Faculdade de Tecnologia de Americana "Ministro Ralph Biasi"

Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Lucas Henrique Barbosa Berganton

Introdução ao desenvolvimento de kernels de sistemas operacionais para a arquitetura x86

Americana, SP

2025

Lucas Henrique Barbosa Berganton

Introdução ao desenvolvimento de kernels de sistemas operacionais para a arquitetura x86

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas na área de concentração em Sistema Operacional.

Orientador: Prof. Me. Rossano Pablo Pinto

Este trabalho corresponde à versão final do Trabalho de Conclusão de Curso apresentado por Lucas Henrique Barbosa Berganton e orientado pelo Prof. Me. Rossano Pablo Pinto.

Americana, SP
2025

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana Ministro Ralph Biasi- CEETEPS Dados Internacionais de Catalogação-na-fonte

BERGANTON, Lucas Henrique Barbosa

Introdução ao desenvolvimento de kernels de sistemas operacionais para a arquitetura x86. / Lucas Henrique Barbosa Berganton – Americana, 2025.

90f.

Monografia (Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas) - - Faculdade de Tecnologia de Americana Ministro Ralph Biasi – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof. Ms. Rossano Pablo Pinto

1. Assembly – linguagem de programação 2. C – linguagem de programação 3. Sistemas operacionais. I. BERGANTON, Lucas Henrique Barbosa II. PINTO, Rossano Pablo III. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana Ministro Ralph Biasi

CDU: 681.3.061Assembly
681.3.061C
681.3.066

Elaborada pelo autor por meio de sistema automático gerador de ficha catalográfica da Fatec de Americana Ministro Ralph Biasi.

Lucas Henrique Barbosa Berganton

Introdução ao desenvolvimento de kernels de sistemas operacionais para a arquitetura x86

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas pelo Centro Paula Souza – FATEC Faculdade de Tecnologia de Americana Ministro Ralph Biasi.
Área de concentração: Análise e Desenvolvimento de Sistemas.

Americana, 4 de dezembro de 2025.

Banca Examinadora:

Rossano Pablo Pinto
Mestre
Fatec Americana "Ministro Ralph Biasi"

Eduardo Antonio Vicentini
Mestre
Fatec Americana "Ministro Ralph Biasi"

Ivan Menerval da Silva
Doutor
Fatec Americana "Ministro Ralph Biasi"

AGRADECIMENTOS

Em primeiro lugar, agradeço à meu pai Odair Berganton e à minha mãe Silvane Silverio Barbosa Berganton, pelo apoio e amor que sempre me deram.

À meus amigos, que sempre me motivaram e sempre estiveram comigo em todos os momentos.

Aos meus professores, que contribuíram fortemente na formação do meu caráter acadêmico e profissional.

E ao meu orientador, Prof. Me. Rossano Pablo Pinto, pelo acompanhamento e auxílio que me guiaram na realização deste trabalho.

RESUMO

Este trabalho tem como objetivo central analisar o funcionamento e o desenvolvimento de *kernels* de sistemas operacionais para a arquitetura x86, com o intuito de elucidar como os recursos computacionais são acessados e abstraídos para fornecer uma interface simplificada ao usuário e ao programador. Para concretizar este estudo, adotou-se uma abordagem prática baseada na análise de um *kernel* didático denominado BergOS, desenvolvido pelo autor em linguagens C e *assembly* para a plataforma IA-32, membro da família x86. A motivação reside na premissa de que conceitos teóricos complexos, como a separação entre espaços de *kernel* e usuário, só são satisfatoriamente compreendidos quando examinados em implementações reais, as quais são inerentemente dependentes da arquitetura do hardware. Com base em uma revisão bibliográfica que aborda tópicos sobre sistemas operacionais, arquitetura de computadores, programação *assembly*, entre outros, as partes do BergOS são analisadas minuciosamente. Começando pelo *bootloader*, que carrega o *kernel* na memória, passando pelo *driver* de VGA, que usa E/S mapeada na memória para se comunicar com o dispositivo de vídeo e renderizar caracteres na tela, até a definição e implementação da interface do emulador de terminal do BergOS, que fornece rotinas de alto nível, oferecendo uma camada de abstração para outras partes do *kernel* e programas aplicativos poderem escrever mensagens na tela sem se preocupar com os detalhes do *hardware*. A contribuição deste trabalho está no fato de ele se aprofundar na conexão inerente entre sistema operacional e *hardware*, não se limitando apenas ao campo teórico e abstrato, mas apresentando e analisando uma implementação real e simples dos conceitos. Mesmo abordando detalhes da arquitetura x86 em profundidade, ainda há aspectos importantes que não são estudados neste trabalho, como a separação de espaço de *kernel* e espaço de usuário, mecanismos de interrupção, entrada de dados com teclado e processos, que são ganchos para estudos futuros. A conclusão é que o estudo de sistemas operacionais deve vir acompanhado de um estudo de arquitetura de computadores, e que, apesar de sua simplicidade, o BergOS demonstra ser uma fonte interessante de exemplos práticos dos conceitos teóricos abstratos.

Palavras Chave: Sistema Operacional; Kernel; x86.

ABSTRACT

This work aims to analyze the operation and development of operating system kernels for the x86 architecture, in order to elucidate how computational resources are accessed and abstracted to provide a simplified interface for the user and programmer. To carry out this study, we developed a practical approach based on the analysis of a didactic kernel called BergOS, developed by the author in C and assembly languages for the IA-32 platform, a member of the x86 family. The motivation lies in the proposition that complex theoretical concepts, such as the separation between kernel and user spaces, are only satisfactorily understood when examined in real implementations, as these are indirectly dependent on the hardware architecture. Based on a literature review covering details about operating systems, computer architecture, assembly programming, among others, the parts of BergOS are meticulously verified. Starting with the bootloader, which loads the kernel into memory, moving on to the VGA driver, which uses memory-mapped I/O to communicate with the video device and render characters on the screen, and finally to the definition and implementation of the BergOS terminal emulator interface, which provides high-level routines, offering an abstraction layer so that other parts of the kernel and application programs can write messages to the screen without worrying about hardware details. The contribution of this work lies in its in-depth exploration of the inherent connection between operating system and hardware, not limiting itself to the theoretical and abstract field, but presenting and analyzing a real and simple implementation of the concepts. Even while addressing details of the x86 architecture in depth, there are still important aspects not covered in this work, such as the separation of kernel space and user space, interrupt mechanisms, keyboard input, and processes, which are hooks for future studies. The conclusion is that the study of operating systems should be accompanied by a study of computer architecture, and that, despite its simplicity, BergOS proves to be an interesting source of practical examples of abstract theoretical concepts.

Keywords: *Operating System; Kernel; x86.*

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

Este trabalho tem por objetivo analisar o funcionamento e desenvolvimento de *kernels* de sistemas operacionais para a família de arquiteturas x86.

O objetivo geral é apresentar uma visão sobre como *kernels* são programados, estudando como são acessados os recursos computacionais e como eles são abstraídos para apresentar uma interface simples para o usuário e o programador.

Como objetivo específico, buscou-se analisar um *kernel* simples para a arquitetura x86 chamado BergOS, escrito em linguagem C e *assembly*. Esse *kernel*, desenvolvido pelo autor, faz uso de estruturas e recursos da arquitetura e implementa mecanismos básicos de entrada/saída, servindo, assim, como uma manifestação prática dos conceitos teóricos abordados.

A motivação para este trabalho está na certeza de que certos conceitos teóricos só são satisfatoriamente compreendidos com exemplos reais. A separação do espaço de usuário e espaço de *kernel*, por exemplo, é realizada pelo processador, que é configurado pelo sistema operacional para se comportar conforme o desejado, o que torna esse recurso dependente da arquitetura da máquina para a qual ele foi programado. Uma abordagem geral, que tente ser independente de arquitetura, terá de se restringir a ideias vagas e abstratas, limitando a compreensão do tema.

Por outro lado, este trabalho faz um estudo com tecnologias específicas. No caso, o *kernel* BergOS, escrito para a arquitetura IA-32, membro da família de arquiteturas x86, usando o GCC como compilador de C e o NASM como *assembler*. A implementação de conceitos como a comunicação do *kernel* com um *driver* de vídeo para saída de dados é detalhada através de um estudo dos mecanismos arquiteturais que permitem a comunicação do processador com o dispositivo de vídeo.

