chamada da instrução. Quando um *far jump* é executado, instruções posteriores podem até ser carregadas na memória, contudo, não são executadas até que o *far jump* seja executado completamente. Para executar um *far jump*, é necessário a especificação do segmento destino e do deslocamento dentro do segmento.

O arquivo *boot/switch_to_protected_mode*, contendo o Código 4.8, é responsável por desabilitar as interrupções, carregar a GDT, realizar o *far jump* e executar as primeiras operações no *Protected Mode*. Como o *flat model* está sendo utilizado, todos os registradores de segmento foram inicializados com o descritor do segmento de dados. Logo após, a pilha foi realocada para um novo endereço (0x90000), além de ser limpa. É ideal que exista um segmento específico para a pilha, pois seu modo de crescimento se difere dos outros segmentos. Contudo, essa operação não se faz necessária no momento, além de se desviar da abordagem *flat model*. A diretiva bits indica ao Assembler o modo de operação que o código abaixo dela deverá executar, visto que parte do código deste arquivo executa em *Real Mode* e outra em *Protected Mode*.

Código 4.8 – Habilitação do *Protected Mode*

```
[bits 16]
%include "boot/gdt_struct.asm"
switch_to_pm:
  cli
  lgdt [gdt_descriptor]
  mov eax, cr0
  or eax, 0x1
  mov cr0, eax
  jmp CODE_SEG:init_pm
[bits 32]
init_pm:
  mov ax, DATA_SEG
  mov ds, ax
  mov ss, ax
  mov es, ax
  mov fs, ax
  mov gs, ax
  mov ebp, 0x90000
  mov esp, ebp
  call begin_pm
```

O vídeo no *Protected Mode* pode ser acessado a partir de um mapeamento de memória feito pelo VGA. Desta maneira, uma matriz de caracteres 25x80 é mapeada a partir do endereço 0xB8000. Cada dois bytes representam um caractere da matriz: o primeiro byte define o caractere impresso segundo a tabela ASCII; o seguindo byte define a cor do caractere e sua cor de fundo. Desta maneira, o arquivo *boot/print_string_pm.asm*, contendo o Código 4.9 é responsável por imprimir strings no novo modo. O endereço da *string* a ser impressa deve ser armazenado no registrador EBX. Este código não utiliza a posição atual do cursor, de maneira que a *string* sempre é impressa no início da matriz de caracteres.

Código 4.9 – Impressão de *string* utilizando mapeamento de memória

```
[bits 32]
VIDEO_MEMORY equ 0xb8000
WHITE_ON_BLACK equ 0x0f
print_string_pm:
  pusha
  mov edx, VIDEO_MEMORY
  print_string_pm_loop:
    mov al, [ebx]
    mov ah, WHITE_ON_BLACK
    cmp al, 0
    je print_string_pm_end
    mov [edx], ax
```

```
13      add ebx, 1
14      add edx, 2
15      jmp print_string_pm_loop
16    print_string_pm_end:
17      popa
18      ret
```

Para executar o novo setor de *boot*, as seguintes alterações foram feitas no arquivo *boot/boot_sector.asm*: impressão inicial no modo *Real Mode*; inserção das diretivas %include dos novos arquivos; chamada da função que troca o modo de operação do processador; execução de uma impressão no modo *Protected mode*. O código do novo arquivo é apresentado no Código 4.10. A compilação e a execução permanecem as mesmas, e o resultado é apresentado na Figura 4.5.

Código 4.10 – Quarta versão do setor de *boot*

```
1  [org 0x7c00]
2  mov bp, 0x9000
3  mov sp, bp
4  mov [BOOT_DRIVE], dl
5  mov bx, REAL_MODE_MSG
6  call print_string_rm
7  call switch_to_pm
8  jmp $
9  %include "boot/print_string_rm.asm"
10 %include "boot/print_string_pm.asm"
11 %include "boot/print_hex.asm"
12 %include "boot/switch_to_protected_mode.asm"
13 [bits 32]
14 begin_pm:
15   mov ebx, PROT_MODE_MSG
16   call print_string_pm
17   jmp $
18 BOOT_DRIVE db 0
19 REAL_MODE_MSG db "Started in 16-bit Real Mode", 0
20 PROT_MODE_MSG db "Successfuly landed in 32-bit Protected Mode", 0
21 DISK_ERROR_MSG_1 db "Disk load error in jc", 0
22 DISK_ERROR_MSG_2 db "Disk load error in cmp", 0
23 times 510-($-$$) db 0
24 dw 0xaa55
```

4.2.5 Execução do núcleo

Inicialmente, o núcleo fará somente uma modificação na memória mapeada do VGA escrevendo o caractere 'X' na primeira posição da matriz, indicando que o carregamento foi executado com sucesso e a sua execução está sendo feita da maneira correta. O

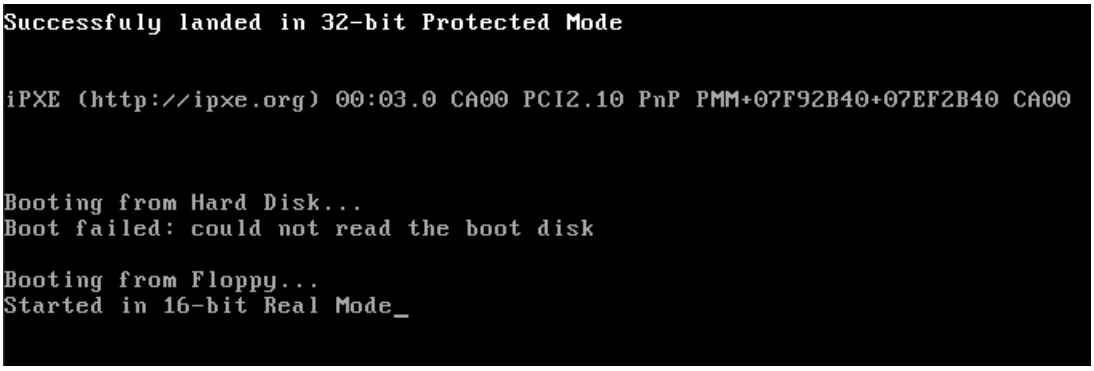


Figura 4.5 – Teste da quarta versão do setor de *boot*

Código 4.11 apresenta a primeira versão da função principal núcleo, que foi escrito no arquivo *kcore/kernel_main.c*.

Código 4.11 – Primeira versão da função principal do núcleo

```
1 #include <stdint.h>
2 void entry ()
3 {
4   uint8_t* video_memory = (uint8_t*) 0xb8000;
5   *video_memory = 'X';
6 }
```

O arquivo incluído neste código é um dos poucos arquivos de cabeçalhos disponíveis no ambiente *Freestanding*. Este arquivo contém diversos *defines* relacionados a tipos de dados e constantes que são utilizadas com frequência, além de incluir outros arquivos de cabeçalho relacionados ao tratamento de tamanho de dados. O tipo uint8_t por exemplo é definido em um dos arquivos incluídos pelo *stdint.h*, o *bits/types.h*. A utilização dos tipos definidos por essa biblioteca são preferíveis pois não existe qualquer semântica relacionadas a eles, a não ser a quantidades de bits que cada variável conterá.

Para que este código seja carregado, é necessário que ele esteja anexado ao setor de *boot* no dispositivo de armazenamento, para que seja possível a sua leitura através da função de acesso à este dispositivo implementada anteriormente. Para isso, o código presente no arquivo *boot/boot_sector.asm* deve ser alterado, como mostra o Código 4.12.

Código 4.12 – Quinta versão do setor de *boot*

```
1 [org 0x7c00]
2 KERNEL_OFFSET equ 0x1000
3 mov [BOOT_DRIVE], dl
4 mov bp, 0x9000
5 mov sp, bp
6 mov bx, REAL_MODE_MSG
7 call print_string_rm
8 call load_kernel
```

```
9  call switch_to_pm
10 jmp $
11 %include "boot/print_string_rm.asm"
12 %include "boot/print_string_pm.asm"
13 %include "boot/print_hex.asm"
14 %include "boot/switch_to_protected_mode.asm"
15 %include "boot/disk_load.asm"
16 [bits 16]
17 load_kernel:
18   mov bx, KERNEL_LOAD_MSG
19   call print_string_rm
20   mov bx, 0
21   mov es, bx
22   mov bx, KERNEL_OFFSET
23   mov dh,48
24   mov dl, [BOOT_DRIVE]
25   call disk_load
26   ret
27 [bits 32]
28 begin_pm:
29   mov ebx, PROT_MODE_MSG
30   call print_string_pm
31   call KERNEL_OFFSET
32   jmp $
33 BOOT_DRIVE db 0
34 REAL_MODE_MSG db "Started in 16-bit Real Mode", 0
35 PROT_MODE_MSG db "Successfuly landed in 32-bit Protected Mode", 0
36 KERNEL_LOAD_MSG db "Loading kernel into memory", 0
37 DISK_ERROR_MSG_1 db "Disk load error in jc", 0
38 DISK_ERROR_MSG_2 db "Disk load error in cmp", 0
39 times 510-($-$$) db 0
40 dw 0xaa55
```

Este código, logo após a primeira impressão, chama a função load_kernel, que é responsável por carregar o conteúdo de 48 setores do dispositivo de armazenamento que foi utilizado para obter o setor de *boot* para o endereço de memória 0x1000. Logo após, a função que troca o modo de operação do processador é chamada, seguida da segunda impressão. A chamada para o núcleo é realizada através da execução da instrução call, passando como parâmetro o endereço 0x1000. Como a instrução call está referenciando um endereço específico, é preciso que o código de entrada para o núcleo esteja exatamente neste endereço. Para isso, durante o processo de *linking*, se faz necessário especificar o local que o código deve estar. Como o *linker* organiza as entradas por arquivos, é preferível definir um arquivo que possui somente a função de entrada que chama a função principal do núcleo, de modo a tornar o arquivo *kcore/kernel_main.c* maleável. O arquivo *kcore/kernel_entry.asm*, que contém o Código 4.13, é responsável por realizar este processo.

Código 4.13 – Entrada para o núcleo

```
1 global _start
2 [bits 32]
3 _start:
4 [extern entry]
5 call entry
6 jmp $
```

Tendo em vista que mudanças no processo de compilação do código se fazem necessárias, este é o local ideal para o início da utilização do GNU Make. O Código 4.14 apresenta primeira versão do *Makefile*. Inicialmente, macros são criadas para auxiliar a escrita do script, de modo a definir o local onde existem códigos C, locais onde existirão códigos objetos após o processo de compilação, o compilador C, *flags* utilizadas para a compilação e o *debugger*. O código subsequente é composto por regras que indicam a dependência destas, assim como suas ações. O arquivo é analisado em sequência, de forma que as regras mais específicas devem ser declaradas antes das regras mais gerais.

Os códigos C são compilados por meio da regra possui como dependência códigos C e geram arquivos ELF. Esse requerimento é expresso a partir da diretiva %.o: %.c, e a ação é executada a partir da diretiva $CC $CFLAGS $< -o $@. O símbolo **$<** representa o primeiro arquivo de dependência que casa com o padrão **%.c** da regra, assim como o símbolo **$@** representa o elemento gerado, que casa com o padrão expresso pela regra **%.o**. O símbolo **$^** é substituído por todos os arquivos que casam com o padrão definido pelos arquivos de dependência. A montagem de arquivos Assembly segue a mesma lógica.

O arquivo *kcore/kernel_main.bin* é responsável por conter o código do núcleo em formato binário. Para montá-lo, se faz necessária a presença de todos os arquivos ELF que compõem o núcleo, além do arquivo especial que executará o código de entrada. A ação realizada utiliza o *linker*, que recebe como parâmetro os arquivos ELF. Com a utilização da *flag* –oformat binary, o arquivo gerado após a sua execução é um arquivo binário puro, sem cabeçalhos ELF. Como o *linker* respeita a ordem dos parâmetros, o primeiro arquivo a ser alocado será o ELF *kcore/kernel_entry.o* no endereço 0x1000, definido a partir da *flag* -Ttext 0x1000. Essa organização é necessária para que o carregamento do código seja realizado com sucesso pelo setor de *boot*.

Código 4.14 – Primeira versão do *Makefile*

```
1 C_SOURCES = $(wildcard kcore/*.c)
2 OBJ = ${C_SOURCES:.c=.o}
3 CC = i686-elf-gcc
4 CFLAGS= -g -ffreestanding -c -nostdlib -lgcc
5 GDB = i686-elf-gdb
6 run: os-image.bin
7   qemu-system-i386 -fda os-image.bin
8 build: os-image.bin
```

```
9  os-image.bin: boot/boot_sector.bin kcore/kernel_main.bin
10   cat $^ > os-image.bin
11 kcore/kernel_main.bin: kcore/kernel_entry.o ${OBJ}
12   i686-elf-ld -z muldefs -o $@ -Ttext 0x1000 $^ --oformat binary
13 %.o: %.c
14   ${CC} ${CFLAGS} $< -o $@
15 %.bin: %.asm
16   nasm $< -f bin -o $@
17 %.o: %.asm
18   nasm $< -f elf -o $@
19 clean:
20   rm -rf *.bin *.dat boot/*.bin kcore/*.bin
21   rm -rf kcore/*.o
```

A execução do comando make run resulta na execução do script contido no arquivo *Makefile* que executando o sistema, como mostra a Figura 4.6.

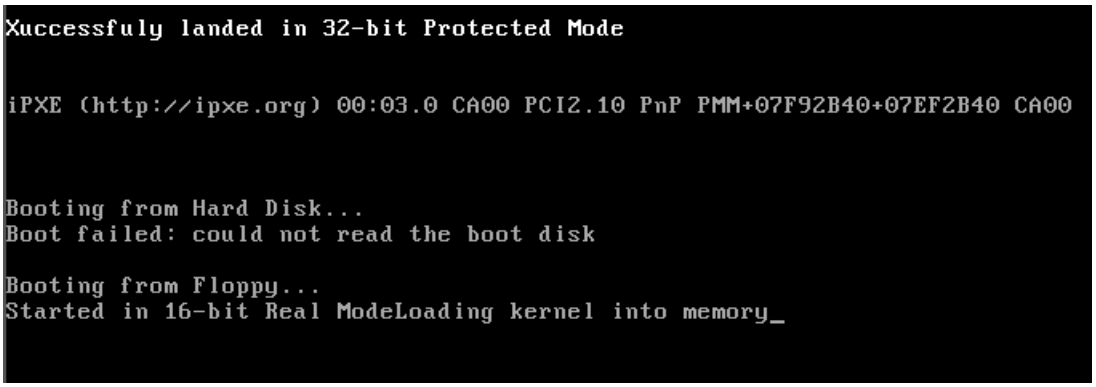


Figura 4.6 – Teste da quinta versão do setor de *boot*

Uma alternativa para implementaçao deste módulo é a utilização de um *boot loeader*, como o GNU (2012);

4.3 Módulo 1: Implementação de uma biblioteca C

Uma biblioteca C é um conjunto de funções presentes no sistema operacional, que tem como objetivo disponibilizar macros, definições de tipos, funções para tarefas como tratamento de strings, processamento de entrada/saída, gerenciamento de memória, computação matemática e outros serviços do sistema (WIKIPEDIA, 2020c). Em ambientes baseados no UNIX, é comum que as bibliotecas sejam disponibilizadas de maneira dinâmica, sendo carregadas na memória e ligadas ao processo em tempo de execução com o auxílio de um *dinamic linker* (WIKIPEDIA, 2020e). A American National Standards Institute (ANSI) adota um padrão de comportamento das funções presentes nas bibliotecas C, chamado ISO C library (International Organization for Standardization, 2018). Esse padrão permite a

existência de diferentes implementações que são compatíveis entre si. Alguns compiladores possuem em seu código implementações de bibliotecas C que seguem este padrão, de modo que o programa pode chamar essas funções, sem a necessidade de chamar uma biblioteca compartilhada dinamicamente. Tendo em vista que o objetivo deste trabalho é puramente didático e ainda não existe um *linker* implementado, não foi utilizado nenhum tipo de padronização externa. Um conjunto restrito de funções que representam a interface deste módulo foram implementadas, sendo elas: kmalloc, kmalloc_u, kfree, itoa, reverse, strlen, memcpy, memmov, memcmp, print_stack_asm.

A organização do trabalho passará a ter dois novos diretórios: *include* e *libc*. O diretório *include* é responsável por conter todos os arquivos de cabeçalho deste projeto e será referenciado pela *flag* de compilação -I include, encarregada de adicionar este diretório à lista de diretórios que contém arquivos de cabeçalhos acessada pelo GCC. O diretório *libc* conterá os arquivos que implementam as funções que formam a biblioteca C. Essas alterações implicam nas seguintes mudanças do *Makefile*: adição do diretório *libc* na lista de diretórios que possuem códigos fonte C; adição de um *wildcard* que referencia os arquivos de cabeçalho; como a função print_stack_asm utiliza código Assembly, é preciso indicar que o arquivo que contém este código deve ser compilado para o formato ELF sem que passe pelo *linker*, adicionando-o à lista OBJ; adição da *flag* -I include para a compilação de códigos C. O Código 4.15 apresenta o código da nova versão do *Makefile*.

Código 4.15 – Segundaa versão do *Makefile*

```
C_SOURCES = $(wildcard kcore/*.c libc/*.c)
HEADERS = $(wildcard include/*.h)
OBJ = ${C_SOURCES:.c=.o libc/print_stack_asm.o}
CC = i686-elf-gcc
CFLAGS= -g -ffreestanding -c -I include -nostdlib -lgcc
GDB = i686-elf-gdb
run: os-image.bin
  qemu-system-i386 -fda os-image.bin
build: os-image.bin
os-image.bin: boot/boot_sector.bin kcore/kernel_main.bin
  cat $^ > os-image.bin
kcore/kernel_main.bin: kcore/kernel_entry.o ${OBJ}
  i686-elf-ld -z muldefs -o $@ -Ttext 0x1000 $^ --oformat binary
%.o: %.c ${HEADERS}
  ${CC} ${CFLAGS} $< -o $@
%.bin: %.asm
  nasm $< -f bin -o $@
%.o: %.asm
  nasm $< -f elf -o $@
clean:
  rm -rf *.bin *.dat boot/*.bin kcore/*.bin
  rm -rf kcore/*.o libc/*.o
```

O espaço de memória compreendido entre os endereços 0x100000 e 0x10C800 foram reservados para conter a *heap* do sistema operacional, conforme presentado na seção 4.9. Este espaço é mantido sob a responsabilidade do kernel, mas possui uma interface para acesso externo, composta pelas funções da libc *kmalloc*, *kmalloc_u* e *kfree*. As funções *kmalloc* e *kmalloc_u* são responsáveis por alocar espaços de memória na *heap*, e retornam o início do segmento de memória alocado. A função *kfree* é responsável por desalocar um espaço de memória de tamanho especificado pelo parâmetro, de modo a permitir novas alocações neste espaço. Essas funções são detalhadas na seção 4.9.

A função itoa (*int to ascii*) recebe um valor numérico de 32 bits e o transforma em *string*, que será apontada pelo vetor passado por parâmetro. O algoritmo, inicialmente, verifica o sinal do número recebido por parâmetro e utiliza o valor oposto, caso este seja negativo. Como a *string* representa o número no formato decimal, este é analisado a cada 10 unidades (começando pela unidade menos significativa), e o valor ASCII deste algarismo é adicionado à *string*. Ao final da tradução, o caractere '-' é adicionado caso o número seja negativo. O último passo é a execução da função reverse, responsável por inverter a *string* que recebe como parâmetro. Esta função utiliza duas variáveis para referenciar o primeiro e o último caractere da *string*, que são trocados com o auxílio de uma terceira variável. A variável que referencia o primeiro caractere é incrementada, enquanto que a varável que referencia o último caractere é decrementada e o processo se repete enquanto a segunda variável for maior que a primeira. A função strlen é responsável por retornar o tamanho da *string* que recebe como parâmetro. Um contador é incrementado a cada iteração pela *string* até que o caractere de valor 0x0, que indica o final da *string*, seja encontrado.

As funções descritas acima foram implementadas no arquivo *libc/string.c*, como mostra o Código 4.16. As implementações destas funções foram baseadas em Kernighan e Ritchie (1988). O arquivo de cabeçalho utilizado para este código contém somente a inclusão do arquivo *stdint.h* e os cabeçalhos das funções apresentadas.

Código 4.16 – Tratamento de *strings* da biblioteca C

```
1  #include <string.h>
2  void itoa (uint32_t n, char s[])
3  {
4    uint32_t i, sign;
5    if ((sign = n) < 0)
6      n = -n;
7    i = 0;
8    do
9    {
10     s[i++] = n % 10 + '0';
11   } while ((n /= 10) > 0);
12
13   if (sign < 0)
```

```
14      s[i++] = '-';
15    s[i] = '\0';
16
17    reverse(s);
18  }
19  void reverse(char s[])
20  {
21    uint32_t c, i, j;
22    for (i = 0, j = strlen(s)-1; i < j; i++, j--)
23    {
24      c = s[i];
25      s[i] = s[j];
26      s[j] = c;
27    }
28  }
29  uint32_t strlen (char s[])
30  {
31    uint32_t i = 0;
32    while (s[i] != '\0')
33      ++i;
34    return i;
35  }
```

Quatro funções de manipulação geral de memória foram criadas a fim de facilitar a realização de novas atividades: *memcpy*, *memmov* e *memcmp*. A função *memcpy* é responsável por copiar bytes de um espaço de memória para outro espaço de memória. Essa função recebe três parâmetros que indicam o endereço inicial que contém o bytes a serem copiados, a quantidade de bytes a serem copiados, e o endereço inicial para onde os bytes serão escritos. O objetivo desta função é similar ao da função *memmov*, mas estas diferem na implementação. A função *memcpy* simplesmente itera pela memória, a partir do endereço inicial, copiando byte a byte para o endereço destino. A função *memmov* possui algumas verificações que a permite realizar cópias de dados em endereço se sobrepõem: caso o endereço inicial de origem seja igual ao endereço de memória inicial de destino, a função simplesmente retorna; caso o endereço inicial de origem seja o menor que o endereço inicial de destino, a cópia é feita do maior endereço para o menor endereço; caso o endereço inicial de origem seja maior que o endereço inicial de destino, a iteração acontece da mesma forma que na função *memcpy*. Desta forma, a função *memmov* não sobrescreve os valores que ainda não foram copiados do espaço de memória de origem. A função *memcmp* recebe dois endereços de memória e um valor que indica a quantidade de bytes que serão analisados pela função. Esta função retorna -1 caso o valor apontado pelo primeiro argumento for menor que o valor apontado pelo segundo, e retorna 1 caso o oposto aconteça. O retorno será 0 caso os valores apontados pelos parâmetros sejam iguais.

Estas funções foram implementadas no arquivo *libc/mem.c* como mostra o Có-

digo 4.17. O arquivo de cabeçalho utilizado para este código contém somente a inclusão do arquivo *stdint.h* e os cabeçalhos das funções apresentadas.

Código 4.17 – Manipulação de memória da biblioteca C

```
#include <mem.h>
void memcpy (uint8_t *source, uint8_t *dest, uint32_t nbytes)
{
  int i;
  for (i = 0; i < nbytes; i++)
    *(dest + i) = *(source + i);
}
void memset (uint8_t *dest, uint8_t val, uint32_t len)
{
  for (;len != 0; len--)
    *dest++ = val;
}
void memmov (uint8_t *source, uint8_t *dest, uint32_t len)
{
  if (source == dest)
    return;
  if (source < dest)
  {
    source += len;
    dest += len;
    uint32_t i;
    for (i = 0; i < len; i++)
      *(--dest) = *(--source);
  }
  else
  {
    uint32_t i;
    for (i = 0; i < len; i++)
      *(dest++) = *(source++);
  }
}
int8_t memcmp (uint8_t *source, uint8_t *targ, uint32_t len)
{
  uint32_t i;
  for (i = 0; i < len; i++)
  {
    if (*source < *targ)
      return -1;
    else if (*(source++) > *(targ++))
      return 1;
  }
  return 0;
}
```

```
44 void print_stack_c (uint32_t address, uint32_t value)
45 {
46   kprintf ("stack->%x: %x\n", 2,address, value);
47 }
```

A função print_stack_asm é responsável por imprimir valores presentes na pilha. Como é necessário o acesso à registradores, parte deste código foi escrita em Assembly no arquivo *libc/print_stack_asm.asm*, como mostra o Código 4.18. O processador i386 utiliza a convenção de chamadas de funções que é definida na System V ABI (Santa Cruz Operation, 1997), de forma que os parâmetros são empurrados para pilha na sequência inversa que são representadas nas funções C. Caso uma chamada de função no formato func (a, b, c) seja feita, a execução da função func será realizada com os seguintes valores na pilha, a partir do topo: endereço de retorno; c; b; a.

A partir de testes realizados, percebeu-se que o compilador GCC adiciona alguns preenchimentos na pilha dependendo da quantidade de parâmetros passados, alinhando-os à 16 bytes. O Quadro 4.1 demonstra o comportamento da pilha durante a chamada de funções.

Quadro 4.1 – Comportamento da pilha durante a chamada de funções

Código chamador	**Código chamado**	**Número de parâmetros de 4 bytes cada**	**Bytes adicionados à pilha**
C	Assembly	0	4
C	Assembly	De 1 a 4	4 + 16
C	Assembly	De 5 a 8	4 + 32
C	C	0	16
C	C	De 1 a 4	16 + 16
C	C	De 5 a 8	16 + 32
Assembly	Assembly	0	4
Assembly	Assembly	N	4 + (N*4)
Assembly	C	0	16
Assembly	C	N	16 + (N*4)

O objetivo da função print_stack_asm é imprimir valores na pilha como se não existisse uma chamada de função, portanto, a função deve saber se o código chamador é um código C ou um código Assembly, para que os bytes adicionados à pilha sejam ignorados. Por esse motivo, além da quantidade de bytes a serem impressos, a função espera um segundo parâmetro que indica em que linguagem está o código chamador: caso seja um código C, o valor do segundo parâmetro é 0x1C; caso seja um código Assembly, o segundo

parâmetro é 0x14. Esses valores indicam a quantidade de bytes que devem ser ignorados pela função print_stack_asm. Para cada byte presente na pilha, esta função chama a função print_stack_c, presente no arquivo *libc/mem.c* e apresentado no Código 4.17, que é responsável por imprimir este valor utilizando uma função de impressão que será descrita na seção 4.5. As funções print_stack_asm e print_stack_c foram adicionadas à este trabalho após a escrita de diversos módulos, por isso utiliza uma função de um módulo apresentado futuramente. Estas funções estão descritas neste ponto somente por uma questão organização do texto.

O teste deste módulo pode ser realizado a partir de operações com valores na memória dentro do espaço mapeado no VGA, permitindo a visualização. Para demonstrar uma possibilidade de teste, a função memcpy foi utilizada, como mostra o Código 4.19. Como cada caractere ocupa dois bytes, este código copia 4 caracteres 'X' para quatro espaços à frente, como mostra Figura 4.7.

Código 4.18 – Impressão de valores da pilha

```
1  global print_stack_asm
2  extern print_stack_c
3  print_stack_asm:
4  mov edx, [esp+8]
5  mov eax, [esp+4]
6  push eax
7  push dword 0
8  print_stack_loop:
9  mov edx, [esp+0x10]
10 pop ecx
11 pop eax
12 cmp ecx, eax
13 je print_stack_end_loop
14 push eax
15 push ecx
16 add ecx, edx
17 push dword[esp+ecx]
18 add ecx, esp
19 add ecx, 4
20 push ecx
21 call print_stack_c
22 add esp, 8
23 pop ecx
24 add ecx, 1
25 push ecx
26 jmp print_stack_loop
27 print_stack_end_loop:
28 ret
```

Código 4.19 – Função principal do núcleo para o teste da função memcpy

```c
#include <stdint.h>
#include <mem.h>

void entry ()
{
  uint8_t* video_memory = (uint8_t*) 0xb8000;
  *video_memory = 'X';
  *(video_memory+2) = 'X';
  *(video_memory+4) = 'X';
  *(video_memory+6) = 'X';
  memcpy (0xb8000, 0xb8010, 8);
}
```

```
XXXXessfXXXXlanded in 32-bit Protected Mode


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F915F0+07EF15F0 CA00



Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
Started in 16-bit Real ModeLoading kernel into memory_
```

Figura 4.7 – Teste da biblioteca C

4.4 Módulo 2: Implementação de um mecanismo de comunicação com dispositivos de entrada/saída

Para que o processador possa se comunicar com os dispositivos de estrada/saída, os controladores destes disponibilizam portas de acesso, que são registradores presentes no controlador do dispositivo, aos quais o processador tem acesso. Para que uma porta possa ser acessada, ela precisa possuir um endereço. O endereçamento das portas pode ocorrer de duas maneiras distintas: através do *memory-mapped I/O*; ou através do *port-mapped I/O*. Existe uma terceira abordagem denominada *I/O channels*, que utiliza um processador independente para o dispositivo de entrada/saída, como é o caso das placas gráficas, mas não será abordada neste trabalho. O método *memory-mapped I/O* consiste na restrição de uma parte não usada da memória física para o mapeamento das portas de um dispositivo. Desta maneira, o processador não tem conhecimento que está escrevendo uma porta. O método *port-mapped I/O* mapeia as portas em um espaço de endereçamento diferente que o da memória física. Esse espaço de endereçamento pode ser acessado a partir da utilização de um pino extra na interface física da CPU, normalmente conectado ao barramento de controle, ou pela utilização de um barramento dedicado. Ao gerar um

endereço, o processador deve indicar se o endereço gerado é referente à memória ou ao espaço de endereçamento de entrada/saída. Portanto, o acesso ao espaço de endereçamento de entrada/saída é feito a partir da utilização de instruções especiais, como in e out.

Considerando que o acesso às portas através do método *port-mapped I/O* só pode ser feito a partir de instruções especiais em Assembly, e o núcleo deste projeto foi escrito em C, a utilização do *Inline Assembly* se mostrou útil para otimizar a quantidade de arquivos do projeto. O GCC pode lidar com instruções em Assembly em meio ao código C a partir da utilização da instrução asm. Esta instrução recebe um modelo da instrução Assembly requerida como *string*, operandos de saída, operandos de entrada, e uma lista de registradores utilizados, como mostra o exemplo a seguir:

```
asm ( assembler template
    : output operands                   (optional)
    : input operands                    (optional)
    : clobbered registers list          (optional)
    );
```

O modelo da instrução, neste trabalho, foi escrito utilizando a sintaxe AT&T. Os operandos de entrada e saída são utilizados para que valores presentes em variáveis do código C possam ser utilizados pelo Assembly. Os operandos de saída representam a forma com que o compilador e o Assembler tratarão as variáveis em C para que estas armazenem algum valor de saída do código Assembly. De maneira análoga, os operandos de entrada representam a forma de tratamento de valores presentes em variáveis C, que atuarão como parâmetros para o código Assembly. Um operando é composto por dois componentes: uma restrição no formato *string*, contendo alguma característica que o GCC deve se atentar ao realizar a operação e o registrador utilizado no mapeamento; e a variável utilizada no mapeamento. A lista de registradores presente na ultima sessão do comando asm informa ao compilador sobre os registradores que serão utilizados. O compilador, então, armazena o valor desses registradores antes de realizar a chamada do código Assembly.

O arquivo *include/ports.h* apresenta, além da inclusão do arquivo *stdint.h*, os cabeçalhos das funções que formam a interface deste módulo, responsáveis por acessar as portas de entrada/saída: port_byte_in; port_byte_out; port_word_in; port_word_out; port_dword_in; port_dword_out. As funções com sufixo *in* indicam que a função irá ler dados de uma porta, enquanto que as funções com sufixo *out* escrevem dados em uma porta. As funções apresentam a quantidade de bytes lidos ou escritos no nome da função (*byte* para 8 bits, *word* para 16 bits, *dword* para 32 bits), e o código Assembly utiliza registradores com tamanhos apropriados para realizar a leitura ou escrita. As funções de leitura recebem como parâmetro o endereço da porta de leitura, enquanto as funções de escrita recebem o endereço da porta e o dado a ser escrito. Um novo diretório denominado

drivers foi criado para conter os arquivos referentes ao tratamento de dispositivos. O arquivo *drivers/ports.c*, contendo o Código 4.20, contém as funções descritas acima.

Código 4.20 – Utilização de portas através do método *port-mapped I/O*

```
#include <ports.h>
uint8_t port_byte_in (uint16_t port)
{
  uint8_t result;
  asm volatile ("in %%dx, %%al" : "=a" (result) : "d" (port));
  return result;
}
void port_byte_out (uint16_t port, uint8_t data)
{
  asm volatile ("out %%al, %%dx" : : "a" (data) , "d" (port));
}
uint16_t port_word_in (uint16_t port)
{
  uint16_t result;
  asm volatile ("in %%dx, %%ax" : "=a" (result) : "d" (port));
  return result;
}
void port_word_out (uint16_t port, uint16_t data)
{
  asm volatile ("out %%ax, %%dx" : : "a" (data), "d" (port));
}
uint32_t port_dword_in (uint16_t port)
{
  uint32_t result;
  asm volatile ("in %%dx, %%eax" : "=a" (result) : "d" (port));
  return result;
}
void port_dword_out (uint16_t port, uint32_t data)
{
  asm volatile ("out %%eax, %%dx" : : "a" (data), "d" (port));
}
```

A sintaxe do *Inline Assembly* se difere na sintaxe AT&T somente em um ponto: o nome dos registradores devem ser precedidos por "%%" em vez de "%". Isso ocorre pois o símbolo "%" no *Inline Assembly* é utilizado para referenciar os registradores presentes nos campos de operandos de saída e operandos de entrada. A instrução in espera dois parâmetros: endereço da porta e registrador que receberá o valor lido. O endereço da porta é armazenado no registrador DX através do operando de entrada da instrução asm, da mesma maneira que o valor lido, após ser movido para o registrador EAX (ou algum registrador análogo de tamanho menor), é mapeado na variável *result*, através do operando de saída. O símbolo "=", presente no operando de saída, indica que o GCC não deve

se preocupar sobre o valor inicial que a variável relacionada possui. A letra presente na *string* de restrição indica o registrador: "a" se refere ao registrador "EAX" e "d" se ao registrador "EDX". Essa letra pode referenciar registradores de 16 bits e 8 bits, dependendo do tamanho do operando. A instrução out espera dois parâmetros: o valor a ser escrito na porta, e o endereço da mesma. Ambos os valores são mapeados utilizando operandos de entrada da instrução asm, pois os dois valores se comportarão como parâmetros. A instrução volatile informa ao processador que a instrução atual não deve sofrer qualquer tipo de otimização, fazendo com que o código executado seja exatamente o mesmo.

Com a modificação da estrutura de diretórios do projeto, duas modificações no *Makefile* se fazem necessárias: adição dos arquivos do diretório *drivers* na lista de códigos C e adição dos arquivos ELF gerados pela compilação desses arquivos C na regra responsável por limpar o projeto após uma compilação. Para realizar a execução deste novo módulo, pode-se utilizar a porta de controle e de dados do VGA que também será utilizada na seção 4.5. Ao escrever o valor 14 na porta de endereço 0x3D4, a porta 0x3D5 referencia os bits de alta do deslocamento do cursor dentro da matriz mapeada pelo VGA. De maneira análoga, ao escrever o valor 15 na porta de endereço 0x3D4, a porta 0x3D5 referencia os bits de baixa ordem deste deslocamento. O arquivo principal do núcleo altera a posição do cursor para a primeira posição da matriz do VGA, como mostra o Código 4.21. O resultado pode ser observado na Figura 4.8.

Código 4.21 – Função principal do núcleo para o teste das funções de acesso às portas

```
1  #include <stdint.h>
2  #include <ports.h>
3
4  void entry ()
5  {
6    port_byte_out (0x3d4, 14);
7    port_byte_out (0x3d5, 0);
8    port_byte_out (0x3d4, 15);
9    port_byte_out (0x3d5, 0);
10 }
```

4.5 Módulo 3: Implementação de um *driver* para vídeo

O *driver* de vídeo foi construído com base no mapeamento de memória realizado pelo VGA. Além deste mapeamento, existem algumas portas no espaço de endereçamento de entrada/saída que foram utilizadas para a implementação de funcionalidades como a requisição e definição da posição do cursor. O Código 4.22, escrito no arquivo *include/vga.h*, é responsável por apresentar a interface deste módulo, contendo *defines* utilizados durante o código, e o cabeçalho das funções.

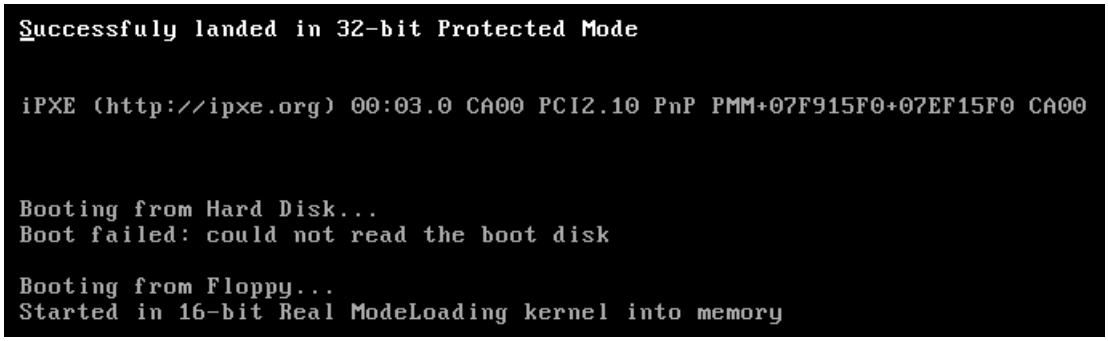


Figura 4.8 – Teste do mecanismo de comunicação com dispositivos de entrada/saída

Código 4.22 – Cabeçalho para o *driver* de vídeo

```
#ifndef SCREEN_H
#define SCREEN_H
#define VIDEO_ADDRESS 0xb8000
#define MAX_ROWS 25
#define MAX_COLS 80
#define WHITE_ON_BLACK 0x0f
#define YELLOW_ON_BLACK 0x0e
#define CYAN            0x3
#define LIGHT_GREEN     0xa
#define WHITE           0xf
#define LIGHT_RED       0xc
#define VGA_CTRL_PORT 0x3d4
#define VGA_DATA_PORT 0x3d5
#include <stdint.h>
#include <stdarg.h>
#include <mem.h>
#include <ports.h>
void clear_screen ();
void kprint_at (uint8_t *message, int32_t offset);
void kprint (uint8_t *message);
void kprint_debug (uint8_t *message, uint8_t color);
void kprintf (char *str, int n, ...);
#endif
```

A função clear_screan é responsável por limpar todos os caracteres da tela e posicionar o cursor no primeiro caractere da matriz. Essa função na realidade imprime o caractere " " com a cor de fundo preta, dando a impressão de que todos os caracteres impressos foram apagados. O posicionamento do cursor foi realizado a partir da função set_cursor_offset definida de maneira estática. Quando a porta de controle do VGA (0x3D4) é escrita com o valor 14, a porta de dados (0x3D5) referencia os bits de alta ordem do deslocamento do cursor. Para que esta porta referencie os bits de baixa ordem, a porta de controle deve ser escrita com o valor 15. A porta de dados pode ser escrita ou

lida, podendo ser utilizada para definir um novo deslocamento do cursor ou descobrir o deslocamento atual.

A função kprint_at tem como principal objetivo imprimir uma string a partir de um determinado deslocamento na matriz. Para realizar esta operação, a função recebe como parâmetro o endereço de início da string e o deslocamento inicial na matriz. Caso o endereço inicial recebido por esta função seja -1 (a função kprint chama a função kprint_at com este parâmetro), a posição atual do cursor é utilizada para imprimir a string. A função kprint_debug funciona como a função kprint, mas recebe uma cor como segundo parâmetro, permitindo um destaque dentre outras strings já impressas. As funções descritas utilizam como base a função estática print_char, que imprime um caractere com determinada cor, em uma posição da matriz mapeada pelo VGA. O caractere "\n" foi utilizado como caractere de quebra de linha, sendo responsável por modificar a localização do cursor para a próxima linha. Caso a linha atual seja a última, a função scroll_screen é chamada, copiando, a partir da segunda linha da matriz, as informações de uma linha para a linha anterior utilizando a função memcpy.