Este trabalho escolheu a arquitetura x86 por ser uma arquitetura madura, com abundância de documentação e sistemas operacionais suportados; e por ser comum em computadores pessoais, tanto *desktops* quanto *laptop*s, o que torna fácil para o leitor ter a experiência de executar o BergOS em uma máquina real.

A escolha da linguagem C se motiva por ser uma linguagem de sistema simples, que foi projetada especialmente para a programação de sistemas operacionais, com

implementações eficientes e uma boa integração com outras linguagens de sistema, muito devido à sua já mencionada simplicidade. O uso do NASM como *assembler* é vantajoso, pois além de ter diretivas de pré-processamento poderosas, a sua sintaxe é a usada nos manuais da Intel.

O BergOS é um projeto de código aberto e pode ser acessado através de um repositório público no GitHub (BERGANTON, 2025). Esse repositório também contém instruções para compilar e executar o BergOS.

Como o código do BergOS pode evoluir com o tempo ao ponto de ser substancialmente diferente daquele estudado neste trabalho, uma *branch* chamada *tcc* foi criada no repositório oficial. O foco dessa *branch* é apenas servir de referência para este trabalho; portanto, mesmo que o projeto evolua, o código estudado aqui poderá ser acessado facilmente.

Em determinados trechos, neste trabalho, um caminho de diretórios será usado para especificar algum arquivo do BergOS. Por exemplo, o código do kernel está em *./kernel/main.c*. Esses caminhos sempre tomam como ponto de partida o diretório raiz do projeto, conforme o repositório no GitHub.

O trabalho é dividido em seis capítulos e um apêndice. O capítulo 1 (Introdução) estabelece os objetivos e motivações do estudo. O capítulo 2 (Revisão Bibliográfica) faz uma revisão teórica de tópicos referentes a sistemas operacionais, arquitetura de computadores, arquitetura x86 e programação em linguagem *assembly*, que são necessários para a compreensão dos capítulos seguintes, onde esses conhecimentos serão aplicados.

O capítulo 3 (*Bootloader*) trará uma análise minuciosa do programa do *bootloader* do BergOS e como ele carrega o *kernel* para a memória e deixa a máquina em um estado esperado por ele. Neste capítulo, os conceitos abordados no capítulo 2 serão postos em prática, como programação em *assembly*, manipulação de registradores, segmentação de memória, interrupções de software, dentre outros.

O capítulo 4 (*Driver* de VGA) começa apresentando o padrão VGA e como ele faz uso de mecanismos de E/S mapeada na memória para fornecer uma maneira do programador manipular o vídeo apenas lendo e escrevendo na memória principal.

Após isso, será feita uma análise do *driver* de VGA do BergOS, onde todo o código é estudado, da interface à implementação.

O capítulo 5 (Emulador de terminal) define e implementa uma interface com rotinas feitas para manipular o emulador de terminal do BergOS. A implementação da interface faz uso do *driver* de VGA estudado no capítulo 4 para manipular o vídeo e escrever os caracteres na tela. É neste momento em que o BergOS estabelece uma camada de abstração, permitindo que outras partes do *kernel* e programas aplicativos sejam programados usando interfaces e protocolos livres das complexidades e especificidades do *hardware*. Por fim, o capítulo 6 (Considerações finais) conclui o estudo levantando suas contribuições e limitações.

O apêndice A (Processo de Compilação do BergOS) aborda dois temas principais: o processo de compilação do BergOS, incluindo suas dificuldades e os meios para contorná-las; e as questões referentes à execução do sistema, e como ele pode ser executado em emuladores como QEMU e em máquinas reais compatíveis com a arquitetura IA-32.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo serão introduzidos os conceitos teóricos referentes a sistemas operacionais e a arquitetura x86.

2.1 Sistema Operacional

Um sistema operacional é um *software* complexo, e isso já se torna evidente na dificuldade em defini-lo. É intuitivo concebê-lo como o conjunto de programas e aplicativos que vêm junto a uma instalação de um sistema, como *shells*, interfaces gráficas, gerenciadores de arquivos etc. Apesar de serem recursos importantes e úteis, eles não são, a princípio, aspectos substanciais e podem facilmente ser substituídos sem nenhuma alteração profunda no sistema. Um sistema operacional tem um papel muito mais profundo do que é imediatamente visível ao usuário comum.

Os programas citados são classificados como programas aplicativos, isto é, são feitos para realizar tarefas específicas e desejadas por usuários, mas não têm relação com o sistema computacional em si. Já os programas de sistema são destinados a gerenciar o computador em alguma instância. O programa de sistema mais básico é o sistema operacional, que, em uma definição sucinta, é "[…] um programa que gerencia o *hardware* de um computador. Ele também fornece uma base para os programas aplicativos e atua como intermediário entre o usuário e o *hardware* do computador" (SILBERSCHATZ; GALVIN; GAGNE, 2015, n. p.).

Um sistema operacional tem, então, um trabalho complexo. Além de gerenciar os recursos de um sistema computacional, também deve facilitar que estes sejam acessados de forma conveniente e segura, protegendo-os contra usos indevidos. Isso é atingido com o auxílio do *hardware*, que permite a execução de um programa em modo núcleo ou modo usuário. O núcleo do sistema operacional é o que executa em modo núcleo, como afirma Tanenbaum e Bos (2016, p. 1).

> O sistema operacional, a peça mais fundamental de *software*, opera em modo núcleo (também chamado modo supervisor). Nesse modo ele tem acesso completo a todo o *hardware* e pode executar qualquer instrução que a máquina for capaz de executar. O resto do *software* opera em modo usuário, no qual apenas um subconjunto das instruções da máquina está disponível. Em particular, aquelas instruções que afetam o controle da máquina ou realizam E/S (Entrada/Saída) são proibidas para programas de modo usuário (TANENBAUM; BOS, 2016, p. 1).

A definição apresentada pode ser aprofundada. Tanenbaum e Bos detalham essa perspectiva argumentando que um sistema operacional pode ser entendido através de duas abordagens, como uma máquina estendida e como um gerenciador de recursos.

2.1.1 Sistema Operacional Como Uma Máquina Estendida

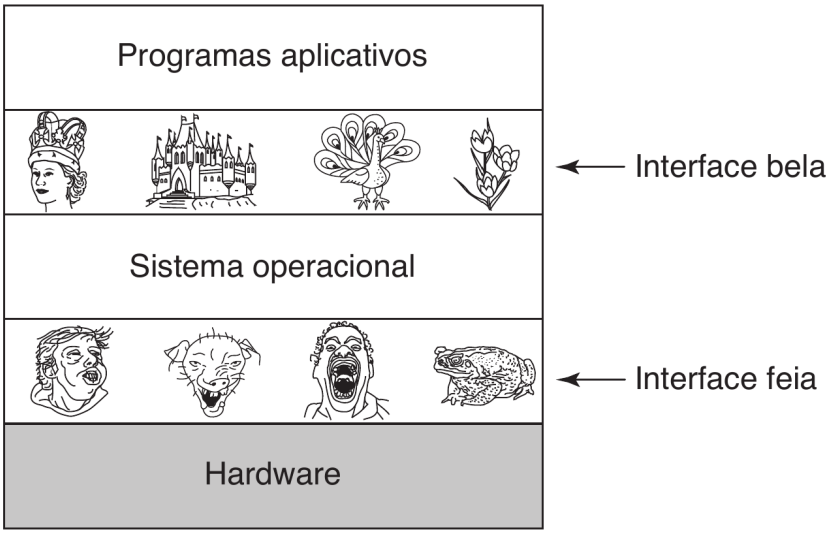
Em um sistema computacional, o *hardware* se apresenta aos programadores com uma complexidade exótica. “Processadores reais, memórias, discos e outros dispositivos são muito complicados e apresentam interfaces difíceis, desajeitadas, idiossincráticas e inconsistentes para as pessoas que têm de escrever *softwares* para elas utilizarem” (TANENBAUM; BOS, 2016, p. 3).

Para manipular um processador, por exemplo, é necessário conhecer os detalhes de sua arquitetura, como seus registradores, instruções, mecanismos de acesso à memória, mecanismos de segurança e mecanismos de acesso aos dispositivos de E/S, o que torna sua programação não apenas complicada, mas dependente de uma arquitetura em específico.