A função kprintf tem como objetivo imprimir, no local atual do cursor, uma string formatada. Esta função recebe a string como primeiro parâmetro, a quantidade de elementos que serão substituídos na string como segundo parâmetro e a lista de parâmetros utilizados para preencher a string. Os parâmetros pertencentes à lista são classificados de três maneiras: inteiros em decimal, especificados pelo símbolo "%d"; inteiros em hexadecimal, especificados pelo símbolo "%x"; string, especificadas pelo símbolo "%s". A função kprintf utiliza a função itoa para escrever inteiros em decimal e um algoritmo similar ao escrito no Código 4.4 para escrever inteiros em hexadecimal. Vale ressaltar que esta função utiliza um mecanismo provido pelo GCC que permite que uma função possua um número de parâmetros variado. Para isto, o cabeçalho da função possui os caracteres "...", indicando que uma lista de parâmetros é esperada. Esta lista é acessada a partir de uma variável local que deve ser criada com o tipo va_list. A variável deve ser inicializada a partir da função va_start que recebe como parâmetro a variável local que deve referenciar a lista e a quantidade de parâmetros esperados para a lista. O acesso a esta é realizado de forma sequencial a partir da função va_arg, que recebe como parâmetros a variável que referencia a lista de argumentos e o tipo que será utilizado para lidar com este argumento. O tipo e as funções utilizadas para este propósito são definidas como macros no arquivo de cabeçalho *stdarg.h*.

O Código 4.23 contém o código do arquivo *drivers/vga.c*, que implementa as funções descritas acima. Para testar este novo módulo, o arquivo principal do núcleo *kcore/kernel_main.c* foi alterado para conter o Código 4.24, e a Figura 4.9 apresenta a execução deste módulo.

Código 4.23 – *Driver* de vídeo

```
#include <vga.h>
static uint32_t print_char (uint8_t character, uint8_t attributeint,
   uint32_t offset);
static uint32_t get_cursor_offset();
static void set_cursor_offset (uint32_t offset);
static void scroll_screen();
uint8_t attribute = YELLOW_ON_BLACK;

static uint32_t print_char (uint8_t character, uint8_t attribute,
   uint32_t offset)
{
  if(character == '\n')
  {
    offset /= 2;
    uint32_t row = offset / MAX_COLS;
    row++;
    return MAX_COLS * row * 2;
  }
  uint8_t *vid_addr = (uint8_t*) VIDEO_ADDRESS;
  *(vid_addr + offset) = character;
  offset++;
  *(vid_addr + offset) = attribute;
  offset++;
  return offset;
}
static uint32_t get_cursor_offset ()
{
  uint32_t offset;
  port_byte_out (VGA_CTRL_PORT,14);
  offset = port_byte_in (VGA_DATA_PORT);
  offset <<= 8;
  port_byte_out (VGA_CTRL_PORT, 15);
  offset += port_byte_in (VGA_DATA_PORT);
  return offset;
}
static void set_cursor_offset (uint32_t offset)
{
  uint8_t offset_byte = (uint8_t) (offset>>8);
  port_byte_out (VGA_CTRL_PORT, 14);
  port_byte_out (VGA_DATA_PORT, offset_byte);
  port_byte_out (VGA_CTRL_PORT, 15);
  offset_byte = (uint8_t) (offset & 0xff);
  port_byte_out (VGA_DATA_PORT, offset_byte);
}
static void scroll_screen ()
{
  uint8_t *vid_addr = (uint8_t *) VIDEO_ADDRESS;
```

```
46   vid_addr = vid_addr + (MAX_COLS * 2);
47   uint32_t i;
48   for(i = 0 ; i < MAX_ROWS; i++)
49   {
50     memcpy (vid_addr, vid_addr - (MAX_COLS * 2) , MAX_COLS*2);
51     vid_addr = vid_addr + (MAX_COLS * 2) ;
52   }
53   set_cursor_offset (MAX_ROWS * MAX_COLS - MAX_COLS);
54 }
55 void kprint_at (uint8_t *message, int32_t offset)
56 {
57   if(offset < 0)
58     offset = get_cursor_offset () * 2;
59   for (;*message != '\0' ; message ++)
60   {
61     offset = print_char(*message, attribute, offset);
62     if (offset > MAX_COLS * MAX_ROWS * 2 - 1)
63     {
64       scroll_screen();
65       offset = get_cursor_offset() * 2;
66     }
67   }
68   set_cursor_offset(offset / 2);
69 }
70 void kprint (uint8_t *message)
71 {
72   kprint_at (message, -1);
73 }
74 void kprint_debug (uint8_t *message, uint8_t color)
75 {
76   attribute = color;
77   kprint_at (message, -1);
78   attribute = YELLOW_ON_BLACK;
79 }
80 void kprintf (char *str, int n, ...)
81 {
82   char buff [strlen (str)];
83   va_list ap;
84   va_start (ap, n);
85   int32_t i, index = 0;
86   for (i = 0; i < strlen (str); i++)
87   {
88     if (str[i] == '%')
89     {
90       if (str[i+1] != '%')
91       {
92         buff[index] = '\0';
```

```
93          kprint (buff);
94          if (str[i+1] == 'd')
95          {
96            char s[10];
97            itoa (va_arg (ap, int), s);
98            kprint (s);
99          }
100         else if (str[i+1] == 'x')
101         {
102           char hex_v[11] = "0x????????\0";
103           char map [16] = "0123456789abcdef";
104           int int_v = va_arg (ap, int);
105           int tmp = int_v;
106           int digit = 2;
107           int shift = 28;
108           for (;digit < 10; digit++)
109           {
110             tmp >>= shift;
111             tmp &= 0xf;
112             hex_v[digit] = map[tmp];
113             shift -= 4;
114             tmp = int_v;
115           }
116           kprint (hex_v);
117         }
118         else if (str[i+1] == 's')
119           kprint (va_arg (ap, char*));
120
121         index = 0;
122         ++i;
123       }
124       else
125       {
126         buff[index] = str[i];
127         ++index;
128         ++i;
129       }
130     }
131     else
132     {
133       buff[index] = str[i];
134       ++index;
135     }
136   }
137   if (index > 0)
138   {
139     buff[index] = '\0';
```

```
140     kprint (buff);
141   }
142 }
143
144 void clear_screen()
145 {
146   uint32_t i;
147   for (i = 0; i < MAX_ROWS * MAX_COLS * 2; i+=2)
148     print_char(' ', WHITE_ON_BLACK, i);
149   set_cursor_offset (0);
150 }
```

Código 4.24 – Função principal do núcleo para teste do *driver* de vídeo

```
1 #include <vga.h>
2 void entry ()
3 {
4   clear_screen ();
5   kprint ("Welcome to the OS!\n\0");
6 }
```

Figura 4.9 – Teste do *driver* para vídeo

4.6 Módulo 4: Implementação de um mecanismo de tratamento de interrupções

Interrupções são geradas por dispositivos de entrada/saída, ou até mesmo pelo próprio processador, para indicar a ocorrência de algum evento importante. Para que o processador possa lidar com os diferentes tipos de interrupções, uma tabela denominada IDT deve ser definida, contendo, para cada interrupção registrada, informações como o endereço do código que prepara o contexto para a execução da função de tratamento da

interrupção e informações que controlam o acesso à esta função. Posto isso, a interface deste módulo é definida pelos arquivos *include/idt.h* e *include/isr.h*, onde se destacam as seguintes funções: isr_install, responsável por configurar e inicializar o sistema de gerenciamento de interrupções; register_interrupt_handler, responsável por registrar uma função de tratamento para uma interrupção.

De maneira análoga à GDT, o endereço da IDT é carregado em um registrador específico da CPU, denominado IDRT, a partir da instrução Assembly lidt. Para organizar os códigos referentes às interrupções e códigos fortemente ligados ao processador, o diretório *cpu/* foi criado. Os arquivos *include/idt.h* e *cpu/idt.c* contém respectivamente o Código 4.25 e o Código 4.26, responsáveis por definir, carregar e modificar a IDT.

Código 4.25 – Cabeçalho para a definição da IDT

```
#ifndef IDT_H
#define IDT_H
#define IDT_ENTRIES 256
#define KERNEL_CS 0x08
#define low_16(address) (uint16_t)((address) & 0xFFFF)
#define high_16(address) (uint16_t)(((address) >> 16) & 0xFFFF)
#include <stdint.h>
typedef struct
{
  uint16_t low_offset;
  uint16_t sel;
  uint8_t always0;
  uint8_t flags;
  uint16_t high_offset;
}__attribute__((packed)) idt_gate_t;
typedef struct
{
  uint16_t limit;
  uint32_t base;
}__attribute__((packed)) idt_register_t;
idt_gate_t idt [IDT_ENTRIES];
idt_register_t idt_reg;
void set_idt_gate (uint32_t n, uint32_t handler);
void load_idt ();
#endif
```

Código 4.26 – Definição da IDT

```
#include <idt.h>
void set_idt_gate (uint32_t n, uint32_t handler)
{
  idt [n].low_offset = low_16(handler);
  idt [n].sel = KERNEL_CS;
  idt [n].always0 = 0;
```

```
7   idt [n].flags = 0x8E;
8   idt [n].high_offset = high_16(handler);
9 }
10 void load_idt()
11 {
12   idt_reg.base = (uint32_t) &idt;
13   idt_reg.limit = IDT_ENTRIES * sizeof(idt_gate_t) -1;
14   asm volatile ("lidtl (%0)" : : "r" (&idt_reg));
15 }
```

O Código 4.25 é responsável por declarar, além de alguns *defines*, as duas estruturas necessárias para a utilização da IDT. A estrutura que representa cada entrada da tabela, denominada idt_gate_t, é definida da seguinte maneira: os bits 0-15 armazenam os 16 bits de baixa ordem do endereço do código que lida com a interrupção; os bits 16-31 armazenam o seletor do segmento de código da função de interrupção, ou seja, segmento de código do núcleo; os bits 32-39 não são utilizados; os bits 40-43 definem o tipo da entrada, que nesse caso, é uma entrada de interrupção do processador 80386, no modo 32 bits; o bit 44 é denominado *Storage Segment*, e é definido como 0 para entradas de interrupções; os bits 45-46 definem o nível de privilégio mínimo do código que pode chamar a função que lida com a interrupção; o bit 47 define se a interrupção atual está sendo utilizada; os bits 48-63 armazenam os bits de alta ordem do endereço do código de lida com a interrupção. O GCC, na arquetura x86, buscando eficiência no acesso à memória, realiza um alinhamento de 4 bytes, adicionando bits que servem somente para preenchimento na *struct*. A diretiva ___attribute___((packed)) informa ao GCC que estes bits não devem ser adicionados, mantendo a *struct* com o tamanho realmente definido.

A *struct* seguinte, denominada idt_register_t é o descritor da IDT que será armazenado no registrador IDTR, informando o endereço inicial e o tamanho da tabela na memória. Um array de elementos do tipo idt_gate_t é declarado para representar a IDT em si, além de uma variável do tipo idt_register_t, que representa o descritor da tabela. O arquivo de cabeçalho ainda define duas funções: set_idt_gate, responsável por modificar uma entrada da IDT; load_idt, responsável por carregar a IDT no registrador IDTR.

O Código 4.26 implementa as duas funções apresentadas no arquivo de cabeçalho. A função set_idt_gate modifica uma entrada na IDT, recebendo como parâmetro o número da interrupção a ser modificada e o endereço da função. Vale ressaltar que as *flags* são iguais para todas as entradas da tabela: as entradas tem o bit *present* definido como 1; o nível mínimo para execução é o anel de privilégio 0; o tipo da entrara é uma entrada de interrupção para o processador 80386 de 32 bits; o segmento do código é o segmento de código definido no *flat model*. A função load_idt, a partir da instrução lidt, carrega o descritor da IDT no registrador IDTR.

No momento que uma interrupção ocorre, o processamento realizado no momento é suspenso até que a interrupção seja tratada. Para que nenhuma informação do processamento seja perdida e que a interrupção possa ser tratada de maneira correta, um arquivo denominado *cpu/interrupt.asm* foi criado, contendo o Código 4.27.

Automaticamente, no momento da ocorrência de uma interrupção, o processador empilha os valores dos registradores SS, ESP, EFLAGS, CS e EPI, que descrevem o estado atual do processador, e esses valores são desempilhados por meio da instrução iret. Em seguida, a partir do endereço disponibilizado pela IDT, o código apontado pela *label* associada à interrupção, definida no Código 4.27, é executado. As *labels*, que funcionam como pontos de entrada para esse código, são responsáveis por empilhar códigos de erro e o número da interrupção. Vale ressaltar que algumas exceções não necessitam que o código de erro seja empilhado, pois o próprio processador realiza este procedimento. Em seguida, o *stub* é chamado. Essa parte do código realiza operações gerais para todas as interrupções, como: executar a instrução pusha; empilhar segmento de dados utilizado pelo processamento que estava em execução antes da interrupção ocorrer; modificar o segmento de dados para o segmento de dados do núcleo; empilhar o endereço do topo da pilha; chamar a função de tratamento da interrupção; desempilhar valores empilhados anteriormente; retornar para a execução do código que estava sendo executado antes da ocorrência da interrupção.

Código 4.27 – Entrada e saída do tratamento de interrupções

```
1  [bits 32]
2  [extern isr_handler]
3  [extern irq_handler]
4  extern print_stack_asm
5  isr_common_stub:
6    pusha
7    mov ax, ds
8    push eax
9    mov ax, 0x10
10   mov ds, ax
11   mov es, ax
12   mov fs, ax
13   mov gs, ax
14   push esp
15   call isr_handler
16   pop eax
17   pop eax
18   mov ds, ax
19   mov es, ax
20   mov fs, ax
21   mov gs, ax
22   popa
```

```
23   add esp, 8
24   iret
25 irq_common_stub:
26   pusha
27   mov ax, ds
28   push eax
29   mov ax, 0x10
30   mov ds, ax
31   mov es, ax
32   mov fs, ax
33   mov gs, ax
34   push esp
35   call irq_handler
36   pop ebx
37   pop eax
38   mov ds, ax
39   mov es, ax
40   mov fs, ax
41   mov gs, ax
42   popa
43   add esp, 8
44   iret
45 global isr0
46 global isr1
47 global isr2
48 global isr3
49 global isr4
50 global isr5
51 global isr6
52 global isr7
53 global isr8
54 global isr9
55 global isr10
56 global isr11
57 global isr12
58 global isr13
59 global isr14
60 global isr15
61 global isr16
62 global isr17
63 global isr18
64 global isr19
65 global isr20
66 global isr21
67 global isr22
68 global isr23
69 global isr24
```

```
70 global isr25
71 global isr26
72 global isr27
73 global isr28
74 global isr29
75 global isr30
76 global isr31
77
78 global irq0
79 global irq1
80 global irq2
81 global irq3
82 global irq4
83 global irq5
84 global irq6
85 global irq7
86 global irq8
87 global irq9
88 global irq10
89 global irq11
90 global irq12
91 global irq13
92 global irq14
93 global irq15
94 isr0:
95     push byte 0
96     push byte 0
97     jmp isr_common_stub
98 isr1:
99     push byte 0
100    push byte 1
101    jmp isr_common_stub
102 isr2:
103    push byte 0
104    push byte 2
105    jmp isr_common_stub
106 isr3:
107    push byte 0
108    push byte 3
109    jmp isr_common_stub
110 isr4:
111    push byte 0
112    push byte 4
113    jmp isr_common_stub
114 isr5:
115    push byte 0
116    push byte 5
```

```
117     jmp isr_common_stub
118 isr6:
119     push byte 0
120     push byte 6
121     jmp isr_common_stub
122 isr7:
123     push byte 0
124     push byte 7
125     jmp isr_common_stub
126 isr8:
127     push byte 8
128     jmp isr_common_stub
129 isr9:
130     push byte 0
131     push byte 9
132     jmp isr_common_stub
133 isr10:
134     push byte 10
135     jmp isr_common_stub
136 isr11:
137     push byte 11
138     jmp isr_common_stub
139 isr12:
140     push byte 12
141     jmp isr_common_stub
142 isr13:
143     push byte 13
144     jmp isr_common_stub
145 isr14:
146     push byte 14
147     jmp isr_common_stub
148 isr15:
149     push byte 0
150     push byte 15
151     jmp isr_common_stub
152 isr16:
153     push byte 0
154     push byte 16
155     jmp isr_common_stub
156 isr17:
157     push byte 17
158     jmp isr_common_stub
159 isr18:
160     push byte 0
161     push byte 18
162     jmp isr_common_stub
163 isr19:
```

```
164     push byte 0
165     push byte 19
166     jmp isr_common_stub
167 isr20:
168     push byte 0
169     push byte 20
170     jmp isr_common_stub
171 isr21:
172     push byte 0
173     push byte 21
174     jmp isr_common_stub
175 isr22:
176     push byte 0
177     push byte 22
178     jmp isr_common_stub
179 isr23:
180     push byte 0
181     push byte 23
182     jmp isr_common_stub
183 isr24:
184     push byte 0
185     push byte 24
186     jmp isr_common_stub
187 isr25:
188     push byte 0
189     push byte 25
190     jmp isr_common_stub
191 isr26:
192     push byte 0
193     push byte 26
194     jmp isr_common_stub
195 isr27:
196     push byte 0
197     push byte 27
198     jmp isr_common_stub
199 isr28:
200     push byte 0
201     push byte 28
202     jmp isr_common_stub
203 isr29:
204     push byte 0
205     push byte 29
206     jmp isr_common_stub
207 isr30:
208     push byte 30
209     jmp isr_common_stub
210 isr31:
```

```
211     push byte 0
212     push byte 31
213     jmp isr_common_stub
214 irq0:
215   push byte 0
216   push byte 32
217   jmp irq_common_stub
218 irq1:
219   push byte 1
220   push byte 33
221   jmp irq_common_stub
222 irq2:
223   push byte 2
224   push byte 34
225   jmp irq_common_stub
226 irq3:
227   push byte 3
228   push byte 35
229   jmp irq_common_stub
230 irq4:
231   push byte 4
232   push byte 36
233   jmp irq_common_stub
234 irq5:
235   push byte 5
236   push byte 37
237   jmp irq_common_stub
238 irq6:
239   push byte 6
240   push byte 38
241   jmp irq_common_stub
242 irq7:
243   push byte 7
244   push byte 39
245   jmp irq_common_stub
246 irq8:
247   push byte 8
248   push byte 40
249   jmp irq_common_stub
250 irq9:
251   push byte 9
252   push byte 41
253   jmp irq_common_stub
254 irq10:
255   push byte 10
256   push byte 42
257   jmp irq_common_stub
```

```
258 irq11:
259   push byte 11
260   push byte 43
261   jmp irq_common_stub
262 irq12:
263   push byte 12
264   push byte 44
265   jmp irq_common_stub
266 irq13:
267   push byte 13
268   push byte 45
269   jmp irq_common_stub
270 irq14:
271   push byte 14
272   push byte 46
273   jmp irq_common_stub
274 irq15:
275   push byte 15
276   push byte 47
277   jmp irq_common_stub
```

As diretivas extern e global indicam que as *labels* são compartilhadas com outros arquivos, de maneira que a primeira indica que a *label* está definida em outro código, e a segunda indica que outros códigos tem acesso à *label*.

Para se conectar aos *stubs* definidos no Código 4.27, dois arquivos foram criados: *include/isr.h* e *cpu/isr.c*. O código contido em cada um deles são apresentados no Código 4.28 e Código 4.29 respectivamente.