Com dispositivos de E/S a situação é ainda pior. As interfaces para manipulá-los tendem a ser primitivas e complicadas. Para ler um *byte* de um disco, por exemplo, podem ser necessárias várias instruções de máquina. Também é esperado que o mesmo programa que executa no processador seja capaz de lidar com diferentes modelos de dispositivos de uma mesma classe.

Em suma, o *hardware* é complicado, e qualquer programador se beneficiaria de estar longe de seus detalhes. Sendo assim, Tanenbaum e Bos (2016, p. 3) concluem que “Uma das principais tarefas dos sistemas operacionais é esconder o *hardware* e em vez disso apresentar programas (e seus programadores) com abstrações de qualidade, limpas, elegantes e consistentes com as quais trabalhar. Sistemas operacionais transformam o feio em belo […]”. A Figura 1 traz uma representação visual dessa ideia.

Figura 1 – Abstração fornecida pelo sistema operacional



Fonte: Tanenbaum e Bos (2016)

Essa interface que abstrai as complexidades do *hardware* fornecendo uma visão mais simples do sistema aos programas aplicativos é o que conforma a noção de máquina estendida, que permite maior portabilidade e uma programação conveniente baseada em abstrações.

2.1.2 Sistema Operacional Como Um Gerenciador de Recursos

Essa abordagem enxerga o sistema operacional como um programa que administra os recursos de um sistema computacional. “Resumindo, essa visão do sistema operacional sustenta que a sua principal função é manter um controle sobre quais programas estão usando qual recurso, conceder recursos requisitados, contabilizar o seu uso, assim como mediar requisições conflitantes de diferentes programas e usuários” (TANENBAUM; BOS, 2016, p. 4).

Dado o fato de que computadores modernos permitem que múltiplos programas compartilhem recursos, um gerenciamento cuidadoso passa a fazer parte das funcionalidades de um sistema operacional. “O gerenciamento de recursos inclui a multiplexação (compartilhamento) de recursos de duas maneiras diferentes: no tempo e no espaço.” (TANENBAUM; BOS, 2016, p. 4).

A multiplexação no tempo é o intervalo no qual um programa terá direito a usar o processador. Depois de concluir o seu tempo, ou ser bloqueado devido à dependência de um evento externo para continuar sua execução, o sistema operacional fará outro programa em espera tomar seu lugar.

A multiplexação no espaço é o direito dos programas a uma parte do recurso. A memória principal, por exemplo, é dividida em partições que são associadas a um programa cada, de modo que cada um tenha sua própria memória e não possa acessar os recursos do outro.

2.1.3 Inconsistências na Definição

Nenhuma definição de sistema operacional é totalmente satisfatória. Se formos mais rigorosos com o que foi definido até aqui, o *firmware Basic Input Output System* (BIOS) de uma placa-mãe pode ser considerado um sistema operacional, afinal, ele administra recursos do sistema computacional com a operação *Power On Self Test* (POST), que detecta falhas no *hardware*, e fornece uma camada de abstração através das funções BIOS, formando uma interface mais simples e conveniente para interação com o *hardware*, ainda que limitada. O sistema MS-DOS, bem como outros sistemas antigos, era fortemente baseado nas funções BIOS, permitindo que programas aplicativos as usassem em sua programação (DODGE; IRVINE; NGUYEN, 2005, p. 80). Sendo assim, as funções BIOS compunham boa parte da máquina estendida do sistema da Microsoft.

A falta de consenso é corroborada por Silberschatz, Galvin e Gagne “[…] não temos uma definição universalmente aceita sobre o que compõe o sistema operacional”, e concluem “Uma definição mais comum, que é a que costumamos seguir, é que o sistema operacional é o único programa que permanece em execução no computador durante todo o tempo — chamado, em geral, de *kernel*” (2015, n. p.). O *kernel* é a parte do sistema operacional que executa em modo núcleo, portanto, tem acesso a todas as instruções do processador.

Mas a afirmação de que o sistema operacional é o que executa em modo núcleo também gera problemas. Sistemas embarcados ou processadores antigos podem não ter uma divisão entre modo núcleo e modo usuário, e mesmo alguns programas que executam em modo usuário estão tão intimamente ligados ao funcionamento do

sistema operacional que se torna difícil não considerá-los como parte dele, assim como afirma Tanenbaum e Bos (2016, p. 2).

> […] muitas vezes há um programa que permite aos usuários que troquem suas senhas. Não faz parte do sistema operacional e não opera em modo núcleo, mas claramente realiza uma função sensível e precisa ser protegido de uma maneira especial. Em alguns sistemas, essa ideia é levada ao extremo, e partes do que é tradicionalmente entendido como sendo o sistema operacional (como o sistema de arquivos) é executado em espaço do usuário. Em tais sistemas, é difícil traçar um limite claro. Tudo o que está sendo executado em modo núcleo faz claramente parte do sistema operacional, mas alguns programas executados fora dele também podem ser considerados uma parte dele, ou pelo menos estão associados a ele de modo próximo (TANENBAUM; BOS, 2016, p. 2).

Não parece correto ou pragmático considerar o BIOS como um sistema operacional, tornando uma definição muito ampla pouco adequada. O mesmo ocorre ao desconsiderar um sistema embarcado sem modo núcleo como sistema operacional, o que também torna uma definição muito restrita pouco adequada.

O debate é amplo e exceções vão existir, mas para fins deste trabalho será adotada a definição já apresentada de um sistema operacional como um intermediário entre o usuário e o computador que gerencia o hardware e fornece uma base para os programas aplicativos (SILBERSCHATZ; GALVIN; GAGNE, 2015, n. p.).

2.2 Arquitetura x86

Como um sistema operacional tem a tarefa de gerenciar o *hardware*, se torna fundamental entender em detalhes a arquitetura para qual ele será programado, bem como sua arquitetura do conjunto de instruções. Como definido por Stallings (2017, p. 2).

> Arquitetura de computador refere-se aos atributos de um sistema visíveis a um programador ou, em outras palavras, aqueles atributos que possuem um impacto direto sobre a execução lógica de um programa. Um termo que é muitas vezes usado de maneira intercambiável com as arquiteturas de computadores é arquitetura de conjunto de instrução (ISA — do inglês, Instruction Set Architecture). O ISA define os formatos de instruções, códigos de operação da instrução (opcodes), registradores, memória de dados e instrução; o efeito das instruções executadas nos registradores e na memória; e um algoritmo para o controle da execução das instruções (STALLINGS, 2017, p. 2).

2.2.1 História da arquitetura x86

Em 1971 a Intel fez um importante avanço para a área da computação, o desenvolvimento do 4004, o primeiro microprocessador da história. Após esse evento, a tecnologia de microprocessadores foi evoluindo, culminando no lançamento do microprocessador 8086 em 1978. “O 8086 tem registradores de 16 *bits* e um barramento de dados externo de 16 *bits*, com endereçamento de 20 *bits*, proporcionando um espaço de endereçamento de 1 MB.” (INTEL CORPORATION, 2025, Vol. 1 2-1, tradução nossa).

Devido ao sucesso do 8086, a Intel desenvolveu outros processadores que mantinham compatibilidade com a ISA do 8086, expandindo suas funcionalidades com novas instruções, modos de operações e tecnologias. Essa família de microprocessadores baseados na ISA do 8086 forma o que é chamado genericamente de arquitetura x86.

Em 1982 foi lançado o 80286 que, dentre outras novidades, expande a capacidade de endereçamento do 8086 de 20 *bits* para 24 *bits* e adiciona um novo modo de operação, o modo protegido. “O modo de operação determina quais instruções e recursos da arquitetura estão disponíveis” (INTEL CORPORATION, 2025, Vol. 1 3-1, tradução nossa). O modo protegido usa os registradores de segmento como índices para tabelas que descrevem as permissões e atributos daquele segmento. Por questões de compatibilidade com o 8086, o 80286, bem como seus sucessores, não iniciam no modo protegido, mas no modo real.

Em 1985 foi lançado o 80386, o primeiro microprocessador de 32 *bits* da família x86. Para permitir a execução de programas de 16 *bits* em modo protegido foi adicionado o modo virtual-8086.