Código 4.28 – Cabeçalho para o tratamento de interrupções

```
1  #ifndef ISR_H
2  #define ISR_H
3  #include <stdint.h>
4  #include <idt.h>
5  #include <vga.h>
6  #include <ports.h>
7  #include <string.h>
8  extern void isr0();
9  extern void isr1();
10 extern void isr2();
11 extern void isr3();
12 extern void isr4();
13 extern void isr5();
14 extern void isr6();
15 extern void isr7();
16 extern void isr8();
17 extern void isr9();
```

```
18 extern void isr10();
19 extern void isr11();
20 extern void isr12();
21 extern void isr13();
22 extern void isr14();
23 extern void isr15();
24 extern void isr16();
25 extern void isr17();
26 extern void isr18();
27 extern void isr19();
28 extern void isr20();
29 extern void isr21();
30 extern void isr22();
31 extern void isr23();
32 extern void isr24();
33 extern void isr25();
34 extern void isr26();
35 extern void isr27();
36 extern void isr28();
37 extern void isr29();
38 extern void isr30();
39 extern void isr31();
40
41 extern void irq0();
42 extern void irq1();
43 extern void irq2();
44 extern void irq3();
45 extern void irq4();
46 extern void irq5();
47 extern void irq6();
48 extern void irq7();
49 extern void irq8();
50 extern void irq9();
51 extern void irq10();
52 extern void irq11();
53 extern void irq12();
54 extern void irq13();
55 extern void irq14();
56 extern void irq15();
57 #define IRQ0 32
58 #define IRQ1 33
59 #define IRQ2 34
60 #define IRQ3 35
61 #define IRQ4 36
62 #define IRQ5 37
63 #define IRQ6 38
64 #define IRQ7 39
```

```
65 #define IRQ8 40
66 #define IRQ9 41
67 #define IRQ10 42
68 #define IRQ11 43
69 #define IRQ12 44
70 #define IRQ13 45
71 #define IRQ14 46
72 #define IRQ15 47
73 typedef struct
74 {
75   uint32_t ds;
76   uint32_t edi, esi, ebp, useless, ebx, edx, ecx, eax;
77   uint32_t int_no, err_code;
78   uint32_t eip, cs, eflags, esp, ss;
79 }registers_t;
80 typedef void (*isr_t) (registers_t *r);
81 void isr_install();
82 void isr_handler (registers_t *r);
83 void irq_handler (registers_t *r);
84 void register_interrupt_handler (uint8_t n, isr_t handler);
85 #endif
```

Código 4.29 – Tratamento de interrupções

```
1  #include <isr.h>
2  isr_t interrupt_handlers[256];
3  void isr_install()
4  {
5    set_idt_gate (0, (uint32_t)isr0);
6    set_idt_gate (1, (uint32_t)isr1);
7    set_idt_gate (2, (uint32_t)isr2);
8    set_idt_gate (3, (uint32_t)isr3);
9    set_idt_gate (4, (uint32_t)isr4);
10   set_idt_gate (5, (uint32_t)isr5);
11   set_idt_gate (6, (uint32_t)isr6);
12   set_idt_gate (7, (uint32_t)isr7);
13   set_idt_gate (8, (uint32_t)isr8);
14   set_idt_gate (9, (uint32_t)isr9);
15   set_idt_gate (10, (uint32_t)isr10);
16   set_idt_gate (11, (uint32_t)isr11);
17   set_idt_gate (12, (uint32_t)isr12);
18   set_idt_gate (13, (uint32_t)isr13);
19   set_idt_gate (14, (uint32_t)isr14);
20   set_idt_gate (15, (uint32_t)isr15);
21   set_idt_gate (16, (uint32_t)isr16);
22   set_idt_gate (17, (uint32_t)isr17);
23   set_idt_gate (18, (uint32_t)isr18);
24   set_idt_gate (19, (uint32_t)isr19);
```

```
25    set_idt_gate (20, (uint32_t)isr20);
26    set_idt_gate (21, (uint32_t)isr21);
27    set_idt_gate (22, (uint32_t)isr22);
28    set_idt_gate (23, (uint32_t)isr23);
29    set_idt_gate (24, (uint32_t)isr24);
30    set_idt_gate (25, (uint32_t)isr25);
31    set_idt_gate (26, (uint32_t)isr26);
32    set_idt_gate (27, (uint32_t)isr27);
33    set_idt_gate (28, (uint32_t)isr28);
34    set_idt_gate (29, (uint32_t)isr29);
35    set_idt_gate (30, (uint32_t)isr30);
36    set_idt_gate (31, (uint32_t)isr31);
37    port_byte_out(0x20, 0x11);
38    port_byte_out(0xA0, 0x11);
39    port_byte_out(0x21, 0x20);
40    port_byte_out(0xA1, 0x28);
41    port_byte_out(0x21, 0x04);
42    port_byte_out(0xA1, 0x02);
43    port_byte_out(0x21, 0x01);
44    port_byte_out(0xA1, 0x01);
45    port_byte_out(0x21, 0x0);
46    port_byte_out(0xA1, 0x0);
47    set_idt_gate (32, (uint32_t)irq0);
48    set_idt_gate (33, (uint32_t)irq1);
49    set_idt_gate (34, (uint32_t)irq2);
50    set_idt_gate (35, (uint32_t)irq3);
51    set_idt_gate (36, (uint32_t)irq4);
52    set_idt_gate (37, (uint32_t)irq5);
53    set_idt_gate (38, (uint32_t)irq6);
54    set_idt_gate (39, (uint32_t)irq7);
55    set_idt_gate (40, (uint32_t)irq8);
56    set_idt_gate (41, (uint32_t)irq9);
57    set_idt_gate (42, (uint32_t)irq10);
58    set_idt_gate (43, (uint32_t)irq11);
59    set_idt_gate (44, (uint32_t)irq12);
60    set_idt_gate (45, (uint32_t)irq13);
61    set_idt_gate (46, (uint32_t)irq14);
62    set_idt_gate (47, (uint32_t)irq15);
63    load_idt();
64 }
65 char *exception_messages[] = {
66    "Division By Zero",
67    "Debug",
68    "Non Maskable Interrupt",
69    "Breakpoint",
70    "Into Detected Overflow",
71    "Out of Bounds",
```

```
72    "Invalid Opcode",
73    "No Coprocessor",
74
75    "Double Fault",
76    "Coprocessor Segment Overrun",
77    "Bad TSS",
78    "Segment Not Present",
79    "Stack Fault",
80    "General Protection Fault",
81    "Page Fault",
82    "Unknown Interrupt",
83
84    "Coprocessor Fault",
85    "Alignment Check",
86    "Machine Check",
87    "Reserved",
88    "Reserved",
89    "Reserved",
90    "Reserved",
91    "Reserved",
92
93    "Reserved",
94    "Reserved",
95    "Reserved",
96    "Reserved",
97    "Reserved",
98    "Reserved",
99    "Reserved",
100   "Reserved"
101 };
102 void isr_handler (registers_t *r)
103 {
104   kprint_debug ("interrupt received: ", LIGHT_RED);
105   char s[3];
106   itoa (r->int_no, s);
107   kprint_debug (s, LIGHT_RED);
108   kprint ("\n");
109   kprint_debug (exception_messages[r->int_no], LIGHT_RED);
110   kprint ("\n");
111   if (interrupt_handlers [r->int_no] != 0)
112   {
113     isr_t handler = interrupt_handlers [r->int_no];
114     handler (r);
115   }
116 }
117 void register_interrupt_handler (uint8_t n, isr_t handler)
118 {
```

```
119   interrupt_handlers [n] = handler;
120 }
121 void irq_handler (registers_t *r)
122 {
123   if (r->int_no >= 40)
124     port_byte_out (0xa0, 0x20);
125   port_byte_out (0x20, 0x20);
126   if (interrupt_handlers [r->int_no] != 0)
127   {
128     isr_t handler = interrupt_handlers [r->int_no];
129     handler (r);
130   }
131 }
```

As *labels* definidas como *global* no Código 4.27 são definidas como *extern* no Código 4.28, permitindo a utilização em ambos os arquivos. A *struct* registers_t mapeia exatamente os valores empilhados automaticamente pelo processador e pelo *stub*. Como o último valor a ser empilhado foi o endereço do topo da pilha, as funções de tratamento chamadas pelo *stub* recebem um ponteiro para essa *struct* como parâmetro. Não se faz necessária a utilização da diretiva ___attribute___((packed)), pois essa *struct* já está naturalmente alinhada, tendo em vista que todos os seus componentes possuem 4 bytes. Como o núcleo irá prover uma interface para que *drivers* possam definir suas próprias funções de tratamento de interrupções, o tipo isr_t foi definido como um ponteiro de função que indica como os *drivers* devem definir as funções de tratamento de interrupções.

O Código 4.29 é responsável por implementar as funções apresentadas no Código 4.28. O vetor interrupt_handlers é responsável por armazenar todas as funções de tratamento de interrupções do sistema operacional, sendo modificado a partir da função register_interrupt_handler, que recebe como parâmetro o número da interrupção a ser registrada, e um ponteiro para a função a ser executada. Vale ressaltar que o processador i386 exige que as primeiras 32 interrupções sejam tratadas, pois estas caracterizam as exceções.

Inicialmente, as exceções são tratadas a partir da impressão de uma string que a identifica. Para este propósito, o vetor exception_messages armazena as strings referentes à cada exceção, de modo que a função isr_handler verifica o número da interrupção na pilha e imprime a strings correspondente. Para interrupções de hardware, é esperado que o *driver* do dispositivo tenha registrado a função de tratamento, pois a função irq_handler acessa o vetor interrupt_handlers e executa esta função.

O processador recebe interrupções de hardware através do dispositivo PIC, que gerencia as interrupções dos dispositivos. O PIC inicialmente possui números de interrupções padrões para cada uma das 16 linhas de IRQ. As IRQs que chegam pelas linhas 0..7 são mapeadas nas interrupções 0x8..0xF. As IRQs que chegam pelas linhas 8..15 são

mapeadas nas interrupções 0x70..0x77. As requisições de interrupções do *Master PIC* causam conflitos com as exceções, pois ambas disparam interrupções 0x8.. 0xF. Para contornar este problema, o PIC requer um remapeamento das interrupções geradas pelo *Master PIC*, que é realizada na função ist_install. Esta função, além de conter o código que registra as funções das *labels* presentes no Código 4.27 na IDT, faz o mapeamento necessário das interrupções geradas pelo PIC.

A porta de endereço 0x20 é a *command port* do *Master PIC*, enquanto a porta endereçada por 0xA0 é a *command port* do *Slave PIC*. A porta de endereço 0x21 é a *data port* do *Master PIC*, enquanto a porta de endereço 0xA1 é a *data port* do *stave PIC*. O valor 0x11 é enviado para ambas as *command ports*, em que os bits referentes ao valor 0x10 indicam a inicialização do PIC, e o valor do bit 0x1 indica o modo de operação, que neste caso, é o modo de operação *8086/88 (MCS-80/85) mode*, para o processador Intel 8086. Após a inicialização, o PIC espera mais três palavras de inicialização, denominadas ICW. As ICWs indicam o espaço de mapeamento das IRQs para interrupções, a forma de ligação entre o *Master PIC* e o *Slave PIC*, e informações adicionais sobre o ambiente. A próxima escrita nas portas de dados indicará a interrupção que as IRQs gerarão, permitindo o remapeamento das interrupções geradas pelo PIC. A escrita do valor 32 (0x20) na porta de dados do *Master PIC* indica que as interrupções geradas por este iniciarão a partir do número 32, e a escrita do valor 40 (0x28) na porta de dados do *Slave PIC* indica que as IRQs geradas por este iniciarão a partir da interrupção 40. A escrita do valor 0x4 na porta de dados do *Master PIC* indica que o *Slave PIC* está conectado ao pino número 2 do *master PIC* através de um mapa de bits, onde o valor 4 atribui ao bit 2 o valor 1. A escrita do valor 0x2 na porta de dados do *Slave PIC* indica que este operará em modo cascata, ou seja, estará conectado à uma entrada de IRQ de outro PIC. A escrita do valor 0x1 na porta de dados de ambas as portas de dados são redefinições do modo de operação, indicando o modo de operação *8086/88 (MCS-80/85) mode*. Após a definição das palavras de inicialização, a escrita na porta de dados altera a máscara de interrupções do PIC. Um registrador interno, denominado IMR, funciona como um mapa de bits que indica quais linhas de IRQs serão ignoradas. Como nenhuma linha será ignorada neste trabalho, o valor 0 é escrito em ambas as porta de dados.

O código principal do núcleo deve ser alterado para o teste deste módulo, como mostra o Código 4.30. A função isr_intall povoa a IDT com as *labels* corretas, configura o PIC e carrega a IDT no registrador IDTR. Logo após as interrupções, que foram desabilitadas no momento da ativação do *Protected Mode*, são reativadas a partir da instrução sti e a interrupção de número 0 é chamada. A adição dos novos códigos requer algumas alterações no *Makefile*, como: adicionar o diretório *cpu/* na lista de diretórios que contém códigos fonte C e na lista de diretórios a serem limpos quando a diretiva clean for utilizada; adicionar o arquivo *cpu/interrupts.o* como arquivo objeto necessário. A Figura 4.10 apresenta a execução deste módulo.

Código 4.30 – Função principal do núcleo para teste do mecanismo de tratamento de interrupções

```
#include <stdint.h>
#include <vga.h>
#include <isr.h>

void entry ()
{
  clear_screen ();
  kprint ("Welcome to the OS!\n\0");
  isr_install ();
  asm volatile ("sti");
  asm ("int $0");
}
```

Figura 4.10 – Teste do mecanismo de tratamento de interrupções

4.7 Módulo 5: Implementação de um *driver* para o PIT

O PIT é um dispositivo que pode ser utilizado para medição de tempo e sincronização de tarefas, pois ele é capaz de enviar sinais em tempos determinados por software. Neste trabalho, suas funcionalidades são utilizadas na seção 4.12, onde está descrita a implementação e gerenciamento de tarefas. Para que seja possível sua utilização, este módulo deve proporcionar a interface definida no arquivo *inclide/pit.h*, que contém o Código 4.31. Este arquivo contém macros que são utilizadas na implementação e utilização do *driver*, além das seguintes funções: pit_init, responsável por iniciar sua execução em uma frequência definida pelo parâmetro; função pit_callback, responsável por implementar o que deve ser executado quando o PIT gerar uma interrupção.

Código 4.31 – Cabeçalho para o *driver* do PIT

```
#ifndef PIT_H
#define PIT_H
#define PIT_INPUT_FREQUENCY 1193180
```

```
#define PIT_DATA_PORT_CH0 0x40
#define PIT_DATA_PORT_CH1 0x41
#define PIT_DATA_PORT_CH2 0x42
#define PIT_CTRL_PORT 0x43
#define PIT_CTRL_VALUE 0b00110100
#include <stdint.h>
#include <string.h>
#include <ports.h>
#include <vga.h>
#include <isr.h>
#include <mem.h>
void pit_init (uint16_t reload_register);
uint32_t tick = 0;
#endif
```

O Código 4.32, presente no arquivo *include/pit.c* é responsável por implementar o *driver* do PIT. A função de inicialização pit_init recebe um parâmetro de 16 bits, referente ao valor inicial do contador do PIT. Este valor é dividido em duas variáveis de oito bits para que possam ser carregadas na porta de dados do canal 0. O registrador de controle é escrito com o valor definido na macro PIT_CTRL_VALUE, que define o seguinte comportamento: codificação binária dos valores; modo de operação *rate_generator*; modo de acesso à porta de dados em que a primeira escrita se refere aos bits de baixa ordem do valor inicial do contador, e a segunda escrita se refere aos bits de alta ordem do valor inicial do contador; configurações referentes ao canal 0.

O modo de operação *rate generator* indica que o PIT executará a função de divisor de frequência. Desta maneira o valor do contador, inicializado com o valor escrito na porta de dados, é decrementado a cada pulso. Quando o valor do contador chegar a 1, este é reescrito com o valor inicial, e o próximo sinal gerará uma interrupção. Existem duas peculiaridades que devem ser levadas em consideração neste modo de operação: tendo em vista que o contador possui 16 bits, a atribuição do valor 0 à este representa um divisor de 65536 (ou 10000 caso a codificação BCD seja utilizada); o valor 1 não deve ser utilizado como valor inicial do contador, pois a reescrita deste ocorre na mudança do valor 2 para o valor 1.

O Código 4.32 também implementa a função que é executada quando o processador recebe uma interrupção do PIT. Inicialmente, esta função somente incrementa uma variável que é inicializada com o valor 0, e imprime o seu valor atual. Como esta variável não é local, seu valor é mantido a cada chamada desta função.

Código 4.32 – *Driver* do PIT

```
#include <pit.h>
static void pit_callback (registers_t *regs);
static void pit_callback (registers_t *regs)
```

```
4 {
5   tick++;
6   kprintf ("Tick: %d\n", 1, tick);
7
8 }
9 void pit_init (uint16_t reload_register) {
10   register_interrupt_handler(IRQ0, pit_callback);
11   uint8_t low  = (uint8_t)(reload_register & 0xFF);
12   uint8_t high = (uint8_t)((reload_register >> 8) & 0xFF);
13   port_byte_out (PIT_CTRL_PORT, PIT_CTRL_VALUE);
14   port_byte_out (PIT_DATA_PORT_CH0, low);
15   port_byte_out (PIT_DATA_PORT_CH0, high);
16 }
```

Para realizar o teste deste módulo, poucas alterações no código principal do núcleo são necessárias como mostra o Código 4.33, e nenhuma modificação no Makefile é requerida. O maior divisor foi utilizado para a melhor visualização do teste. A execução deste módulo é apresentada na Figura 4.11.

Código 4.33 – Função principal do núcleo para teste do *driver* do PIT

```
1 #include <stdint.h>
2 #include <vga.h>
3 #include <isr.h>
4 #include <pit.h>
5 void entry ()
6 {
7   clear_screen ();
8   kprint ("Welcome to the James OS!\n\0");
9   isr_install ();
10   asm volatile ("sti");
11   pit_init (0x0);
12 }
```

4.8 Módulo 6: Implementação de um *driver* para o teclado

O teclado é um dos periféricos mais importantes de um computador, pois, a partir dele, o usuário é capaz de interagir com o sistema. A cada tecla pressionada, o controlador do teclado envia uma requisição de interrupção ao PIC e armazena o valor da tecla pressionada em um *buffer* interno, endereçado na porta de entrada/saída 0x60. O arquivo *include/keyboard.h*, que define a interface deste módulo, juntamente com o arquivo *drivers/keyboard.c*, contendo respectivamente o Código 4.34 e o Código 4.35 são responsáveis por implementar o *driver* para um teclado.