A família x86 continuou evoluindo com novos lançamentos, como o 80486, Pentium, Pentium Pro, Pentium II, Pentium III, Core 2 etc. A AMD também produz microprocessadores compatíveis com a família x86. Uma de suas contribuições mais importantes foi o desenvolvimento de uma extensão da arquitetura, chamada de x86_64, para processadores de 64 *bits*. A extensão também foi adotada pela Intel.

BergOS é programado para o 80386, fazendo uso do modo protegido e de instruções de 32 *bits*. Portanto, a arquitetura correspondente, chamada IA-32, será a estudada neste trabalho.

2.2.2 Assembly

O uso de linguagens de programação de alto nível na programação de sistemas operacionais é adequado. Por serem baseadas em máquinas abstratas, apresentam uma gramática elegante que torna a programação agradável e menos propensa a erros. Porém, na programação de sistemas operacionais é necessário a manipulação da máquina real, que possui diferenças substanciais das máquinas abstratas de linguagens de alto nível. Mesmo C, conhecida pelo controle que fornece ao programador sobre a máquina subjacente, não é capaz de manipular recursos específicos de uma arquitetura como a x86, visto que registradores, segmentação de memória, tabela de descritores, modos de operação e ponteiros de pilha não fazem parte de sua máquina abstrata. Portanto, mesmo que o uso de uma linguagem de alto nível seja recomendado na maior parte do *software* que compõe o sistema operacional, se torna indispensável o uso de uma linguagem de baixo nível, isto é, uma linguagem de programação capaz de expressar instruções de uma máquina real.

As linguagens de máquina são as linguagens de baixo nível que são diretamente interpretadas por alguma máquina real. Sua programação é trabalhosa e propensa a erros. Para tornar a programação de baixo nível mais conveniente, fornecendo um mínimo de abstração com notações em texto das instruções de máquina, existem as linguagens de montagem ou linguagens *assembly*. Como descreve Zhirkov (2018, n. p.).

> A linguagem *Assembly* para um dado processador é uma linguagem de programação constituída de mnemônicos para cada possível instrução binária codicada (código de máquina). Ela deixa a programação em códigos de máquina muito mais simples, pois o programador então não precisa memorizar a codicação binária das instruções, apenas seus nomes e os parâmetros (ZHIRKOV, 2018, n. p.).

Após o programa ter sido escrito, um *software* chamado *assembler* é usado para transformar o programa *assembly* na linguagem de máquina correspondente. O BergOS é programado usando o *assembler* NASM.

Nem toda instrução *assembly* tem uma instrução em código de máquina equivalente. Frequentemente *assemblers* fornecem instruções que definem o comportamento da montagem ou informações do binário final.

O *assembly* da arquitetura x86 tem duas sintaxes distintas. A sintaxe oficial, usada nos manuais da Intel, é simplesmente chamada de Sintaxe Intel, enquanto a outra é chamada de Sintaxe AT&T. O *assembler* NASM usa a Sintaxe Intel, portanto é a sintaxe usada nos códigos do BergOS.

2.2.3 Registradores

A arquitetura x86 é baseada na arquitetura do computador IAS lançada em 1952, chamada de arquitetura de Von Neumann. A arquitetura de Von Neumann, mesmo que antiga, ainda é a base para a maioria dos computadores atuais, como reforça Stallings “Com raras exceções, todos os computadores de hoje têm essa mesma estrutura e função geral e são, por conseguinte, referidos como máquinas de von Neumann.” (2017, p. 11).

Um elemento importante da arquitetura de Von Neumann e, portanto, dos computadores atuais, são os registradores. Registradores “São células de memória colocadas diretamente no chip da CPU.” (ZHIRKOV, 2018, n. p.). Eles são mais rápidos que a memória principal e são extensivamente usados na programação em baixo nível. A maior parte das instruções de um programa envolvem mover dados entre registradores e entre registradores e a memória.

Como dito por Zhirkov “Na maior parte das vezes, um programador trabalhará com registradores de propósito geral.” (2018, n. p.). Os registradores AX, BX, CX, DX, SI, DI, BP e SP são os registradores de propósito geral da arquitetura do 8086, portanto, todos são de 16 *bits*. Eles podem ser usados livremente pelo programador, mas algumas instruções os usam como operandos ou para armazenar os resultados de um cálculo. Nesse caso, os registradores assumem significados especiais, conforme pode ser visto na Tabela 1.

Tabela 1 – Significados especiais dos registradores de propósito geral

Registrador	Significado	Uso
AX	Accumulator	Usado em cálculos

		aritméticos, para operandos e resultados.
BX	Base	Usado como ponteiro para dados.
CX	Counter	Usado como contador em operações de strings e loops.
DX	Data	Armazena dados de operações de E/S.
SI	Source index	Ponteiro para origem dos dados em operações de string.
DI	Destination index	Ponteiro para o destino dos dados em operações de string.
BP	Base Pointer	Ponteiro para a base da pilha de hardwre.
SP	Stack Pointer	Ponteiro para o topo da pilha de hardware.

Fonte: Elaborado pelo autor (2025).

Apesar dos registradores AX, BX, CX e DX terem 16 *bits*, seus 8 *bits* mais significativos podem ser acessados individualmente pelos nomes AH, BH, CH e DH respectivamente, bem como os menos significativos pelos nomes AL, BL, CL e DL respectivamente.

Com a introdução da arquitetura IA-32 e a expansão do barramento interno, os registradores de propósito geral foram expandidos para se adaptar à nova capacidade de 32 *bits*, sendo acessíveis pelos nomes EAX, EBX, ECX, EDX, ESI, EDI, EBP e ESP. Os 16 *bits* menos significativos dos novos registradores ainda são acessíveis pelos nomes antigos.

Outros registradores importantes da arquitetura x86 são o IP (*Instruction Pointer*), que armazena o endereço da próxima instrução a ser executada, o

equivalente ao PC (*Program Counter*) da arquitetura de Von Neumann; e o *FLAGS* que contém um grupo de *flags* de status, *flag* de controle e um grupo de *flags* de sistemas. Assim como os registradores de propósito geral, IP e FLAGS também têm versões de 32 *bits* para arquitetura IA-32, sendo chamados de EIP e EFLAGS.

2.2.4 Segmentos de memória

Segundo a Intel Corporation, a “segmentação fornece um mecanismo de isolamento de módulos individuais de código, dados e pilha, permitindo que múltiplos programas (ou tarefas) sejam executados no mesmo processador sem interferir um no outro.” (2025, Vol. 3A 3-1, tradução nossa). O mecanismo de segmentação funciona de forma diferente no modo real e no modo protegido.

Os registradores de propósito especial CS, DS, ES, FS, GS e SS, chamados de registradores de segmento, são usados no cálculo do endereço real que será acessado pelo processador. A Tabela 2 descreve os registradores de segmento.

Tabela 2 – Registradores de Segmento

Registrador	Significado	Uso
CS	*Code Segment* (Segmento de Código).	Usado para obter endereços relacionados a código executável.
DS	*Data Segment* (Segmento de Dados).	Usada para obter endereços relacionados a dados.
ES	*Extra Segment* (Segmento Extra).	Não tem um significado especial e pode ser usado livremente pelo programador.
FS		Não tem um significado especial e pode ser usado livremente pelo programador.
GS		Não tem um significado especial e pode ser usado livremente pelo programador.
SS	*Stack Segment* (Segmento de Pilha).	Usado para obter endereços relacionados a pilha.

Fonte: Elaborado pelo autor (2025).

No modo protegido, a segmentação é feita definindo tabelas especiais na memória que descrevem os segmentos. Os registradores de segmento passam a armazenar um valor chamado seletor de segmento, que serve de índice para selecionar um dos segmentos descritos nessas tabelas especiais (ZHIRKOV, 2018, n. p.). A *Global Descriptor Table* (GDT), é a única tabela de descritores de segmentos que precisa ser definida para ativar o modo protegido, portanto, será analisada no capítulo 3.

Apesar da sua importância no contexto da arquitetura x86, a segmentação é considerada um mecanismo legado. Como confirma Zhirkov "[…] a segmentação é uma criatura selvagem um tanto quanto difícil de lidar. Há motivos pelos quais ela não foi amplamente adotada pelos sistemas operacionais, nem igualmente pelos programadores (hoje em dia, ela foi praticamente abandonada)." (2018, n. p.).