Código 4.34 – Cabeçalho para o *driver* de teclado

Figura 4.11 – Teste do *driver* para o PIT

```
#ifndef KEYBOARD_H
#define KEYBOARD_H
#define KEYBOARD_SCANCODE 0x60
#include <stdint.h>
#include <ports.h>
#include <isr.h>
#include <string.h>
#include <vga.h>
void keyboard_init ();
#endif
```

Código 4.35 – *Driver* de teclado

```
#include <keyboard.h>
static void print_letter (uint8_t scancode);
static void keyboard_callback (registers_t *regs);
static void keyboard_callback (registers_t *regs)
{
  uint8_t scancode = port_byte_in(KEYBOARD_SCANCODE);
  char sc_ascii[10];
  itoa (scancode, sc_ascii);
  kprint_debug ("scan_code: ", WHITE);
  kprint_debug (sc_ascii, WHITE);
  kprint_debug (" (", WHITE);
  print_letter (scancode);
  kprint_debug (")", WHITE);
  kprint("\n");
}
```

```
16 void keyboard_init ()
17 {
18   register_interrupt_handler(IRQ1, keyboard_callback);
19 }
20 static void print_letter (uint8_t scancode)
21 {
22   switch (scancode) {
23     case 0x0: kprint_debug ("ERROR", WHITE); break;
24     case 0x1: kprint_debug ("esc", WHITE); break;
25     case 0x2: kprint_debug ("1", WHITE); break;
26     case 0x3: kprint_debug ("2", WHITE); break;
27     case 0x4: kprint_debug ("3", WHITE); break;
28     case 0x5: kprint_debug ("4", WHITE); break;
29     case 0x6: kprint_debug ("5", WHITE); break;
30     case 0x7: kprint_debug ("6", WHITE); break;
31     case 0x8: kprint_debug ("7", WHITE); break;
32     case 0x9: kprint_debug ("8", WHITE); break;
33     case 0x0A: kprint_debug ("9", WHITE); break;
34     case 0x0B: kprint_debug ("0", WHITE); break;
35     case 0x0C: kprint_debug ("-", WHITE); break;
36     case 0x0D: kprint_debug ("+", WHITE); break;
37     case 0x0E: kprint_debug ("backspace", WHITE); break;
38     case 0x0F: kprint_debug ("tab", WHITE); break;
39     case 0x10: kprint_debug ("q", WHITE); break;
40     case 0x11: kprint_debug ("w", WHITE); break;
41     case 0x12: kprint_debug ("e", WHITE); break;
42     case 0x13: kprint_debug ("r", WHITE); break;
43     case 0x14: kprint_debug ("t", WHITE); break;
44     case 0x15: kprint_debug ("y", WHITE); break;
45     case 0x16: kprint_debug ("u", WHITE); break;
46     case 0x17: kprint_debug ("i", WHITE); break;
47     case 0x18: kprint_debug ("o", WHITE); break;
48     case 0x19: kprint_debug ("p", WHITE); break;
49     case 0x1A: kprint_debug ("[", WHITE); break;
50     case 0x1B: kprint_debug ("]", WHITE); break;
51     case 0x1C: kprint_debug ("enter", WHITE); break;
52     case 0x1D: kprint_debug ("left_ctrl", WHITE); break;
53     case 0x1E: kprint_debug ("a", WHITE); break;
54     case 0x1F: kprint_debug ("s", WHITE); break;
55     case 0x20: kprint_debug ("d", WHITE); break;
56     case 0x21: kprint_debug ("f", WHITE); break;
57     case 0x22: kprint_debug ("g", WHITE); break;
58     case 0x23: kprint_debug ("h", WHITE); break;
59     case 0x24: kprint_debug ("j", WHITE); break;
60     case 0x25: kprint_debug ("k", WHITE); break;
61     case 0x26: kprint_debug ("l", WHITE); break;
62     case 0x27: kprint_debug (";", WHITE); break;
```

```
63      case 0x28: kprint_debug ("'", WHITE); break;
64      case 0x29: kprint_debug ("`", WHITE); break;
65      case 0x2A: kprint_debug ("left_shift", WHITE); break;
66      case 0x2B: kprint_debug ("\\", WHITE); break;
67      case 0x2C: kprint_debug ("z", WHITE); break;
68      case 0x2D: kprint_debug ("x", WHITE); break;
69      case 0x2E: kprint_debug ("c", WHITE); break;
70      case 0x2F: kprint_debug ("v", WHITE); break;
71      case 0x30: kprint_debug ("b", WHITE); break;
72      case 0x31: kprint_debug ("n", WHITE); break;
73      case 0x32: kprint_debug ("m", WHITE); break;
74      case 0x33: kprint_debug (",", WHITE); break;
75      case 0x34: kprint_debug (".", WHITE); break;
76      case 0x35: kprint_debug ("/", WHITE); break;
77      case 0x36: kprint_debug ("right_shift", WHITE); break;
78      case 0x37: kprint_debug ("*", WHITE); break;
79      case 0x38: kprint_debug ("left_alt", WHITE); break;
80      case 0x39: kprint_debug ("space", WHITE); break;
81      default:
82        if (scancode <= 0x7f)
83          kprint_debug ("unknown", WHITE);
84        else if (scancode <= 0x39 + 0x80)
85        {
86          print_letter(scancode - 0x80);
87          kprint_debug (" released", WHITE);
88        }
89        else
90          kprint_debug ("unknown released", WHITE);
91        break;
92    }
93  }
```

O Código 4.35 possui a implementação de três funções. A função keyboard_init, que possui o cabeçalho definido no arquivo Código 4.34, é responsável por registrar a função de tratamento da interrupção gerada pelo teclado na entrada IRQ de número 1. A função keyboard_callback é responsável por tratar o evento de interrupções lançadas pelo teclado. Inicialmente, o *buffer* é acessado utilizando a macro KEYBOARD_SCANCODE definida no Código 4.34, que identifica o seu endereço. Após a obtenção do *scancode* da tecla que gerou a interrupção, algumas informações são impressas, e a função print_letter é chamada. A partir do *scancode* recebido por parâmetro, essa função imprime o valor associado à tecla. A ação de "soltar" uma tecla gera um *scancode* diferente do gerado ao pressioná-la, de modo que o valor gerado ao pressionar uma tecla, acrescido do valor 0x80, resulta no *scancode* gerado quando a mesma tecla "liberada".

Para testar a nova versão do sistema, duas alterações no código do núcleo são

necessárias: a inclusão do arquivo de cabeçalho do *driver* para teclado, e a chamada da função keyboard_init. Nenhuma alteração no Makefile é necessária. Para melhor visualização do teste, as mensagens de debug do *driver* do PIT foram retiradas. A Figura 4.12 apresenta a execução desta versão do sistema.

Figura 4.12 – Teste do *driver* para o teclado

4.9 Módulo 7: Implementação de um mecanismo de gerenciamento para uma área de *heap*

A área de *heap* pode ser vista como um espaço da memória que é acessado através de uma interface controlada pelo núcleo do sistema operacional. Seguindo o mesmo princípio da *heap* do espaço de endereçamento de processos, este módulo do sistema operacional tem como objetivo disponibilizar uma forma de alocação de memória explícita.

O modo de operação do gerenciamento da *heap* consiste em um espaço de memória separado do código da imagem do sistema operacional, sendo acessado com o auxílio de um mapa de bits e permitindo que espaços sejam reutilizados. A interface deste módulo é descrita no arquivo de cabeçalho *include/kheap.h*, que além de alguns *defines*, possui a declaração das seguintes funções: kheap_init, kfree, kmalloc, kmalloc_u, print_bit_map. Os arquivos *include/kheap.h* e *libc/kheap.c* contém, respectivamente, o Código 4.36 e o Código 4.37.

Código 4.36 – Cabeçalho para o mecanismo de gerenciamento da área de *heap*

```
1 #ifndef KMALLOC_H
```

```
2 #define KMALLOC_H
3 #include <stdint.h>
4 #include <mem.h>
5 #include <string.h>
6 #include <vga.h>
7 #define BITSET_BASE 0x10c800
8 #define BITSET_LIMIT 0x100000
9 #define HEAP_BASE 0x171000
10 void kheap_init ();
11 void kfree (uint32_t size, uint32_t addr);
12 uint32_t kmalloc (uint32_t sz);
13 uint32_t kmalloc_u (uint32_t sz);
14 void print_bit_map (uint32_t size) ;
15 #endif
```

Código 4.37 – Mecanismo de gerenciamento da área de *heap*

```
1 #include <kheap.h>
2 static uint32_t kmalloc_int (uint32_t size, uint32_t align);
3 static uint32_t kheap_malloc (uint32_t size);
4 static void fill (uint32_t begin, uint32_t size, uint8_t value);
5 uint8_t *bitset_base = (uint8_t*) BITSET_BASE;
6 uint8_t *bitset_limit = (uint8_t*) BITSET_LIMIT;
7 uint8_t *heap_base = (uint8_t*) HEAP_BASE;
8 uint32_t kheap_enable = 0;
9 uint32_t kmalloc_u (uint32_t sz)
10 {
11   return kmalloc_int (sz, 0);
12 }
13 uint32_t kmalloc (uint32_t sz)
14 {
15   return kmalloc_int (sz, 1);
16 }
17 void kheap_init ()
18 {
19   kheap_enable = 1;
20   memset ((uint8_t*)BITSET_LIMIT, 0 ,BITSET_BASE - BITSET_LIMIT);
21 }
22 static uint32_t kmalloc_int (uint32_t size, uint32_t align)
23 {
24   uint8_t *i;
25   uint32_t count = 0;
26   uint32_t begin = 0;
27   if (align && (size & 0x00000fff))
28   {
29     size += 0x1000;
30     size &= 0xfffff000;
31   }
```

```
32   for (i = bitset_base; i > bitset_limit ; i--)
33   {
34     int j;
35     for (j = 7; j >= 0; j--)
36     {
37       begin++;
38       if ((*i >> j) & 1)
39         count = 0;
40       else
41         count++;
42       if (count == size)
43       {
44         fill (begin - size, size, 1);
45         return (uint32_t)(heap_base - (begin - size) - size);
46       }
47     }
48   }
49   return 0;
50 }
51 static void fill (uint32_t begin, uint32_t size, uint8_t value)
52 {
53   uint32_t i;
54   for (i = begin; i < (size + begin); i++)
55   {
56     uint32_t byte = i / 8;
57     uint32_t offset = i % 8;
58 
59     if (value)
60       *(bitset_base - byte) |= (1 << (7 - offset));
61     else
62       *(bitset_base - byte) &= ((1 << 8) - 1) - (1 << (7 - offset));
63   }
64 
65 }
66 void kfree (uint32_t size, uint32_t addr)
67 {
68   uint32_t byte = (uint32_t) HEAP_BASE - addr;
69   fill (byte, size, 0);
70 }
71 void print_bit_map (uint32_t size)
72 {
73   uint8_t *i = bitset_limit;
74   for (i = bitset_base ; i > (bitset_base - size); i --)
75   {
76     int8_t j;
77     for (j = 7 ; j >= 0 ; j--)
78     {
```

```
79        if ((*i >> j) & 1)
80          kprint_debug ("1", CYAN);
81        else
82          kprint_debug ("0", CYAN);
83      }
84    }
85 }
```

O Código 4.36 contém o cabeçalho das funções que compões a interface de acesso ao gerenciamento da *heap*, e *defines* que auxiliam a manipulação da mesma. Os *defines* BITSET_BASE e BITSET_LIMIT armazenam os endereços de memória que delimitam o mapa de bits utilizado para mapear a área de *heap*, enquanto que o *define* HEAP_BASE armazena o endereço de início da *heap*. Vale ressaltar que neste projeto, o tamanho máximo permitido pelo mapa de bits é equivalente a 100 páginas, e o crescimento da *heap* acontece partido dos endereços mais altos para os endereços mais baixos.

A função kheap_init tem como objetivo inicializar o mapa de bits, preenchendo-o completamente com o valor 0, indicando que todos os bytes da *heap* estão livres. A função fill acessa o mapa de bits e preenche, bit a bit, uma determinada área com um valor específico. Esta função recebe a posição do bit que representa o início da memória desejada, e o tamanho desta, além do valor a ser escrito. Não é possível endereçar somente um bit na memória, portanto as seguintes operações são realizadas para que seja possível a escrita bit a bit: encontrar o byte do mapa de bits que contém o bit a ser escrito; encontrar o deslocamento do bit dentro do byte encontrado anteriormente; escrever, através de operações lógicas "AND, OR" o valor especificado por parâmetro. Para escrever o valor 0, o byte que contém o bit a ser escrito é submetido à uma operação AND (&) com um byte que contém todos os bits 1, exceto o bit que se deseja escrever. Para escrever o valor 1, o byte que contém o bit a ser escrito é submetido à uma operação OR (|) com um byte que contém todos os bits 0, exceto o bit que se deseja escrever.

A função kfree tem como objetivo liberar um espaço da *heap*, alterando o valor referente à este espaço no mapa de bits através da função fill. As funções kmalloc e kmalloc_u são responsáveis por alocar um espaço de memória na *heap*, de modo que a função kmalloc sempre aloca espaços alinhados a 4KB (*page align*), e a função kmalloc_u não alinha os endereços requisitados. Ambas as funções chamam a função estática kmalloc_int,

A função kmalloc_int alinha tamanho requisitado caso necessário e percorre o mapa de bits verificando o primeiro espaço que consegue acomodar o espaço requerido (*first-fit*). Quando o espaço é encontrado, a função fill é chamada, passando como parâmetro o bit inicial do mapeamento e o tamanho requerido. A função print_bit_map é utilizada para analisar o estado atual do mapa de bits. O parâmetro que esta função recebe indica a quantidade de bytes que serão impressos por esta função.

Para realizar os testes deste módulo e para melhor visualização, duas versões do código de entrada do núcleo foram utilizadas, como mostra o Código 4.38 e o Código 4.39. O primeiro utiliza a função kmalloc_u, alocando um espaço não alinhado de 7 bytes e liberando os 3 primeiros logo em seguida. O segundo utiliza a função kmalloc, que aloca espaços alinhados à 4096 bytes. A Figura 4.13 apresenta o resultado das duas execuções deste módulo.

Código 4.38 – Função principal do núcleo para o teste do mecanismo de gerenciamento da área de *heap* utilizando a função kmalloc_u

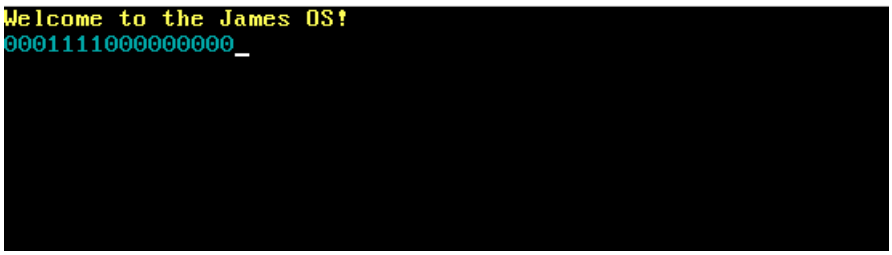
```
#include <stdint.h>
#include <vga.h>
#include <isr.h>
#include <keyboard.h>
#include <kheap.h>
#include <pit.h>
void entry ()
{
  clear_screen ();
  kprint ("Welcome to the James OS!\n\0");
  isr_install ();
  asm volatile ("sti");
  pit_init (PIT_INPUT_FREQUENCY);
  keyboard_init();
  kheap_init ();
  kmalloc_u (7);
  kfree (3, HEAP_BASE);
  print_bit_map (2);
}
```

Código 4.39 – Função principal do núcleo para o teste do mecanismo de gerenciamento da área de *heap* utilizando a função kmalloc

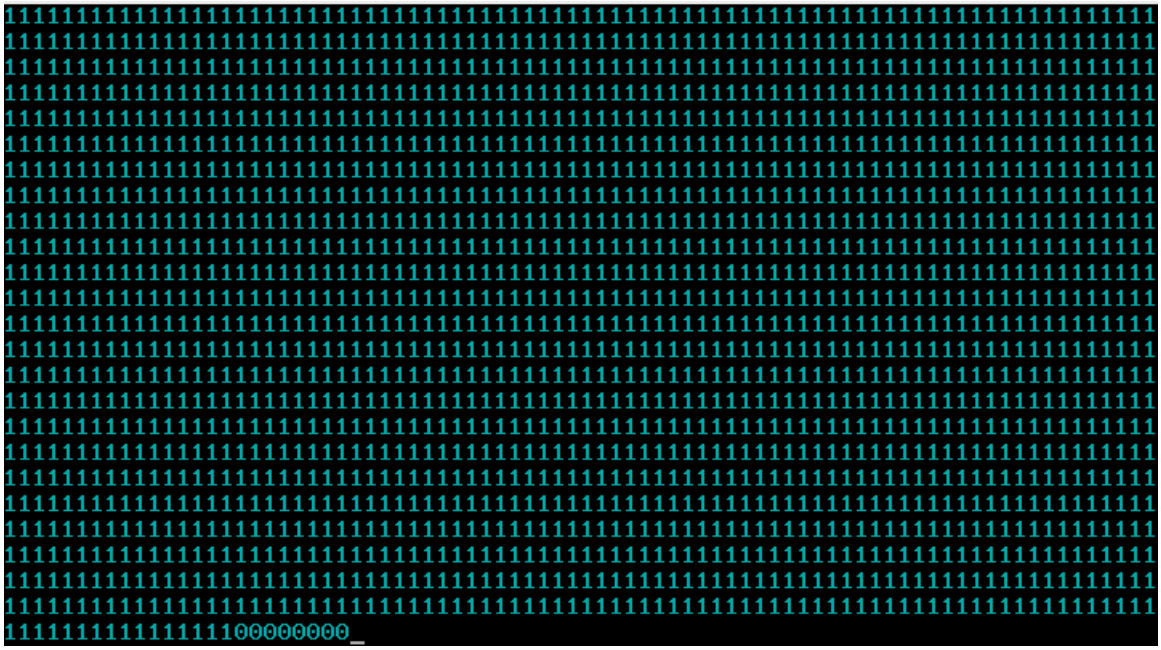
```
#include <stdint.h>
#include <vga.h>
#include <isr.h>
#include <keyboard.h>
#include <kheap.h>
#include <pit.h>

void entry ()
{
  clear_screen ();
  kprint ("Welcome to the James OS!\n\0");
  isr_install ();
  asm volatile ("sti");
  pit_init (PIT_INPUT_FREQUENCY);
  keyboard_init();
```

```
16   kheap_init ();
17   kmalloc (7);
18   print_bit_map (513);
19 }
```



(a)



(b)

Figura 4.13 – Teste do mecanismo de gerenciamento da área de *heap*

4.10 Módulo 8: Implementação de um mecanismo de gerenciamento de memória virtual utilizando paginação

O processador Intel 80386 conta com o auxílio de uma MMU que suporta a utilização de memória virtual pelo método de paginação. Este módulo é responsável por habilitar e gerenciar este recurso, além de disponibilizar uma interface implementada no arquivo *include/paging.h*, que contém o Código 4.40. O código que habilita e gerencia a memória virtual utilizando a interface citada foi escrito no arquivo *libc/paging.c*, que contém o Código 4.41.

Código 4.40 – Cabeçalho para o mecanismo de gerenciamento da memória virtual utilizando paginação

```
#ifndef PAGING_H
#define PAGING_H
#include <stdint.h>
#include <isr.h>
#include <kheap.h>
#define INDEX_FROM_BIT(a) (a/(8))
#define OFFSET_FROM_BIT(a) (a%(8))
typedef struct page_dir_entry {
    uint32_t present    : 1;
    uint32_t rw         : 1;
    uint32_t user       : 1;
    uint32_t w_through  : 1;
    uint32_t cache      : 1;
    uint32_t accessed   : 1;
    uint32_t reserved   : 1;
    uint32_t page_size  : 1;
    uint32_t ignored    : 1;
    uint32_t available  : 3;
    uint32_t frame      : 20;
}page_dir_entry_t;
typedef struct page_table_entry {
    uint32_t present    : 1;
    uint32_t rw         : 1;
    uint32_t user       : 1;
    uint32_t reserved   : 2;
    uint32_t accessed   : 1;
    uint32_t dirty      : 1;
    uint32_t reserved2  : 2;
    uint32_t available  : 3;
    uint32_t frame      : 20;
} page_table_entry_t;
typedef struct page_table
{
  page_table_entry_t pages [1024];
} page_table_t;
typedef struct page_directory
{
  page_dir_entry_t tables_physical [1024];
  page_table_t *tables [1024];
}page_directory_t;
void paging_init ();
void switch_page_directory (page_directory_t *dir, uint8_t phy);
page_table_entry_t* get_page (uint32_t address, int make,
   page_directory_t *dir);
uint32_t virtual2phys (page_directory_t* dir, uint32_t virtual_addr);
```

```
45 page_directory_t *kernel_directory=0;
46 page_directory_t *current_directory=0;
47 #endif
```