2.2.5 Interrupções

Em arquitetura de computadores, uma interrupção é um evento que faz o processador parar o que está fazendo para executar um código de tratamento de interrupção, para depois retomar a execução de onde parou. Conforme elabora Zhirkov (2018, n. p.).

> As interrupções nos permitem alterar o controle de fluxo do programa em um instante arbitrário no tempo. Enquanto o programa estiver executando, eventos externos (dispositivos que exijam a atenção da CPU) ou internos (divisão por zero, nível de privilégio insuficiente para executar uma instrução, um endereço não canônico) poderão provocar uma interrupção, o que resultará em outro código sendo executado. Esse código é chamado de *handler* da interrupção (*interrupt handler*) e faz parte do software de um sistema operacional ou de um *driver* (ZHIRKOV, 2018, n. p.).

Também é possível causar uma interrupção através de uma instrução. Esse tipo de interrupção é chamada de interrupção por *software*. Como será elaborado na seção 2.3, as interrupções por *software* podem ser usadas para acessar as funções BIOS.

Na arquitetura x86, o *handler* de interrupção, bem como seus atributos, é definido em uma tabela semelhante a GDT chamada *Interrupt Descriptor Table* (IDT). BergOS define uma IDT, mas ela não será estudada neste trabalho.

2.3 BIOS

O BIOS é "[...] uma interface ou 'camada' de software que isola os sistemas operacionais e programas aplicativos de dispositivos de hardware específicos" (IBM, 1987, 1-3, tradução nossa). Sua função é fornecer uma leve abstração para que o programador *assembly* possa manipular dispositivos de bloco ou caractere sem se preocupar com suas características específicas.

A abstração é alcançada através de um conjunto de rotinas, às vezes chamadas de funções BIOS. As funções BIOS ficam armazenadas em uma *Read-Only Memory* (ROM) e são carregadas para a memória principal na inicialização do computador.

As interrupções de *hardware* são usadas para acessar rotinas do sistema. O número da interrupção corresponde ao tipo de serviço solicitado; por exemplo, a interrupção de número 0x10 refere-se a serviços de vídeo. O valor definido no registrador AH determina a função específica do BIOS a ser executada. Algumas rotinas exigem parâmetros adicionais, que são passados por meio de outros registradores. As funções BIOS disponíveis no serviço de vídeo podem ser consultadas na Figura 2.

Figura 2 – Funções BIOS disponíveis no serviço de vídeo

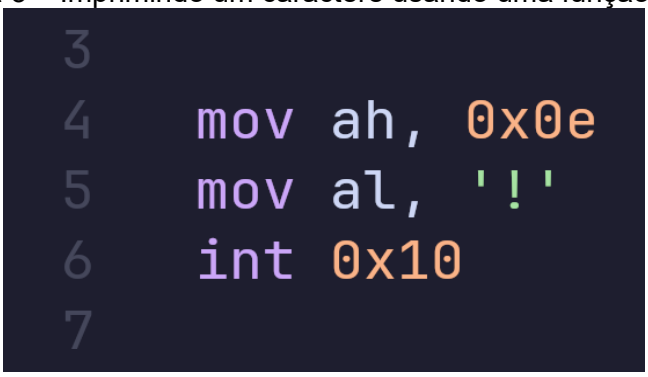
```
(AH) = 00H – Set Mode
(AH) = 01H – Set Cursor Type
(AH) = 02H – Set Cursor Position
(AH) = 03H – Read Cursor Position
(AH) = 04H – Read Light Pen Position
(AH) = 05H – Select Active Display Page
(AH) = 06H – Scroll Active Page Up
(AH) = 07H – Scroll Active Page Down
(AH) = 08H – Read Attribute/Character at Current Cursor Position
(AH) = 09H – Write Attribute/Character at Current Cursor Position
(AH) = 0AH – Write Character at Current Cursor Position
(AH) = 0BH – Set Color Palette
(AH) = 0CH – Write Dot
(AH) = 0DH – Read Dot
(AH) = 0EH – Write Teletype to Active Page
(AH) = 0FH – Read Current Video State
(AH) = 10H – Set Palette Registers
(AH) = 11H – Character Generator
(AH) = 12H – Alternate Select
(AH) = 13H – Write String
(AH) = 14H – Load LCD Character Font/Set LCD High-Intensity Substitute
(AH) = 15H – Return Physical Display Parameters for Active Display
(AH) = 16H to 19H – Reserved
(AH) = 1AH – Read/Write Display Combination Code
(AH) = 1BH – Return Functionality/State Information
(AH) = 1CH – Save/Restore Video State
(AH) = 1DH to FFH – Reserved
```

Fonte: IBM (1987)

É comum em programação de baixo nível o uso de números hexadecimais para representar valores, principalmente endereços de memória. Neste trabalho, todo valor precedido por “0x” deve ser entendido como um valor hexadecimal.

Para escrever um caractere na tela, por exemplo, a função BIOS *Write Teletype to Active Page* pode ser usada. A Figura 3 mostra um código NASM que usa a função BIOS citada para imprimir o caractere “!”:

Figura 3 – Imprimindo um caractere usando uma função BIOS



Fonte: Elaborado pelo autor (2025)

Primeiro o registrador AH é definido com o valor 0xE para selecionar a função BIOS que escreve na página ativa, depois o registrador AL é definido com o valor do caractere “!” (o NASM converte caracteres em aspas simples para valores ASCII), e, por fim, uma interrupção 0x10 é disparada por *software*, fazendo com que a função BIOS execute.

Além da camada de abstração, o BIOS cumpre outras funções importantes no sistema computacional. De acordo com Dodge, Irvine e Nguyen “As funcionalidades do BIOS podem ser divididas em três áreas: POST, *Setup* e *Boot*.” (2005, p. 80, tradução nossa).

A operação POST detecta e inicializa os componentes de *hardware*. Após a conclusão de POST, o sistema BIOS fornece ao usuário a possibilidade de entrar em modo *Setup*, onde é possível alterar algumas configurações do BIOS como a ordem de *boot*. Por fim, o BIOS executa a interrupção de número 0x19 que procura, na sequência definida pela ordem de *boot*, por um dispositivo bootável, carrega seu

primeiro setor para a memória e transfere o controle para o programa contido nele, geralmente um *bootloader* (DODGE, IRVINE, NGUYEN, 2005, p. 80).

2.4 E/S mapeada na memória

Qualquer arquitetura deve apresentar maneiras de se comunicar com dispositivos de E/S. Esses dispositivos podem ter interfaces muito complexas, o que faz da programação com instruções próprias de E/S inconveniente, verbosa e propensa a erros. Contudo, há uma técnica chamada de E/S mapeada na memória que contorna esse problema, tornando a programação de dispositivos de E/S mais fácil por fazer o dispositivo acessível através da memória principal. Como elabora Stallings (2017, p. 200).

> Com a E/S mapeada na memória, existe um único espaço de endereço para locais de memória e dispositivos de E/S. O processador trata os registradores de estado e dados dos módulos de E/S como locais de memória e usa as mesmas instruções de máquina para acessar a memória e os dispositivos de E/S. Assim, por exemplo, com dez linhas de endereço, um total combinado de 210 = 1.024 locais de memória e endereços de E/S podem ser aceitos, em qualquer combinação (STALLINGS, 2017, p. 200).

O emulador de terminal do BergOS usa um *driver* de *Video Graphics Array* (VGA), que utiliza a técnica de E/S mapeada na memória para permitir que o programa escreva ou desenhe na tela através da manipulação de endereços de memória.

O próximo capítulo apresentará o *bootloader* do BergOS acompanhado de uma análise que expõe o processo de carregar o *kernel* na memória e passar o controle da máquina para ele.

3 BOOTLOADER

Um *bootloader* tem o objetivo de carregar o *kernel* na memória e colocar a máquina em um estado esperado por ele. O *bootloader* do BergOS entrega o controle da máquina para o *kernel* com o processador em modo protegido, com as interrupções desligadas e o modo de vídeo definido para 3.