Código 4.41 – Mecanismo de gerenciamento da memória virtual utilizando paginação

```
1  #include <paging.h>
2  static void enable_paging ();
3  static void page_fault_handler (registers_t* regs);
4  static void set_frame (uint32_t frame_addr);
5  static void clear_frame (uint32_t frame_addr);
6  static uint32_t first_frame ();
7  static void alloc_page (page_table_entry_t *page, uint32_t is_kernel,
       int32_t is_writeable);
8  uint8_t page_enabled = 0;
9  uint8_t *bitmap;
10 uint32_t nframes;
11 static void set_frame (uint32_t frame_addr)
12 {
13   uint32_t idx = INDEX_FROM_BIT (frame_addr);
14   uint8_t off = OFFSET_FROM_BIT (frame_addr);
15   bitmap[idx] |= (0x1 << off);
16 }
17 static void clear_frame (uint32_t frame_addr)
18 {
19   uint32_t idx = INDEX_FROM_BIT (frame_addr);
20   uint8_t off = OFFSET_FROM_BIT (frame_addr);
21   bitmap[idx] &= ~(0x1 << off);
22 }
23 static uint32_t first_frame ()
24 {
25   uint32_t i, j;
26   for (i = 0; i < INDEX_FROM_BIT (nframes); i++)
27     if (bitmap[i] != 0xFF)
28       for (j = 0; j < 8; j++)
29         if (!(bitmap[i] & (1 << j)) )
30           return i * 8 + j;
31   return  -1;
32 }
33 static void alloc_page (page_table_entry_t *page, uint32_t is_kernel,
     int32_t is_writeable)
34 {
35   if (page->present)
36     return;
37   else
38   {
39     uint32_t idx = first_frame ();
40     if (idx == -1)
```

```
41      {
42        kprint_debug ("Fail in alloc_page function: no frames available.
      Halting...", LIGHT_GREEN);
43        asm volatile ("cli");
44        asm volatile ("hlt");
45      }
46      set_frame (idx);
47      page->present = 1;
48      page->rw = is_writeable;
49      page->user = (1 + is_kernel) % 2;
50      page->frame = idx;
51    }
52  }
53  void free_page (page_table_entry_t *page)
54  {
55    if (!page->present)
56      return;
57    else
58    {
59      clear_frame (page->frame/0x1000);
60      page->frame = 0x0;
61      page->present = 0x0;
62    }
63  }
64  void paging_init () {
65    uint32_t mem_end_addr = 0x1000000;
66    nframes = mem_end_addr / 0x1000;
67    bitmap = (uint8_t*) kmalloc (INDEX_FROM_BIT (nframes));
68    memset ((uint8_t*) bitmap, 0, INDEX_FROM_BIT (nframes));
69    kernel_directory = (page_directory_t*) kmalloc (sizeof (
      page_directory_t));
70    current_directory = kernel_directory;
71    int i = 0;
72    while (i <= HEAP_BASE)
73    {
74      alloc_page (get_page (i, 1, kernel_directory), 0, 0);
75      i += 0x1000;
76    }
77    register_interrupt_handler (14, page_fault_handler);
78    enable_paging ();
79  }
80  static void enable_paging ()
81  {
82    asm volatile ("mov %0, %%cr3" :: "r" (&kernel_directory->
      tables_physical));
83    uint32_t cr0;
84    asm volatile ("mov %%cr0, %0" : "=r" (cr0));
```

```
85    cr0 |= 0x80000000;
86    asm volatile ("mov %0, %%cr0" :: "r" (cr0));
87    page_enabled = 1;
88  }
89  void switch_page_directory (page_directory_t *dir, uint8_t phy)
90  {
91    page_directory_t* tmp;
92    if (!phy)
93      tmp = (page_directory_t*) virtual2phys (current_directory, (uint32_t
       )dir);
94    else
95      tmp = dir;
96    current_directory = tmp;
97    asm volatile ("mov %0, %%cr3" :: "r" (&tmp->tables_physical));
98  }
99  page_table_entry_t* get_page (uint32_t address, int make,
       page_directory_t *dir)
100 {
101   address /= 0x1000;
102   uint32_t table_idx = address / 1024;
103   if (dir->tables[table_idx])
104     return &dir->tables[table_idx]->pages[address % 1024];
105   else if(make)
106   {
107     uint32_t tmp;
108     dir->tables[table_idx] = (page_table_t*) kmalloc (sizeof (
       page_table_t));
109     tmp = virtual2phys (dir, (uint32_t) dir->tables[table_idx]);
110     dir->tables_physical[table_idx].frame = tmp >> 12;
111     dir->tables_physical[table_idx].present = 1;
112     dir->tables_physical[table_idx].rw = 1;
113     dir->tables_physical[table_idx].user = 1;
114     dir->tables_physical[table_idx].page_size = 0;
115     return &dir->tables[table_idx]->pages[address%1024];
116   }
117   else
118     return 0;
119 }
120 uint32_t virtual2phys (page_directory_t *dir, uint32_t virt_addr)
121 {
122   if (!page_enabled)
123     return virt_addr;
124   page_table_entry_t* page;
125   if (!(page = get_page (virt_addr, 0, dir)))
126     return 0;
127   if (!page->present)
128     return 0;
```

```
129   return ((page->frame << 12) & 0xfff00000) | (virt_addr & 0xfffff);
130 }
131 static void page_fault_handler (registers_t* regs)
132 {
133   kprint_debug ("FECHEM OS PORTOOOOOOOOES!! ESTAMOS SENDO
       ATACAAAAAAAAADOS!!!\n", LIGHT_GREEN);
134   uint32_t faulting_addr;
135   asm volatile ("mov %%cr2, %0" : "=r" (faulting_addr));
136   char str[10];
137   itoa (faulting_addr, str);
138   kprint_debug ("falting addr: ", LIGHT_GREEN);
139   kprint_debug (str, LIGHT_GREEN);
140   kprintf (": %x\n", 1, faulting_addr);
141   uint8_t present = regs->err_code & 0x1;
142   uint8_t rw = regs->err_code & 0x2;
143   uint8_t user = regs->err_code & 0x4;
144   if (!present) kprint_debug ("page not present\n", LIGHT_GREEN);
145   if (rw) kprint_debug ("page is read only\n", LIGHT_GREEN);
146   if (user) kprint_debug ("page is read only\n", LIGHT_GREEN);
147   // kprint_debug ("halting...", LIGHT_GREEN);
148   //asm volatile ("cli");
149   //asm volatile ("hlt");
150   static int t = 0;
151   t++;
152   if (t == 3)
153     alloc_page (get_page (faulting_addr, 1, kernel_directory), 0, 0);
154 }
```

O Código 4.40, inicialmente, apresenta dois *defines* simples que são utilizados no mapeamento das páginas nos quadros. As *structs* declaradas são utilizadas no gerenciamento da memória virtual. A MMU do processador Intel 80386 utiliza uma tabela de página de dois níveis: para proporcionar uma memória virtual de 4GB, por exemplo, uma tabela de um nível necessita de 4MB de memória para armazenar as *structs* de controle. Para contornar este problema, a Intel implementou a arquitetura de dois níveis da tabela de páginas, onde um diretório de páginas contém 1024 entradas, cada uma contendo o endereço de uma tabela de páginas, que também possui 1024 entradas. Desta maneira, as tabelas de páginas são alocadas somente quando necessário, e podem ser desalocadas, pois possuem bits de controle assim como as páginas.

Algumas *structs* neste código foram declaradas utilizando *bit fields*. O *bit field* possibilita a indexação de bits específicos em um grupo necessário de bytes. Quando *bit fields* são utilizados, o tipo de dados que possui a maior quantidade de bytes na *struct* é utilizado como base para a alocação de memória. Cada variável presente na *struct* referencia um ou vários bits, de forma que quando a quantidade de bits referenciados alcançar a quantidade de bits existente no tipo utilizado como base, um segundo espaço de

memória de mesmo tamanho será alocado. Uma *struct* do tipo page_dir_entry_t ocupará somente 4 bytes de memória, pois o tipo uint32_t é capaz de armazenar todos os bits referenciados. Essa *struct* é responsável por descrever cada entrada do diretório de páginas. Cada entrada é composta pelos seguintes componentes:

- **present:** Bit que indica se a tabela de páginas referenciada por essa entrara está na memória física no momento;
- **rw:** Bit que indica se a tabela de páginas pode ser lida e escrita (1), ou se ela pode ser somente lida (0). Vale ressaltar que o bit WP no registrador CR0 indica se a restrição imposta pela *flag* RW é aplicável somente para códigos executando em modo usuário ou para códigos executando em modo usuário e modo *kernel*;
- **user:** Bit que controla o acesso à tabela de páginas baseado no nível de privilégio em que o código está sendo executado;
- **write-through:** Bit que indica se o método utilizado para *caching* será o *write-through* (1), ou *write-back* (0);
- **cache:** Bit que indica se a cache será utilizada para armazenar a tabela de página (0), ou se a tabela não passará pela cache (1);
- **accessed:** Bit que indica se a tabela de páginas foi acessada. Esse bit é automaticamente ativado pela CPU, mas não é limpo por ele, de modo que este trabalho é realizado pelo sistema operacional, caso necessário;
- **page size:** Bit que indica se a página possui tamanho de 4MB (1), ou 4KB (0);
- **frame:** Campo que armazena o endereço físico de onde a tabela está localizada.

A *struct* page_table_entry_t define o formato da entrada da tabela de páginas. Cada entrada desta tabela está relacionada à uma página específica, e seu formato é parecido com as entradas do diretório de páginas:

- **present:** Bit que indica se a página referenciada por essa entrara está na memória física no momento;
- **rw:** Bit que indica se a página pode ser lida e escrita (1), ou se ela pode ser somente lida (0). Vale ressaltar que o bit WP no registrador CR0 indica se a restrição importa pela *flag* RW é aplicável somente para códigos executando em modo usuário ou para códigos executando em modo usuário e modo *kernel*;
- **user:** Bit que controla o acesso à página baseado no nível de privilégio em que o código está sendo executado;

- **accessed:** Bit que indica se a de página foi acessada. Esse bit é automaticamente ativado pela CPU, mas não é limpo por ele, de modo que este trabalho é realizado pelo sistema operacional, caso necessário;
- **dirty:** Bit que indica se a de página foi escrita. Esse bit é automaticamente ativado pela CPU, mas não é limpo por ele, de modo que este trabalho é realizado pelo sistema operacional, caso necessário;
- **frame:** Campo que armazena o endereço físico de início do *frame*.

Vale ressaltar que os bits RW e USER são afetados pelos seus respectivos no diretório de páginas, de maneira que para uma página ser acessível por um código que executa em modo usuário, por exemplo, o bit USER deve ser ativo tanto na entrada da tabela de páginas quanto na página.

A *struct* page_table_t é responsável por descrever a tabela de páginas, que possui somente um vetor de 1024 posições, cada posição contendo uma entrada da tabela de páginas. A *struct* page_directory_t é responsável por descrever o diretório de páginas. Teoricamente, o diretório de páginas deveria conter somente o vetor de entradas do diretório de páginas, contudo, após a habilitação da paginação, todos os endereços gerados pelo código serão virtuais, portanto, o segundo vetor é utilizado pelo núcleo para acessar as páginas após a habilitação da memória virtual.

Dois ponteiros para diretórios de páginas são declarados, o kernel_directory, que representará o diretório do núcleo do sistema, e o current_directory, que representará o diretório de página atual. Neste trabalho, só existirá um diretório de páginas, tendo em vista que existe somente um processo executando em apenas um nível de privilégio. Contudo, na versão que permitirá a execução de diversos processos em diferentes níveis de privilégio, cada processo deve ter um diretório próprio de páginas.

O Código 4.41 é responsável por implementar as funções que realizarão a inicialização e o gerenciamento da memória virtual. O controle de *frames* livres e ocupados, além do método de alocação são iguais aos utilizados pela *heap*: um mapa de bits é utilizado para mapear os *frames* livres e ocupados, e o método *first fit* indicará o *frame* a ser utilizado. A variável bitmap é utilizada como um ponteiro para o primeiro byte do mapa de bits, e a variável nframes armazena a quantidade de *frames* existentes na memória física. As funções set_frame, clear_frame e first_frame atuam no mapa de bits, sendo responsáveis por tornar um *frame* ocupado no mapa de bits, tornar um *frame* liberado no mapa de bits e retornar o endereço do primeiro *frame* disponível, respectivamente.

A função alloc_page é responsável por associar uma página à um *frame*, recebendo a página por parâmetro, assim como os bits de controle de acesso. A função first_frame retorna o endereço do *frame* que será associado à uma página. Este endereço é utilizado

juntamente com os bits de acesso para preencher as entradas necessárias na tabela de página. A função free_page é responsável por limpar o bit *present* da página e tornar o *frame* que estava associado à página livre.

A função page_init é responsável por realizar o procedimento de habilitação da paginação. Assumindo que a quantidade de memória disponível seja de 16MB (0x1000000 bytes), a quantidade de *frames* foi encontrada através da divisão do tamanho da memória pelo tamanho da página. Logo em seguida, o mapa de bits é limpo, indicando que nenhum *frame* foi utilizado. O diretório de páginas do núcleo é alocado na *heap* e em seguida, utilizado para mapear todos os endereços de memória utilizados até o momento. O mapeamento se inicia no endereço 0x0 e se estende, em unidades de 4096, até a base da *heap*, que é o maior endereço utilizado até o momento. Este mapeamento é realizado pois, após a habilitação da paginação, os endereços nesse espaço podem ser acessados como se a paginação não estivesse habilitada, visto que os endereços virtuais serão iguais aos endereços físicos. A função utilizada para o tratamento da interrupção *page fault* é registrada na exceção de número 14. A função que habilita a paginação carrega o diretório de páginas de núcleo no registrador CR3 e altera o bit mais significativo do registrador CR0. Vale ressaltar que a estrutura armazenada no registrador CR3 é o vetor que contém as entradas do diretório de páginas, e não o vetor que contém os endereços virtuais das tabelas de páginas.

A função switch_page_directory é responsável por alterar o diretório de páginas atual. Esta mudança é necessária quando existe a troca de contexto entre processos no espaço de usuário ou entre um processo e o núcleo. Para realizar a troca, o ponteiro current_directory recebe o endereço físico do novo diretório, assim como o registrador CR3.

A função get_page retorna a página que contém o endereço virtual recebido por parâmetro. Caso a página não exista, e o parâmetro make for 1, a nova tabela de páginas que contem a página requerida será alocada. A função virtual2phys é responsável por retornar o endereço físico referente ao endereço virtual recebido por parâmetro. Essa tradução ocorre a partir do acesso ao campo *frame* presente na *struct* que define a página.

A função page_fault_handler é responsável por tratar a interrupção *page fault*. Essa interrupção pode ser lançada a partir de diversos cenários, portanto, o código de erro empilhado pela CPU apresenta o motivo da interrupção, a partir da combinação de seus bits. O endereço acessado que gerou a *page fault* é armazenado no registrador CR2.

Para realizar o teste deste módulo, o código de entrada do núcleo, apresentado no Código 4.42, acessa o último endereço mapeado na função paging_init (0x171FFF), e em seguida acessa o próximo endereço (0x172000). Ao tentar acessar um endereço que não está presente na memória virtual, o processador gera a interrupção *page fault*, executando a função page_fault_handler. Esta função não resolve a falta gerada imediatamente,

para que seja possível demonstrar que após o termino de uma função de tratamento de interrupção, a operação que a gerou é executada novamente. Através de uma variável estática, o código de tratamento da falta de página permite a execução da instrução problemática 3 vezes, até que a página do endereço requisitado seja alocada. A Figura 4.14 apresenta o resultado da execução deste módulo.

Código 4.42 – Função principal do núcleo para o teste do mecanismo de gerenciamento de memória virtual utilizando paginação

```
#include <stdint.h>
#include <vga.h>
#include <isr.h>
#include <keyboard.h>
#include <kheap.h>
#include <paging.h>
#include <pit.h>
void entry ()
{
  clear_screen ();
  kprint ("Welcome to the OS!\n\0");
  isr_install ();
  asm volatile ("sti");
  pit_init (PIT_INPUT_FREQUENCY);
  keyboard_init();
  kheap_init ();
  paging_init ();
  uint8_t *ptr = (uint8_t *)  0x171fff ;
  uint32_t value = (uint32_t ) *ptr ;
  kprint ("0x171fff - ok!\n");
  ptr = (uint8_t *) 0x172000 ;
  value = (uint32_t) *ptr ;
  kprint ("0x172000 - ok!\n") ;

}
```

4.11 Módulo 9: Implementação de um mecanismo de acesso à dispositivos PCI

Este módulo implementa uma forma de comunicação com os dispositivos conectados ao PCI através de uma interface composta pelas seguintes funções: pci_read_data; pci_write_data; pci_gen_address; pci_get_device. Os arquivos *include/pci.h* e *drivers/pci.c* são responsáveis por implementar este interface, e contém, respectivamente, o Código 4.43 e o Código 4.44.

Código 4.43 – Cabeçalho para o mecanismo de acesso à dispositivos PCI

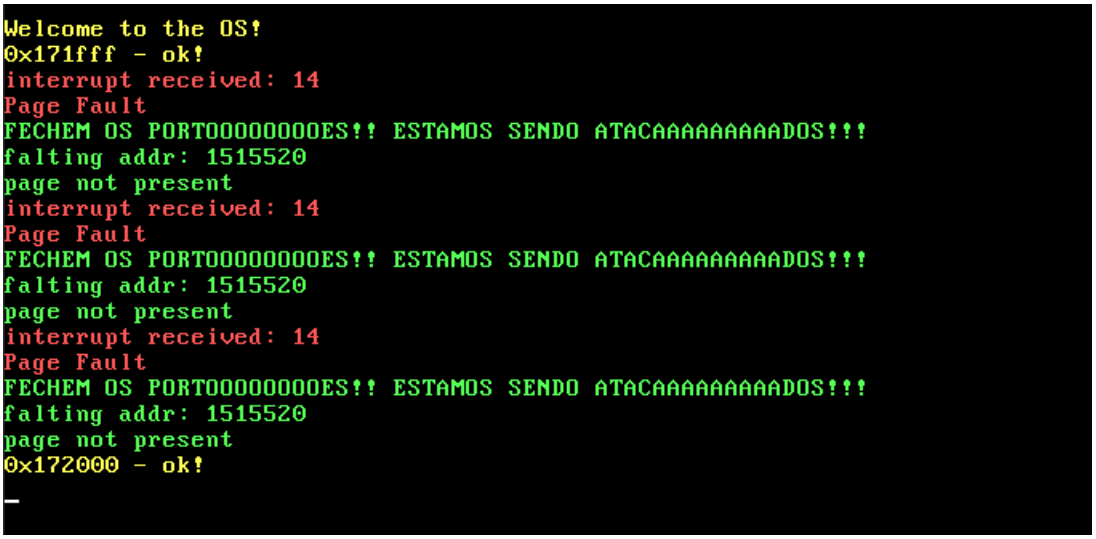


Figura 4.14 – Teste do mecanismo de gerenciamento de memória virtual utilizando paginação

```
#ifndef PCI_H
#define PCI_H
#define PCI_CONFIG_ADDR 0xcf8
#define PCI_CONFIG_DATA 0xcfc
#include <ports.h>
#include <vga.h>
#include <string.h>
uint32_t pci_read_data (uint8_t bus, uint8_t device, uint8_t offset);
void pci_write_data (uint8_t bus, uint8_t device, uint8_t offset,
   uint32_t data);
uint32_t pci_gen_address (uint8_t bus, uint8_t device, uint8_t offset);
void pci_get_device (uint16_t vendor_id, uint16_t device_id, uint16_t *
   bus, uint8_t *device);
void pci_brute ();
#endif
```

Código 4.44 – Mecanismo de acesso à dispositivos PCI

```
#include <pci.h>
uint32_t pci_read_data (uint8_t bus, uint8_t device, uint8_t offset)
{
  uint32_t address = pci_gen_address (bus, device, offset);
  port_dword_out ((uint16_t) PCI_CONFIG_ADDR, address);
  uint32_t tmp = (uint32_t) (port_dword_in ((int16_t) PCI_CONFIG_DATA));
  return tmp;
}
void pci_write_data (uint8_t bus, uint8_t device, uint8_t offset,
   uint32_t data)
{
  uint32_t address = pci_gen_address (bus, device, offset);
```

```
12   port_dword_out ((uint16_t)PCI_CONFIG_ADDR, address);
13   port_dword_out ((uint16_t)PCI_CONFIG_DATA, data);
14 }
15 uint32_t pci_gen_address (uint8_t bus, uint8_t device, uint8_t offset)
16 {
17   uint32_t address;
18   uint32_t lbus  = (uint32_t)bus;
19   uint32_t ldevice = (uint32_t)device;
20   uint32_t lfunc = 0;
21   uint32_t tmp = 0;
22   address = (uint32_t)((lbus << 16) | (ldevice << 11) |
23       (lfunc << 8) | (offset & 0xfc) | ((uint32_t)0x80000000));
24   return address;
25 }
26 void pci_get_device (uint16_t vendor_id, uint16_t device_id, uint16_t *
     bus, uint8_t *device)
27 {
28   int i, j;
29   for (i = 0; i < 256; i++)
30   {
31     for (j = 0; j < 32; j++)
32     {
33       uint32_t data = pci_read_data (i, j, 0);
34       uint32_t reg = (device_id << 16) | vendor_id;
35       if (data == reg)
36       {
37         bus = i;
38         device = j;
39         kprintf ("device found\nbus: %d\ndev: %d\n", 2, bus, device);
40         kprintf ("offset 0x8: %x\n", 1, pci_read_data (bus, device, 0x8)
     );
41
42         return;
43       }
44     }
45   }
46   kprint ("no device found\n");
47
48 }
```

Um dispositivo PCI possui um endereço composto por 3 componentes: *bus*, que define o barramento que o dispositivo está localizado; *device*, que indica o dispositivo dentro do barramento; *funcion*, utilizado quando um dispositivo exerce mais de uma função. Para acessar o *Configuration Address* de um determinado dispositivo, a porta de endereço 0xCF8 é preenchida com os componentes do endereço PCI, juntamente com o deslocamento do registrador desejado. O processo de criação deste endereço é realizado pela função

pci_gen_address. As funções pci_read_data e pci_write_data são responsáveis por ler e escrever valores do *Configuration Address* a partir dos componentes de endereços PCI. Os dados que serão lidos ou escritos são acessados através da porta 0xCFC, denominada CONFIG_DATA. A função get_pci_device é responsável por retornar o *bus* e o número do dispositivo por meio do *device id* e do *vendor id*, que são disponibilizadas pelo fabricante do dispositivo e estão presentes no cabeçalho do *Configuration Space*. O retorno destes valores são realizados através de endereços de variáveis recebidos como parâmetro.