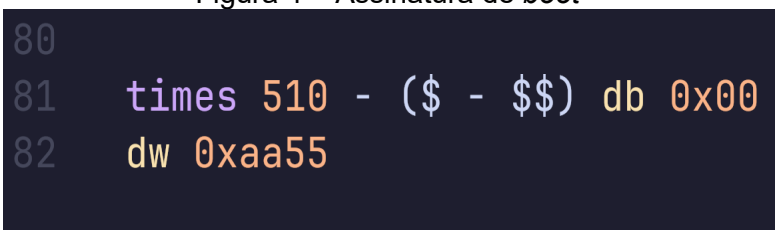
3.1 Pré-bootloader

Na etapa de *setup*, o BIOS fornece ao usuário a opção de alterar a ordem de *boot*, que é uma lista contendo dispositivos de armazenamento em massa, como HDs e SSDs. O BIOS lê o primeiro setor de cada dispositivo procurando por um dispositivo bootável. Um dispositivo bootável é aquele no qual os últimos dois *bytes* do seu primeiro setor contêm um número mágico chamado de assinatura de *boot*, o valor 0xAA55. Após um dispositivo bootável ter sido localizado, o BIOS executa a interrupção 0x19, que carrega o primeiro setor do dispositivo para o endereço 0x7C00 e salta para ele, transferindo o controle para o programa carregado. O estado da máquina após o fim da etapa de *boot* consiste no registrador CS com o valor zero, o registrador IP com o valor 0x7C00 e o registrador DL com o número do dispositivo no qual o *boot* ocorreu (IBM, 1987, 2-113).

Como apenas um setor do dispositivo é carregado, o *bootloader* não tem espaço para ser muito complexo. Caso mais de 512 *bytes*, tamanho de um setor, sejam necessários, um *multi-stage bootloader* pode ser usado, que “em vez de um único programa que carrega o sistema operacional diretamente, os *multi-stage bootloaders* dividem suas funcionalidades em programas menores que carregam uns aos outros sucessivamente.” (DODGE; IRVINE; NGUYEN, 2005, p. 80, tradução nossa). O *bootloader* do BergOS é simples e consegue ser contido em apenas um setor.

O *bootloader* do BergOS é um código *assembly* localizado no arquivo *./arch/i386/boot/bootloader.asm*, no qual as duas últimas linhas são responsáveis pela assinatura de *boot*, como mostra a Figura 4:

Figura 4 – Assinatura de *boot*



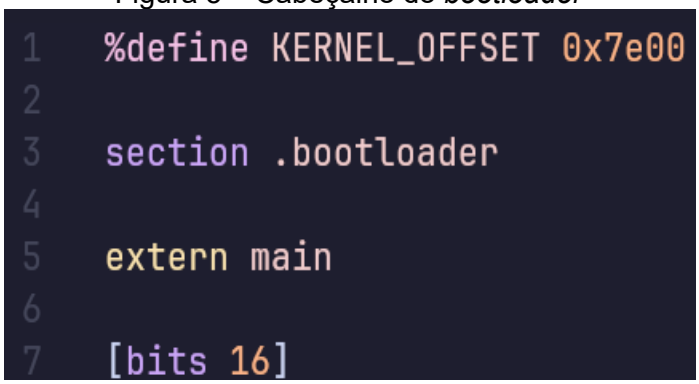
Fonte: Elaborado pelo autor (2025)

A linha 81 é responsável por preencher o resto do setor com zeros. O NASM possui “pseudo-instruções” que são instruções que não fazem parte da arquitetura x86, mas instruem o montador a realizar alguma ação. A instrução *times* é uma pseudo-instrução que diz para o NASM repetir uma instrução por determinado número de vezes. A pseudo-instrução *db* significa que, naquela parte do binário final, o NASM deve preencher com um *byte* de valor definido. O trecho “510 - ($- $$)” é uma forma de obter quantos *bytes* ainda não foram preenchidos para completar 510 *bytes*. A instrução toda diz para o NASM preencher o binário de zeros até o *byte* 510, reservando os últimos dois bytes para a assinatura de *boot*. Por fim, na linha 82, a pseudo-instrução *dw*, semelhante a *db*, com a exceção de que preenche com uma *word* (2 *bytes*) em vez de um *byte*, coloca o valor 0xAA55 nos últimos dois *bytes* do binário final, o que torna o dispositivo cujo primeiro setor contém o binário de *bootloader.asm* em um dispositivo bootável.

3.2 Definindo os segmentos e a pilha

A Figura 5 mostra o início do *bootloader* do BergOS.

Figura 5 – Cabeçalho do *bootloader*



Fonte: Elaborado pelo autor (2025)

A primeira linha de *bootloader.asm* define a macro *KERNEL_OFFSET* com o valor 0x7E00, que é o endereço de memória para o qual o *kernel* do BergOS será carregado. A linha 3 informa ao NASM para pôr o código em uma seção chamada *.bootloader*.

O processo de compilação de BergOS, primeiro compila todo o código para o formato *ELF32* e apenas no processo de linkagem é transformado em binário puro. Como o *bootloader* tem que estar no primeiro setor do disco, definir uma seção própria para ele torna possível definir um *script* de linkagem que garanta que ele seja posto logo no começo do binário final. Uma explicação detalhada sobre esse *script* de linkagem pode ser encontrada no Apêndice A. A linha 5 informa ao NASM que um símbolo chamado *main* é externo e deve ser resolvido durante o processo de linkagem. Esse símbolo refere-se à função principal *main*, escrita em linguagem C, que serve como ponto de entrada do *kernel* do BergOS. A linha 7 é uma diretiva que instrui o NASM a montar as instruções subsequentes no formato de 16 *bits*, o que é necessário uma vez que o processador é inicializado no modo real.

O início do programa do *bootloader* se dá pelo rótulo *set_segmentation*. Esse rótulo, assim como outros, não é necessário e não será usado em instruções de desvio de fluxo como *jmp* ou *call*, ele serve apenas para tornar o código *assembly* mais estruturado e fácil de compreender. A Figura 6 mostra o código de *set_segmentation*.

Figura 6 – Definição dos registradores de segmento no *bootloader*

```
7   [bits 16]
8   set_segmentation:
9     xor ax, ax
10    mov ds, ax
11    mov es, ax
12    mov ss, ax
13
```

Fonte: Elaborado pelo autor (2025)

A instrução *xor*, que representa a operação “ou exclusivo”, recebe dois operandos, um registrador e um valor que pode vir de outro registrador ou da memória, aplica a operação e armazena o resultado no registrador do primeiro operando. A operação de “ou exclusivo” quando aplicada a valores iguais resulta em zero, portanto

a instrução serve para zerar o valor de AX. O mesmo resultado poderia ser obtido com “mov ax, 0”, porém essa instrução, junto ao operando, ocupa 3 *bytes*, enquanto a instrução equivalente com *xor* ocupa 2 *bytes*. É uma prática comum em programação *assembly* poupar *bytes* com instruções mais econômicas. Prática inteligente de se seguir, já que o programa do *bootloader* está limitado a 512 *bytes*. Com o valor de AX zerado ele é usado para definir todos os registradores de segmento como zero.

Depois de definir os registradores de segmento, os registradores de pilha são configurados no rótulo *set_stack*. O programa faz os registradores BP e SP apontar para o endereço 0x7C00. A pilha “cresce para baixo”, o que significa que *push* (instrução que coloca um valor na pilha) subtrai o valor de SP enquanto *pop* (instrução que remove um valor da pilha) soma o valor de SP, o que garante que a pilha não sobrescreverá o programa do *bootloader* durante a execução. A Figura 7 mostra *set_stack*.

Figura 7 – Definindo a pilha

```
13
14  set_stack:
15      mov bp, 0x7c00
16      mov sp, bp
17
```

Fonte: Elaborado pelo autor (2025)

Após definir a pilha, no rótulo *set_video_mode* o *bootloader* executa uma interrupção para acessar uma função de vídeo do BIOS chamada *Set Mode*, com a finalidade de alterar o modo de vídeo para 3. Isso será útil para o *driver* de VGA do BergOS, que depende que o modo de vídeo seja este. O código de *set_video_mode* será explicado no capítulo 4.

3.3 Carregando o kernel para a memória

Os primeiros 512 *bytes* do binário de BergOS são reservados para o *bootloader*. Nos *bytes* seguintes fica localizado o *kernel*. O *kernel* do BergOS está, portanto, a partir do segundo setor do disco no qual o *bootloader* foi executado.

Há várias maneiras de ler um disco. Umas mais antigas e outras mais modernas.

> Historicamente, o endereçamento dos blocos usava um padrão denominado CHS (Cylinder-Head-Sector): para acessar cada bloco, era necessário informar a cabeça (ou seja, a face), o cilindro (trilha) e o setor do disco onde se encontra o bloco. Esse sistema foi mais tarde substituído pelo padrão LBA (Logical Block Addressing), no qual os blocos são endereçados linearmente (0, 1, 2, 3, ...), o que é muito mais fácil de gerenciar pelo sistema operacional. Como a estrutura física do disco rígido continua a ter faces, trilhas e setores, uma conversão entre endereços LBA e CHS é feita pelo firmware do disco rígido, de forma transparente para o restante do sistema (MAZIERO, p. 262).

As funções BIOS suportavam, originalmente, apenas o padrão *Cylinder-Head-Sector* (CHS). Posteriormente algumas implementações começaram a fornecer extensões para suportar *Logical Block Addressing* (LBA). A Phoenix Technologies formalizou, em 1994, um padrão chamado de *Enhanced Disk Drive*, que expande as capacidades do serviço de disco, acessadas pela interrupção 0x13, para lidar com padrões de disco mais modernos, incluindo LBA. BergOS usa funções disponíveis nessa extensão para ler o disco e carregar o *kernel* para a memória.

O rótulo *read_kernel* identifica o código responsável pela leitura do *kernel*:

Figura 8 – Carregando o *kernel* para a memória

```
22
23  read_kernel:
24    mov ah, 0x42
25    mov si, DAP
26    int 0x13
27
28    jnc load_gdt
29
30    mov ah, 0x13
31    mov al, 0x01
32    mov bp, ERROR_KERNEL
33    mov cx, ERROR_KERNEL_LEN
34    mov bl, 0x0f
35    xor dx, dx
36    int 0x10
37
38    cli
39    hlt
40
```

Fonte: Elaborado pelo autor (2025)

Na linha 24, o valor 0x42 é posto em AH, o código que identifica a função BIOS *Extended Read*. A função também exige que o número do dispositivo que ela deve ler seja posto no registrador DL, porém, como o BIOS já inicializa esse registrador com o valor que identifica o disco em que o *boot* ocorreu, não é necessário alterá-lo.

Na linha 25, o registrador *SI* recebe um endereço de memória onde está contida uma estrutura chamada *Disk Address Packet* (DAP). Por fim, uma interrupção 0x13 é disparada para invocar a função BIOS.

Caso um erro ocorra, a *flag carry* do registrador *FLAGS* será ligada (definida para 1). A instrução *jnc* faz um salto para o endereço especificado se a *flag carry* não estiver definida. Ou seja, se nenhum erro ocorreu e a leitura do *kernel* foi concluída, o programa salta para o rótulo *load_gdt*, caso contrário, as linhas 30 a 39 são responsáveis por imprimir uma mensagem de erro e parar a execução do processador.

Embora a função pareça simples, os parâmetros mais complexos são definidos no DAP, e não em registradores. É nele onde será especificado quantos blocos serão lidos, a partir de qual bloco será lido e em qual endereço o conteúdo lido será colocado. Segundo a Phoenix Technologies “A estrutura de dados fundamental para as extensões Int 0x13 é o *Disk Address Packet*. Int 0x13 converte as informações de endereçamento do *Disk Address Packet* para parâmetros físicos apropriados para a mídia.” (1995, p. 7). A Tabela 3 descreve os campos do DAP.

Tabela 3 – Disk Address Packet

Offset	Tipo	Descrição
0	Byte	Tamanho do DAP em bytes. Deve ser maior ou igual a 16.
1	Byte	Reservado, deve ser 0.
2	Byte	Número de blocos para transferir.
3	Byte	Reservado, deve ser 0.
4	Double Word (4 bytes)	O endereço, no padrão segmento:offset, em que as operações de escrita/leitura

		serão realizadas.
8	Quad Word (8 bytes)	Número do primeiro bloco, no padrão LBA, em que as operações de escrita/leitura serão realizadas.

Fonte: Elaborado pelo autor (2025).

O DAP usado pelo *bootloader* do BergOS é definido no rótulo DAP. A Figura 9 mostra a definição do DAP para a leitura do *kernel*.

Figura 9 – Definição do DAP

```
58
59  DAP:
60    db 0x10           ; Tamanho do DAP. Deve ser 16 (0x10).
61    db 0x00           ; Reservado.
62    dw 0x0010         ; Deve ler 16 (0x10) blocos.
63    dw KERNEL_OFFSET  ; Deve ler para o offset definido por ENTRY_OFFSET.
64    dw 0x0000         ; Deve ler para o segmento 0.
65    dd 0x0001         ; Deve ler a partir do bloco 1.
66    dd 0x0000         ; Se junta com o campo anterior.
67  DAP.end:
68
```

Fonte: Elaborado pelo autor (2025)

Juntando o segmento com o *offset*, o endereço para onde o *kernel* será lido é 0x0000:*KERNEL_OFFSET*, que, com a expansão da macro *KERNEL_OFFSET*, resulta em 0x0000:0x7E00.

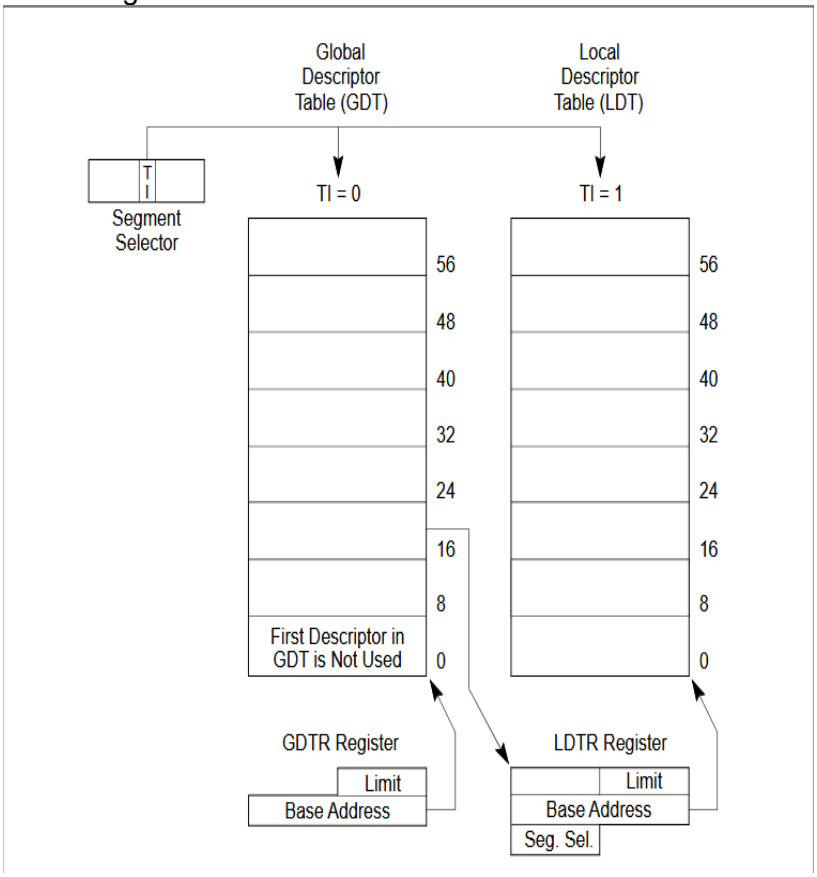
3.4 Colocando o processador em modo protegido

Colocar o processador em modo protegido é simples, bastando apenas mudar um *bit* em um registrador. Porém, para que ele funcione apropriadamente, uma GDT deve ser definida. Uma GDT é uma tabela de descritores de segmentos que, segundo a Intel Corporation, "[*…*] é um *array* de descritores de segmento. Uma tabela de descritores é variável em tamanho e pode conter até 8192 (2^{13}) descritores de 8 *bytes*." (2025, Vol. 3A 3-14, tradução nossa). No modo protegido, diferentemente do modo real, os registradores de segmento deixam de ser usados diretamente no cálculo de endereços e passam a atuar como índices que selecionam um descritor em uma

tabela de descritores de segmentos. Esses descritores descrevem atributos do segmento como endereço base, nível de privilégio, tipo do segmento (código, data) dentre outros.