Para realizar o teste deste módulo, algumas impressões foram adicionadas no código da função pci_get_device, de forma que quando o dispositivo for encontrado, campos *Class code*, *Subclass*, *Prog IF* e *Revision ID* são impressos. A placa de rede rtl8139 foi adicionada ao barramento PCI provido pelo QEMU a partir da *flag* -device rtl8139 no comando de execução deste. Este dispositivo possui o Vendor ID 0x10EC, e o Device ID 0x8139. A Figura 4.15a apresenta a execução do sistema utilizando o seguinte comando de execução do QEMU: qemu-system-i386 -fda os-image.bin. A Figura 4.15b apresenta a execução do sistema utilizando o seguinte comando de execução do QEMU: qemu-system-i386 -fda os-image.bin -device rtl8139. Analisando o teste apresentado na Figura 4.15b, pode-se percebe que o campo *Class code* possui o valor 0x2, que indica que aquele disposivido PCI é um controlador de rede. O campo *Subclass* possui valor 0x0, indicando que o controlador de rede é um controlador Ethernet. Os dois campos seguintes não são utilizados para o controlador de rede.

(a)

(b)

Figura 4.15 – Teste do mecanismo de acesso à dispositivos PCI

4.12 Módulo 10: Implementação de um mecanismo de gerenciamento de multitarefas

O sistema *multitasking* implementado neste módulo foi baseado em Brendan (2019) e consiste em um gerenciador de tarefas que as permite compartilhar o processador,

de forma que vários trabalhos isolados uns dos outros possam existir simultaneamente dentro do ambiente do sistema operacional. Além da gerência, alguns processos especiais foram implementados para melhorar o desempenho do sistema e para realizar atividades relacionadas ao sistema operacional em si.

O sistema de gerenciamento de tarefas possui as seguintes funções: multitask_init; print_task; unblock_task; block_task; sleep; task_entry; task_termination; scheduler; lock_irq; unlock_irq. Essas funções, juntamente à declaração de variáveis e estruturas utilizadas na codificação estão expostas no arquivo *include/multitask.h*, formando a interface deste módulo, apresentada no Código 4.45. O código responsável por implementar as funcionalidades apresentadas neste arquivo de cabeçalho foi escrito no arquivo *multitask/multitask.c*, apresentado no Código 4.46.

Código 4.45 – Cabeçalho para o mecanismo de gerenciamento de tarefas

```
1  #ifndef MULTITASK_H
2  #define MULTITASK_H
3
4  #include <stdint.h>
5  #include <paging.h>
6  #include <vga.h>
7  #include <mem.h>
8  #include <pit.h>
9  #include <idle_task.h>
10 #include <task_terminator.h>
11 #define BLOCKED 0
12 #define RUNNING 1
13 #define READY_TO_RUN 2
14 #define SLEEPING 3
15 #define IDLE 4
16 #define TERMINATED 5
17 struct tcb
18 {
19   uint32_t *esp;
20   uint32_t *ebp;
21   struct page_directory_t* page_dir;
22   struct tcb* next_task;
23   uint32_t pid;
24   char pname [32];
25   uint8_t state;
26   uint32_t sleep_until;
27 }__attribute__ ((packed));
28 struct tcb* create_task (uint8_t (*func) (void), char *pname, uint8_t
       state);
29 struct tcb* search_task (uint32_t pid);
30 void multitask_init ();
31 void print_task (struct tcb*);
```

```
32 void unblock_task (uint32_t pid);
33 void block_task (uint8_t reason);
34 void sleep (uint32_t seconds);
35 void task_entry ();
36 void task_termination ();
37 void scheduler () ;
38 void lock_irq ();
39 void unlock_irq ();
40 struct tcb* current_task;
41 struct tcb* idle_task;
42 struct tcb* head;
43 uint32_t multitasking_on;
44 uint32_t ready_to_run_counter;
45 #endif
```

Código 4.46 – Mecanismo de gerenciamento de tarefas

```
1 #include <multitask.h>
2 static struct tcb* _create_task (struct page_directory_t *page_dir,
    struct tcb *next_task, uint32_t pid, uint8_t state, char *pname,
    uint8_t (*func) (void));
3 static void sleep_until (uint32_t ticks);
4 uint32_t last_pid = 1;
5 uint32_t lock_irq_counter = 0;
6 static struct tcb* _create_task (struct page_directory_t *page_dir,
    struct tcb *next_task, uint32_t pid, uint8_t state, char *pname,
    uint8_t (*func) (void))
7 {
8   struct tcb *new_task = (struct tcb*) kmalloc (0x1000);
9   new_task->ebp = ((uint32_t)new_task + 0x1000) - sizeof (struct tcb);
10  new_task->esp = new_task->ebp - 4;
11  new_task->page_dir = page_dir;
12  new_task->next_task = next_task;
13  new_task->pid = pid;
14  new_task->state = state;
15  if (state == READY_TO_RUN)
16    ++ready_to_run_counter;
17  memcpy (pname, new_task->pname, 32);
18  *new_task->esp = new_task->ebp;
19  *(new_task->esp + 1) = 0;
20  *(new_task->esp + 2) = 0;
21  *(new_task->esp + 3) = 0;
22  *(new_task->esp + 4) = func;
23  return new_task;
24 }
25 struct tcb* create_task (uint8_t (*func) (void), char *pname, uint8_t
    state)
26 {
```

```
27   if (state == IDLE)
28     return idle_task =  _create_task (kernel_directory, 0, 0, state,
     pname, func);
29   return head = _create_task (kernel_directory, head, ++last_pid, state,
       pname, func);
30 }
31 void multitask_init ()
32 {
33   head = (struct tcb *) kmalloc (0x1000);
34   current_task = head;
35   asm volatile ("mov %%esp, %0" : "=r" (head->esp));
36   head->page_dir = current_directory;
37   head->next_task = 0;
38   head->pid = 1;
39   head->state = RUNNING;
40   char *s = "JAMES";
41   memcpy (s, head->pname, 5);
42   create_task (idle_task_function, "IDLE", IDLE);
43   create_task (task_terminator, "CHUAZNEGUER", READY_TO_RUN);
44   ++ready_to_run_counter;
45   multitasking_on = 1;
46 }
47 void block_task (uint8_t reason)
48 {
49   lock_irq ();
50   --ready_to_run_counter;
51   current_task->state = reason;
52   scheduler ();
53   unlock_irq ();
54 }
55 void unblock_task (uint32_t pid)
56 {
57   lock_irq ();
58   ++ready_to_run_counter;
59   struct tcb *tmp;
60   tmp = search_task (pid);
61   current_task->state = READY_TO_RUN;
62   tmp->state = RUNNING;
63   task_switch (tmp);
64   unlock_irq ();
65 }
66 void lock_irq ()
67 {
68   asm volatile ("cli");
69   ++lock_irq_counter;
70 }
71 void unlock_irq ()
```

```
72 {
73   if (--lock_irq_counter == 0)
74     asm volatile ("sti");
75 }
76 void sleep (uint32_t seconds)
77 {
78   sleep_until ((seconds * 100) + tick);
79 }
80 static void sleep_until (uint32_t ticks)
81 {
82   if (ticks < tick)
83     return;
84   current_task->sleep_until = ticks;
85   block_task (SLEEPING);
86 }
87 void scheduler ()
88 {
89   struct tcb* next;
90   if (!ready_to_run_counter)
91   {
92     task_switch (idle_task);
93     return;
94   }
95   if (current_task->state == RUNNING)
96     current_task->state = READY_TO_RUN;
97   next = current_task->next_task;
98   while (1)
99   {
100    if (!next)
101      next = head;
102    if (next->state != READY_TO_RUN)
103      next = next->next_task;
104    else
105      break;
106  }
107  next->state = RUNNING;
108  task_switch (next);
109 }
110 void task_termination ()
111 {
112  lock_irq ();
113  kprintf ("[%s] terminated.\n", 1, current_task->pname);
114  --ready_to_run_counter;
115  current_task->state = TERMINATED;
116  scheduler ();
117 }
118 void task_entry ()
```

```
119 {
120   kprintf ("[%s] started.\n", 1, current_task->pname);
121   unlock_irq ();
122 }
123 void print_task (struct tcb* tcb)
124 {
125   if (!tcb)
126     tcb = current_task;
127   kprint ("---task info---\n");
128   kprintf ("pname: %s\n", 1, tcb->pname);
129   kprintf ("pid: %d\n", 1, tcb->pid);
130   kprintf ("esp: %x\n", 1, tcb->esp);
131   kprintf ("next_task: %x\n", 1, tcb->next_task);
132   kprintf ("page_directory: %x\n", 1, tcb->page_dir);
133   kprintf ("current state: %d\n", 1, tcb->state);
134 }
135 struct tcb* search_task (uint32_t pid)
136 {
137   struct tcb* it;
138   for (it = head; it != 0; it = it->next_task)
139     if (it->pid == pid)
140       return it;
141   return 0;
142 }
```

A *struct* tcb (*task control block*) é responsável por armazenar as informações de controle da *task*, permitindo que esta seja suspensa em algum momento e retorne do ponto que estava. Parte do contexto é armazenado na pilha da *task*, necessitando a existência das variáveis esp e ebp no bloco de controle, que apontam para o topo e para a base da pilha, respectivamente. A variável page_dir armazena o endereço do diretório de páginas da *task*, visto que cada processo de usuário possui seu próprio diretório de páginas. O sistema de gerenciamento armazena as estruturas de controle das *tasks* como uma lista encadeada, de modo que a variável next_task aponta para a estrutura de controle da próxima *task*. A identificação do processo pode ser feita utilizando a variável *pid*, que é única entre as *tasks*, e pelo nome, armazenado na variável pname, que pode se repetir. A variável state é responsável por armazenar o estado atual do processo, podendo receber os seguintes valores: BLOCKED, que indica ao *scheduler* que esta *task* não deve ser selecionada para execução; RUNNING, que indica que a *task* está sendo executada neste momento; READY_TO_RUN, que indica ao *scheduler* que a *task* está pronta para executar; SLEEPING, que indica ao *scheduler* que esta *task* não deve ser selecionada para a execução, mas será desbloqueada quando o *timer* atingir determinado valor; IDLE, que é o estado constante da *task* especial que é executada quando não existe nenhuma *task* no estado READY_TO_RUN; TERMINATED, que indica que a *task* acabou sua execução e pode ter suas estruturas desalocadas da memória.

A variável current_task aponta para o *tcb* da *task* que está sendo executada no memento. A variável idle_task aponta para o *tcb* da *task* especial que é executada quando não existe nenhuma *task* pronta para ser executada. A variável head aponta para o início da lista encadeada de estruturas de controle de *tasks*. A variável multitask_on indica se o módulo de gerenciamento de *tasks* está habilitado. A variável ready_to_run_counter é responsável por armazenar a quantidade de *tasks* no estado READY_TO_RUN.

A função create_task, responsável pela criação de uma *task* de núcleo, recebe como parâmetro o endereço da primeira função que *task* irá executar, o nome da *task* e o estado inicial. Essa função é simplesmente uma interface, pois ela chama a função _create_task, que recebe todos os parâmetros necessários para a sua criação, que são o ponteiro para o diretório de páginas, o ponteiro de início para a lista encadeadas de blocos de controle de *tasks*, o *pid* da *task*, o estado inicial, o nome e o endereço da função inicial. A função inicial deve, obrigatoriamente, retornar um valor de oito bits, referente ao status de termino da *task*. A função _create_task aloca o equivalente a uma página para armazenar o *tcb* da nova *task* e a pilha desta. Logo após, os elementos do *tcb* são preenchidos, e a pilha é inicializada com os valores que representam o contexto. O contexto é composto pelos valores dos registradores EBP, EDI, ESI, EBX e pelo valor de retorno. O valor de retorno é utilizado para a primeira execução da *task* e para a retomada do processamento, caso ela seja suspensa. Vale ressaltar que a nova *task* é inserida no início da lista encadeada, ou seja, a variável *head* passa a apontar para a nova *task*, e esta aponta para a *task* que estava sendo apontada pela variável *head*.

A função multitask_init é responsável por inicializar o sistema de gerenciamento de tarefas. O processamento atual é mapeado em uma *task*, de forma que sua estrutura de controle é criada, povoada, e adicionada à lista encadeada de tarefas. Vale ressaltar que a posição da pilha desta *task* não é preenchida na estrutura, pois essa ação é realizada no momento da troca de contexto. A função multitask_init também é responsável por criar as duas *tasks* especiais: *idle_task* e *task_terminator*.

A *task idle_task* é responsável por parar o processador quando não existe nenhuma outra *task* pronta para executar. A função de entrada para esta *task* é a idle_task_function, que foi implementada nos arquivos *include/idle_task.h* e *kernel_tasks/idle_task.c*, contendo o Código 4.47 e o Código 4.48, respectivamente. A instrução hlt para o processador até que uma interrupção habilitada, uma exceção de debug, o sinal BINIT#, o sinal INIT#, ou o sinal RESET# ocorra, permitindo que o processador salve energia.

Código 4.47 – Cabeçalho para a tarefa *idle_task*

```
1 #ifndef IDLE_TASK_H
2 #define IDLE_TASK_H
3 #include <stdint.h>
4 #include <multitask.h>
```

```
5 uint8_t idle_task_function ();
6 #endif
```

Código 4.48 – Tarefa *idle_task*

```
1 #include <idle_task.h>
2 uint8_t idle_task_function ()
3 {
4   task_entry ();
5   while (1)
6   {
7     kprintf ("[%s] executing...\n", 1, current_task->pname);
8     asm volatile ("hlt");
9     uint32_t i;
10    for (i = 0; i < 1e9; i++);
11  }
12 }
```

A *task task_terminator* é responsável por desalocar as estruturas de *tasks* no estado TERMINATED e organizar a lista encadeada de processos. O código desta *task* foi escrito nos arquivos *include/task_terminator* e *kernel_tasks/task_terminator*, contendo o Código 4.49 e o Código 4.50, respectivamente.

Código 4.49 – Cabeçalho para a tarefa *task_terminator*

```
1 #ifndef TASK_TERMINATOR_H
2 #define TASK_TERMINATOR_H
3 #include <multitask.h>
4 #include <kheap.h>
5 void task_terminator ();
6 #endif
```

Código 4.50 – Tarefa *task_terminator*

```
1 #include <task_terminator.h>
2 void task_terminator ()
3 {
4   task_entry ();
5   struct tcb* tmp;
6   struct tcb* prev;
7   while (1)
8   {
9     uint8_t flag = 0;
10    tmp = head;
11    while (tmp)
12    {
13      if (tmp->state == TERMINATED)
14      {
```

```
15          kprintf ("[%s] exterminating %s...\n",2, current_task->pname,
    tmp->pname);
16          flag = 1;
17          if (tmp == head)
18            head = tmp->next_task;
19          else
20            prev->next_task = tmp->next_task;
21          kfree (0x1000, tmp);
22          kprintf ("[%s] done.\n", 1, current_task->pname);
23          break;
24        }
25        prev = tmp;
26        tmp = tmp->next_task;
27      }
28      if (!flag)
29        sleep (10);
30    }
31 }
```

Esta tarefa itera pela lista de *tasks* a partir da *task* apontada pela variável head. Caso seja encontrada uma *task* em estado TERMINATED, a pilha e o bloco de controle desta são liberadas, e os ponteiros da lista encadeada são adaptados. Após a liberação da memória de todas as *tasks* finalizadas, a *task_terminator* é bloqueada por 10 segundos para que não exista sobrecarga com sua execução.

A função block_task é responsável por bloquear uma *task*, modificando seu estado para SLEEPING ou BLOCKED, e o *scheduler* é chamado para que uma nova tarefa seja executada. A função unblock_task é responsável por tornar uma *task* pronta para executar. Quando esta função é executada, a tarefa desbloqueada é executada imediatamente, movendo a *task* que estava sendo executada para o estado READY_TO_RUN.

As funções lock_irq e unlock_irq funcionam como uma interface que permite a execução de códigos de maneira atômica. Tendo em vista que o ambiente deste projeto consiste na presença de somente um processador, a única forma de interrupção de um processamento é por meio do tratamento de interrupções. Para que um código seja atômico, se faz necessária a utilização da função lock_irq antes do código e da função unlock_irq depois do código, permitindo que este seja executado sem a ocorrência nenhuma interrupção. O contador presente nessas funções permite a chamada de funções aninhadas que bloqueiam as interrupções, de maneira que quando a função mais interna terminar de executar, mesmo se a função unlock_irq for chamada, as interrupções permanecerão desabilitadas para que a função mais externa continue executando de maneira atômica.

A função sleep é responsável por bloquear uma *task* por um determinado tempo. O parâmetro que esta função recebe indica quantos segundos a *task* ficará bloqueada. A função sleep_until é chamada, passando por parâmetro a quantidade de segundos

multiplicada por 100, pois o PIT foi configurado para gerar uma interrupção a cada 10 milissegundos, e somado com a quantidade de ticks gerados até o momento. A função sleep_until modifica o estado da *task* para SLEEPING e escreve no bloco de controle desta em qual *tick* ela deve ser desbloqueada. A função de tratamento do PIT sempre verifica se existe alguma *task* que esteja dormindo e a libera caso tenha atingido o tempo limite. O Código 4.51 mostra o novo código presente no arquivo *drivers/pit.c*.

Código 4.51 – *Driver* do PIT

```
#include <pit.h>
static void pit_callback (registers_t *regs);
static void pit_callback (registers_t *regs)
{
  tick++;
  if (multitasking_on)
  {
    struct tcb* x = head;
    do
    {
      if (x->state == SLEEPING && x->sleep_until < tick)
        unblock_task (x->pid);
      x = x->next_task;
    }
    while (x);
    if (!(tick % 99)) {
      lock_irq ();
      scheduler ();
      unlock_irq ();
    }
  }
}
void pit_init (uint16_t reload_register) {
  register_interrupt_handler(IRQ0, pit_callback);
  uint8_t low  = (uint8_t)(reload_register & 0xFF);
  uint8_t high = (uint8_t)((reload_register >> 8) & 0xFF);
  port_byte_out (PIT_CTRL_PORT, PIT_CTRL_VALUE);
  port_byte_out (PIT_DATA_PORT_CH0, low);
  port_byte_out (PIT_DATA_PORT_CH0, high);
}
```

Além da busca por *tasks* que devem ser movidas do modo SLEEPING para o modo READY_TO_RUN, o novo código do PIT é responsável por chamar a função scheduler periodicamente, que é responsável por definir a próxima *task* a ser executada. Esta função, inicialmente, verifica se existem *tasks* prontas para serem executadas ou se a *idle_task* deve parar o processador. A próxima verificação feita pela função scheduler é o estado da *task* atual, afim de verificar se ela foi bloqueada, terminou sua execução ou deve

voltar a ser escalonada posteriormente. A escolha da próxima *task* a ser executada leva em consideração somente a ordem da lista encadeada, de forma que a próxima *task* a ser executada é a primeira que está no estado READY_TO_RUN, seguindo os ponteiros da lista encadeada a partir do bloco de controle da *task* atual.

A função task_entry deve ser executada sempre que uma *task* iniciar sua execução, assim como a função task_termination deve ser executada por uma *task* sempre que esta terminar sua execução. Sempre que uma *task* inicia uma execução ou volta a executar após uma preempção, ela deve liberar a trava de interrupções através da chamada da função unlock_irq, pois sempre que alguma troca de contexto ocorre, a função lock_irq é chamada. O objetivo da função task_termination é modificar o estado da *task* para TERMINATED, diminuir o contador de tarefas no estado READY_TO_RUN e chamar o *scheduler* para que outra *task* entre em execução.

A função print_task é utilizada para debug, pois imprime todas as informações presentes no *tcb* de uma *task*. Caso o parâmetro passado para esta função seja 0, a função imprimirá os dados presentes no *tcb* da tarefa executando no momento. A função *search_task* retorna o *tcb* de uma *task* dado seu pid. Este valor é adquirido a partir da iteração pela lista encadeada de estruturas de controle das tarefas.

Para realizar a troca de contexto propriamente dita, a função task_switch, escrita em Assembly, foi implementada no arquivo *multitask/task_switch.asm*, que contém o Código 4.52

Código 4.52 – Troca de contexto

```
1  [bits 32]
2  extern current_task
3  struc TCB
4    .ESP resd 1
5    .EBP resd 1
6    .PAGE_DIR resd 1
7    .NEXT_TASK resd 1
8    .PID resd 1
9    .PNAME resb 32
10   .STATE resb 1
11   .SLEEP_UNTIL resd 1
12 endstruc
13 global task_switch
14 task_switch:
15   push ebx
16   push esi
17   push edi
18   push ebp
19   mov edi,[current_task]
20   mov [edi+TCB.ESP],esp
```

```
21  mov esi,[esp+(4+1)*4]
22  mov [current_task],esi
23  mov esp,[esi+TCB.ESP]
24  mov eax,[esi+TCB.PAGE_DIR]
25  mov ecx,cr3
26  cmp eax,ecx
27  je .doneVAS
28  mov cr3,eax
29  .doneVAS:
30  pop ebp
31  pop edi
32  pop esi
33  pop ebx
34  ret
```

Inicialmente, este código cria uma representação da estrutura tcb a partir da diretiva struc. Cada elemento da estrutura foi definido com seu tamanho correspondente, o que permite a utilização dos deslocamentos de memória, assim como ocorre em códigos C. A função global task_switch é responsável por realizar o processo completo de troca de contexto. Ela recebe como parâmetro o endereço do bloco de controle da próxima *task* a ser executada, que ocupa quatro bytes da pilha. Quando ocorre a chamada para esta função, mais quatro bytes referentes ao endereço de retorno são empilhados. Já dentro do código da função, quatro instruções push são executadas, referentes ao salvamento do contexto da *task* atual.

O endereço do bloco de controle da *task* que está sendo executada no momento é carregado no registrador EDI para que a variável referente ao topo da pilha seja atualizada. O endereço do bloco de controle da próxima *task* a ser executada é carregado no registrador ESI, e este é utilizado para atualização do registrador ESP. A partir deste momento, todas as instruções push, pop e derivadas atuarão na pilha da nova *task*. O endereço do diretório de páginas da nova *task* é armazenado no registrador EAX e uma verificação é executada para apurar se o diretório da *task* a ser executada é igual ao diretório de páginas da *task* anterior. Caso os diretórios sejam iguais, não se faz necessária a troca da referência do registrador CR3. O retorno do contexto da tarefa antiga é resgatado através das instruções pop, seguido da execução da instrução ret. Logo antes da execução desta instrução, o valor presente no topo da pilha é o endereço da próxima instrução que a nova *task* deve executar. Caso a nova *task* não tenha começado sua execução, o valor do topo da pilha é o endereço da função passada por parâmetro para a função create_task, que se encarrega de organizar a pilha inicial.

Para realizar o teste deste módulo, o código de entrada do núcleo foi alterado, como mostra o Código 4.53. Duas novas *tasks* são criadas explicitamente, além da *idle_task*, *task_terminator* e da *task* atual que está executando o código de entrada do núcleo. As

tasks criadas explicitamente executam a função general_task_entry, definida no arquivo *kernel_tasks/general_task.c* que contém o Código 4.54.

Código 4.53 – Função principal do núcleo para o teste do mecanismo de gerenciamento de tarefas

```
#include <stdint.h>
#include <vga.h>
#include <isr.h>
#include <keyboard.h>
#include <kheap.h>
#include <paging.h>
#include <multitask.h>
#include <pit.h>
#include <general_task.h>
void entry ()
{
  clear_screen ();
  kprint ("Welcome to the James OS!\n\0");
  isr_install ();
  asm volatile ("sti");
  pit_init (PIT_INPUT_FREQUENCY);
  keyboard_init();
  kheap_init ();
  paging_init ();
  asm volatile ("cli");
  multitask_init ();
  create_task (general_task_function, "ANDERSON", READY_TO_RUN);
  create_task (general_task_function, "CAROLINA", READY_TO_RUN);
  asm volatile ("sti");
  task_termination ();
}
```

Código 4.54 – Função executada por uma *task* geral

```
#include <general_task.h>
uint8_t general_task_function ()
{
  task_entry ();
  uint8_t i;
  for (i = 0 ; i < 3; i++) {
    kprintf ("[%s] executing...\n", 1, current_task->pname);
    sleep (3);
  }
  task_termination ();
}
```

Neste teste, as duas *tasks* criadas receberão os *pids* 3 e 4, visto que a *idle_task* recebe o *pid* 0, a *task* que está executando o código do núcleo recebe o *pid* 1, e a *task_terminator*

recebe o *pid* 2. Uma iteração de 0 até 1e9 foi adicionada ao final do *loop* infinito do código da *idle_task* para melhor visualização do teste, visto que a cada 10 milissegundos uma interrupção ocorre. A *task* de *pid* 1, denominada *James*, é encerrada logo após a habilitação do sistema *multitask* a partir da execução da função task_termination. As *tasks* de *pid* 3 e 4, que possuem nomes *ANDERSON* e *CAROLINA*, respectivamente, executam o código da função general_task_function, o qual imprime uma string 3 vezes, com o intervalo de 3 segundo entre elas. Após esse *loop*, ambas finalizam sua execução. A Figura 2.6 apresenta o resultado do teste deste módulo.

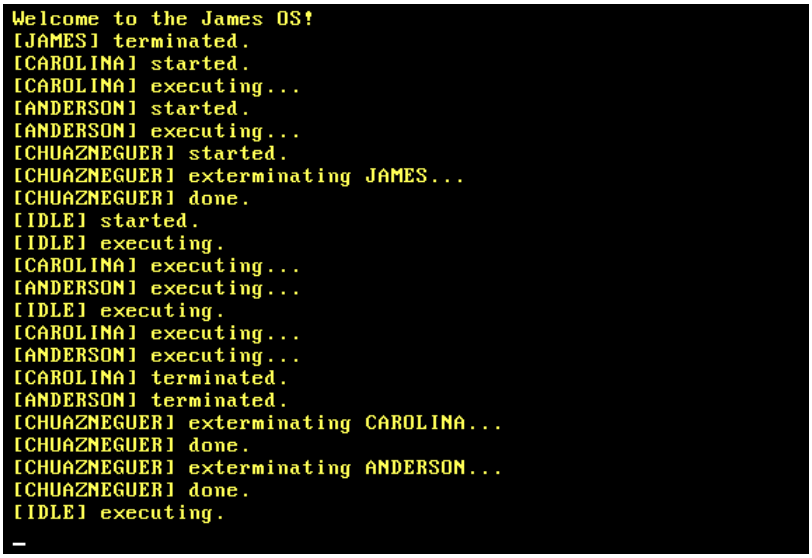


Figura 4.16 – Teste do mecanismo de gerenciamento de tarefas

5 Conclusão

A prática dentro do contexto do estudo de sistemas operacionais é imprescindível, dado que a visualização da implementação de algoritmos e a interação destes com outros componentes possibilitam uma visão abrangente do comportamento do sistema, além de um maior entendimento do seu funcionamento. Para que seja possível proporcionar este cenário, um sistema operacional específico deve prover um ambiente propício para tal objetivo, posto que sistemas que objetivam a usabilidade são extremamente complexos.

Este trabalho apresentou a implementação de um sistema operacional experimental, construído de maneira modular e incremental. A modularidade permitiu o isolamento de cada componente do sistema, de forma que cada módulo pôde ser visto somente a partir da sua interface. A incrementalidade permitiu a esquematização do processo de implementação do sistema operacional, facilitando o acompanhamento do desenvolvimento.

O processo de implementação do sistema operacional foi apresentado em 11 módulos, expondo a interface de cada um, assim como o funcionamento de seus componentes. Durante o processo de construção, pôde-se perceber a dependência de uma forte base teórica, indicando que a prática neste campo possibilita um grande aprendizado, não somente na área de sistemas operacionais, mas também na área de programação, arquitetura de computadores e redes de computadores. O sistema implementado atingiu os requisitos iniciais, os quais se referem à funcionalidade independente em cada módulo, assim como a construção que depende somente de componentes anteriores.

5.1 Trabalhos Futuros

Tendo em vista a vasta abrangência de áreas que o sistema operacional atua, existem diversos módulos que podem ser implementados, afim de ampliar a cobertura do sistema operacional proposto. Entre eles pode-se enumerar:

- implementação da pilha de protocolos TCP/IP, que já está em desenvolvimento;
- implementação de processos que executam em modo usuário;
- modificação do núcleo para uma arquitetura *microkernel*;
- implementação de um *driver* para dispositivos de armazenamento;
- implementação de um *filesystem*;
- implementação de um mecanismo de execução de formatos binários conhecidos, como ELF.

Referências

ANDREWCHEN; THATMADHACKER. *i686-elf-binutils*. 2020. Disponível em: <https://aur.archlinux.org/packages/i686-elf-binutils/>. Acesso em: 10 mai. 2020. Citado na página 41.

ANDREWCHEN; THATMADHACKER. *i686-elf-gcc*. 2020. Disponível em: <https://aur.archlinux.org/packages/i686-elf-gcc/>. Acesso em: 10 mai. 2020. Citado na página 41.

BALASUBRAMANIAN, A. et al. System programming in rust: Beyond safety. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. [S.l.: s.n.], 2017. p. 156–161. Citado na página 41.

BLUNDELL, U. N. *Writing a Simple Operating System — from Scratch*. Birmingham: [s.n.], 2010. Disponível em: <https://www.cs.bham.ac.uk/~exr/lectures/opsys/10_11/lectures/os-dev.pdf>. Citado 4 vezes nas páginas 4, 36, 37 e 38.

BRENDAN. *Brendan's Multi-tasking Tutorial*. 2019. Disponível em: <https://wiki.osdev.org/Brendan%27s_Multi-tasking_Tutorial>. Acesso em: 29 out. 2020. Citado na página 114.

BROKENTHORN. *OS Developmente Series*. 2008. Disponível em: <http://www.brokenthorn.com/Resources/OSDevIndex.html>. Acesso em: 11 nov. 2020. Citado na página 34.

BROWN, R. *The x86 Interrupt List*. 2000. Disponível em: <http://www.cs.cmu.edu/~ralf/files.html>. Acesso em: 08 mar. 2020. Citado na página 15.

CLOUTIER, F. *x86 and amd64 instruction reference*. 2019. Disponível em: <https://www.felixcloutier.com/x86/>. Acesso em: 09 mar. 2020. Citado na página 26.

FENOLLOSA, C. *os-tutorial*. 2018. Disponível em: <https://github.com/cfenollosa/os-tutorial>. Acesso em: 23 abr. 2020. Citado na página 36.

GNU. *GNU GRUB*. 2012. Disponível em: <https://www.gnu.org/software/grub>. Acesso em: 23 abr. 2020. Citado 2 vezes nas páginas 36 e 59.

GNU. *GNU binutils ftp page*. 2020. Disponível em: <https://ftp.gnu.org/gnu/binutils/>. Acesso em: 12 nov. 2020. Citado na página 42.

GNU. *GNU gcc ftp page*. 2020. Disponível em: <https://ftp.gnu.org/gnu/gcc/>. Acesso em: 12 nov. 2020. Citado na página 42.

GNU. *GNU Make*. 2020. Disponível em: <https://www.gnu.org/software/make>. Acesso em: 06 mai. 2020. Citado na página 43.

HERDER, J. N. et al. Minix 3: A highlyreliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, v. 40, p. 80–89, 2006. Citado 3 vezes nas páginas 4, 33 e 34.

INTEL. *Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual*. 2019. Disponível em: <https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html>. Acesso em: 08 mai. 2020. Citado na página 52.

International Organization for Standardization. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. 2018. Disponível em: <https://www.iso.org/standard/74528.html>. Acesso em: 24 nov. 2020. Citado na página 59.

KERNIGHAN, B. W.; RITCHIE, D. M. *The C programming language*. 2rd. ed. [S.l.]: Prentice Hall, 1988. Citado na página 61.

MAZIERO, C. A. *Sistemas Operacionais: Conceitos e Mecanismos*. Curitiba, PR, Brasil: UFPR, 2019. Disponível em: <http://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=socm:socm-livro.pdf>. Citado 6 vezes nas páginas 4, 17, 18, 19, 21 e 22.

MOLLOY, J. *Roll your own toy UNIX-clone OS*. 2008. Disponível em: <http://www.jamesmolloy.co.uk/tutorial_html/>. Acesso em: 23 abr. 2020. Citado na página 35.

OSDEV. *8259 PIC*. 2018. Disponível em: <https://wiki.osdev.org/8259_PIC>. Acesso em: 22 abr. 2020. Citado na página 20.

OSDEV. *C*. 2018. Disponível em: <https://wiki.osdev.org/C>. Acesso em: 24 abr. 2020. Citado na página 41.

OSDEV. *El-Torito*. 2018. Disponível em: <https://wiki.osdev.org/El-Torito>. Acesso em: 29 abr. 2020. Citado na página 17.

OSDEV. *Interrupts*. 2018. Disponível em: <https://wiki.osdev.org/Interrupts>. Acesso em: 21 abr. 2020. Citado na página 20.

OSDEV. *BIOS*. 2019. Disponível em: <https://wiki.osdev.org/BIOS>. Acesso em: 08 mar. 2020. Citado 3 vezes nas páginas 5, 14 e 15.

OSDEV. *CPU Registers x86*. 2019. Disponível em: <https://wiki.osdev.org/CPU_Registers_x86>. Acesso em: 18 abr. 2020. Citado 5 vezes nas páginas 5, 26, 27, 28 e 29.

OSDEV. *Exceptions*. 2019. Disponível em: <https://wiki.osdev.org/Exceptions>. Acesso em: 21 abr. 2020. Citado na página 20.

OSDEV. *Global Descriptor Table*. 2019. Disponível em: <https://wiki.osdev.org/Global_Descriptor_Table>. Acesso em: 29 abr. 2020. Citado 2 vezes nas páginas 4 e 51.

OSDEV. *ISO 9660*. 2019. Disponível em: <https://wiki.osdev.org/ISO_9660>. Acesso em: 29 abr. 2020. Citado na página 17.

OSDEV. *MBR (x86)*. 2019. Disponível em: <https://wiki.osdev.org/MBR_(x86)>. Acesso em: 08 mar. 2020. Citado 2 vezes nas páginas 5 e 16.

OSDEV. *GCC Cross-Compile*. 2020. Disponível em: <https://wiki.osdev.org/GCC_Cross-Compiler>. Acesso em: 24 abr. 2020. Citado na página 42.

OSDEV. *GPT*. 2020. Disponível em: <https://wiki.osdev.org/GPT>. Acesso em: 29 abr. 2020. Citado na página 16.

OSDEV. *James Molloy's Tutorial Known Bugs*. 2020. Disponível em: <https://wiki.osdev.org/James_Molloy's_Tutorial_Known_Bugs>. Acesso em: 23 abr. 2020. Citado na página 36.

OSDEV. *PCI*. 2020. Disponível em: <https://wiki.osdev.org/PCI>. Acesso em: 29 out. 2020. Citado 2 vezes nas páginas 5 e 24.

OSDEV. *Programmable Interval Timer*. 2020. Disponível em: <https://wiki.osdev.org/Programmable_Interval_Timer>. Acesso em: 29 out. 2020. Citado 2 vezes nas páginas 21 e 22.

OSDEV. *UEFI*. 2020. Disponível em: <https://wiki.osdev.org/UEFI>. Acesso em: 29 abr. 2020. Citado na página 16.

PCI Special Interest Group. *PCI Local Bus Specification*. 1998. Disponível em: <https://www.ics.uci.edu/~harris/ics216/pci/PCI_22.pdf>. Acesso em: 29 out. 2020. Citado na página 24.

PESSé, S. *How-to-Make-a-Computer-Operating-System*. 2015. Disponível em: <https://github.com/SamyPesse/How-to-Make-a-Computer-Operating-System>. Acesso em: 28 out. 2020. Citado 3 vezes nas páginas 4, 37 e 39.

PESSé, S. *How to Make an Operating System*. 2018. Disponível em: <https://samypesse.gitbook.io/how-to-create-an-operating-system/>. Acesso em: 23 nov. 2020. Citado na página 39.

QEMU. *QEMU*. 2020. Disponível em: <https://wiki.qemu.org/Main_Page>. Acesso em: 28 abr. 2020. Citado na página 43.

Santa Cruz Operation. *SYSTEM V APPLICATION BINARY INTERFACE - Intel386 Architecture Processor Supplement*. 1997. Disponível em: <http://www.sco.com/developers/devspecs/abi386-4.pdf>. Acesso em: 24 nov. 2020. Citado na página 64.

TANENBAUM, A. S. Structured computer organization. In: _____. *Structured computer organization*. [S.l.]: Pearson Education India, 2012. p. 1–10. Citado 5 vezes nas páginas 4, 14, 25, 31 e 32.

TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 6rd. ed. [S.l.]: Pearson, 2016. Citado 3 vezes nas páginas 4, 14 e 23.

TANENBAUM, A. S.; HERDER, J. N.; BOS, H. Can we make operating systems reliable and secure? *Computer*, IEEE, v. 39, n. 5, p. 44–51, 2006. Citado 2 vezes nas páginas 31 e 33.

von Neumann, J. *First Draft of a Report on the EDVAC*. 1945. Disponível em: <http://web.mit.edu/STS.035/www/PDFs/edvac.pdf>. Acesso em: 27 out. 2020. Citado na página 14.

WIKIPEDIA. *Programmable interval timer*. 2018. Disponível em: <https://en.wikipedia.org/wiki/Programmable_interval_timer>. Acesso em: 29 out. 2020. Citado na página 21.

WIKIPEDIA. *Advanced Vector Extensions*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Advanced_Vector_Extensions>. Acesso em: 18 abr. 2020. Citado na página 27.

WIKIPEDIA. *Bus*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Bus_(computing)>. Acesso em: 05 nov. 2020. Citado na página 31.

WIKIPEDIA. *C standard library*. 2020. Disponível em: <https://en.wikipedia.org/wiki/C_standard_library>. Acesso em: 29 out. 2020. Citado na página 59.

WIKIPEDIA. *Control register*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Control_register>. Acesso em: 19 abr. 2020. Citado na página 28.

WIKIPEDIA. *Dynamic Linker*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Dynamic_linker>. Acesso em: 29 out. 2020. Citado na página 59.

WIKIPEDIA. *Extended Industry Standard Architecture*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Extended_Industry_Standard_Architecture>. Acesso em: 05 nov. 2020. Citado na página 31.

WIKIPEDIA. *FLAGS register*. 2020. Disponível em: <https://en.wikipedia.org/wiki/FLAGS_register>. Acesso em: 19 abr. 2020. Citado na página 28.

WIKIPEDIA. *Intel 80386*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Intel_80386>. Acesso em: 12 nov. 2020. Citado na página 25.

WIKIPEDIA. *Interrupt*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Interrupt>. Acesso em: 21 abr. 2020. Citado 3 vezes nas páginas 4, 19 e 20.

WIKIPEDIA. *PCI Express*. 2020. Disponível em: <https://en.wikipedia.org/wiki/PCI_Express>. Acesso em: 05 nov. 2020. Citado na página 31.

WIKIPEDIA. *Peripheral Component Interconnect*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Peripheral_Component_Interconnect>. Acesso em: 05 nov. 2020. Citado na página 31.

WIKIPEDIA. *Power-on self-test*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Power-on_self-test>. Acesso em: 07 mar. 2020. Citado na página 14.

WIKIPEDIA. *Protection ring*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Protection_ring>. Acesso em: 28 out. 2020. Citado na página 30.

WIKIPEDIA. *Streaming SIMD Extensions*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions>. Acesso em: 18 abr. 2020. Citado na página 27.

WIKIPEDIA. *Task (computing)*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Task_(computing)>. Acesso em: 05 nov. 2020. Citado na página 22.

WIKIPEDIA. *Time-sharing*. 2020. Disponível em: <https://en.wikipedia.org/wiki/Time-sharing>. Acesso em: 05 nov. 2020. Citado na página 23.

WIKIPEDIA. *x86 assembly language*. 2020. Disponível em: <https://en.wikipedia.org/wiki/X86_assembly_language>. Acesso em: 09 mar. 2020. Citado 2 vezes nas páginas 5 e 26.

XEN. *Xen Project Software Overview*. 2018. Disponível em: <https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview>. Acesso em: 26 mai. 2020. Citado na página 43.