Há outras tabelas de descritores na arquitetura x86 como a IDT, fundamental para tornar a arquitetura multitarefa, já que é ela a responsável por lidar com as rotinas de tratamento de interrupção; e a *Local Descriptor Table* (LDT) que é muito semelhante a uma GDT e tinha o propósito de auxiliar os sistemas operacionais na alternância de tarefas, mas os projetistas de sistemas operacionais não a adotaram. Para o modo protegido, apenas a GDT é necessária. A Figura 10 mostra a estrutura de uma GDT e uma LDT.

Figura 10 – A estrutura de uma GDT e uma IDT



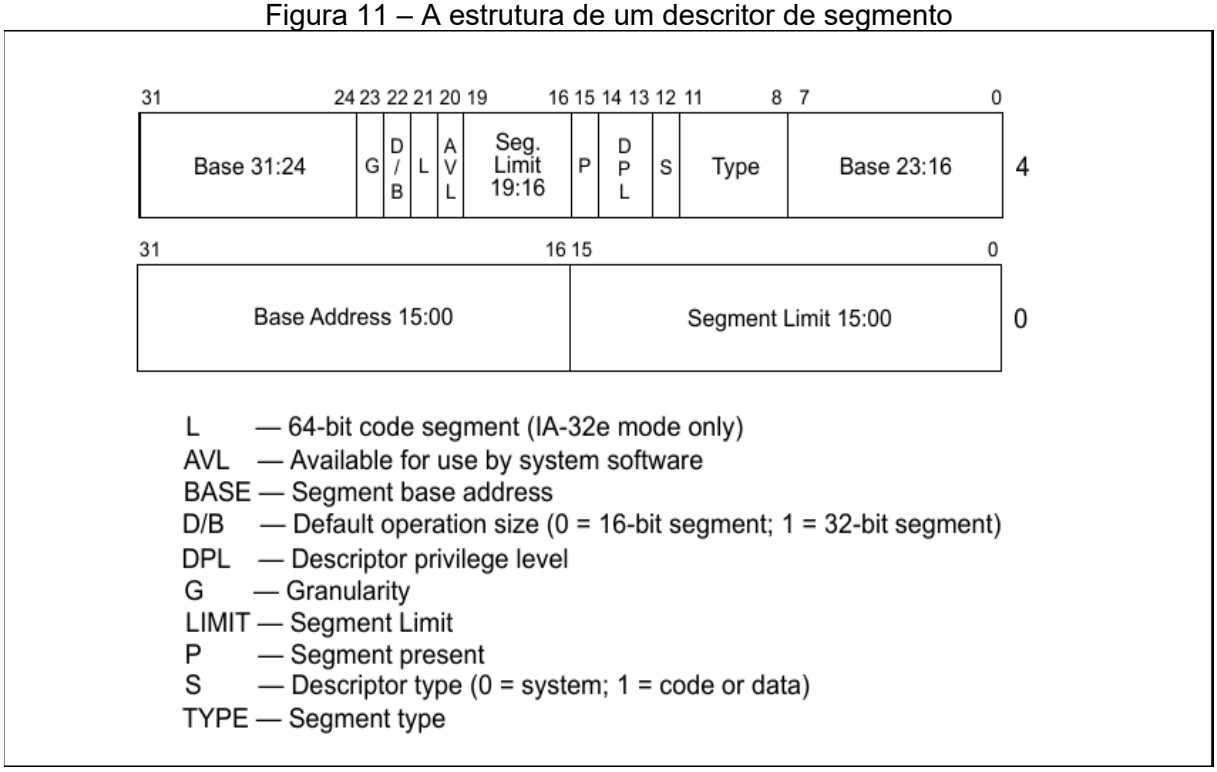
Fonte: Intel Corporation (2025)

Cada entrada de uma GDT é um descritor de segmento e ocupa 8 bytes. O primeiro descritor de uma GDT não é usado. Os registradores de segmento CS, DS,

ES, FS, GS e SS armazenam um valor que serve como um índice para obter o descritor de segmento correspondente na GDT.

A Figura 11 mostra a estrutura de um descritor de segmento.

Figura 11 – A estrutura de um descritor de segmento



Fonte: Intel Corporation (2025)

A estrutura de um descritor de segmento é composta por campos distribuídos de maneira não contígua ao longo de 64 *bits*. Essa disposição caótica e confusa existe para manter compatibilidade com versões antigas da arquitetura x86. Os campos da GDT desempenham o seguinte papel:

- ***Base Address***: O endereço base é definido por três campos fragmentados: “Base 31:24”. “Base 23:16” e “Base Address 15:00”, que, em conjunto, formam um valor de 32 *bits* que indica o início do segmento.

- ***Segment Limit***: O limite de segmento é determinado por dois campos: “Seg. Limit 19:16” e “Segment Limit 15:00”, que compõem um valor de 20 *bits*. Esse valor define o tamanho do segmento. Se o *bit* de granularidade (G) estiver definido como 0, o limite é calculado em incrementos de *bytes*, permitindo

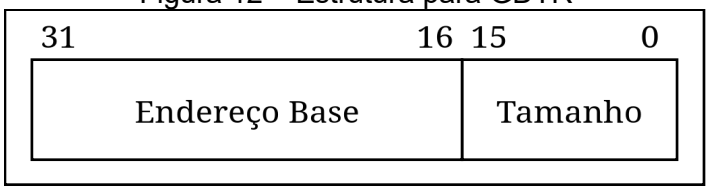
segmentos de até 1 MB (2^{20}). Se G for 1, o limite é calculado em incrementos de 4 KB, possibilitando segmentos de até 4 GB (2^{20} * 4 KB).

- **P (s*egment present*)**: Indica se o segmento está presente na memória.
- **DPL (*descriptor privilege level*)**: Define o nível de privilégio necessário para acessar o segmento. Pode assumir valor de 0 a 3, sendo 0 o mais privilegiado e 3 o menos privilegiado.
- **S (*descriptor Type*)**: Se for 0 indica que o segmento é um segmento de sistema, se for 1 indica que é um segmento de código ou dados.
- **TYPE**: Se o valor de S for 1, este campo serve para selecionar entre um segmento de código ou segmento de dados e definir suas características.
- **D/B (*default operation size*)**: Se for 0, o segmento é tratado como um segmento de 16 *bits*, se for 1, o segmento é tratado como um segmento de 32 *bits*.
- **G (*granularity*)**: Determina a escala do campo *Segment Limit*.
- **L (*64-bit code segment*)**: Disponível apenas em plataformas que suportam *IA-32e*.
- **AVL (a*vailable and reserved bits*)**: Disponível para ser usado pelo sistema.

A GDT desempenha um papel fundamental na separação do espaço de usuário e do espaço de *kernel*. O campo DPL é usado para definir os níveis de privilégio de um segmento. Apesar desse campo poder assumir 4 valores diferentes, historicamente os sistemas operacionais fazem uso de apenas dois deles, com o nível 0 sendo usado para o espaço de *kernel* e o nível 3 sendo usado para o espaço de usuário.

Para que o processador passe a usar a GDT definida, seu endereço e tamanho precisam ser carregados para um registrador especial chamado GDTR (INTEL CORPORATION, 2025, Vol. 3A 3-1). Para isso, outra estrutura na memória será necessária. A Figura 12 ilustra essa estrutura.

Figura 12 – Estrutura para GDTR



Fonte: Elaborado pelo autor (2025)

A estrutura é autoexplicativa, mas um detalhe importante é que o campo “Tamanho” deve conter o tamanho real da GDT subtraído por 1. Isso decorre do fato de que 16 *bits* podem representar números em um intervalo de 0 a 65535 (2^{16} – 1). Porém, como não há GDT com tamanho de zero *bytes*, o valor 0 representa uma GDT de 1 *byte*, o valor 1 uma GDT de 2 *bytes* e assim sucessivamente.

A Figura 13 mostra a GDT usada pelo *bootloader* do BergOS, com a definição da estrutura a ser carregada na GDTR e dos descritores de segmento.

Figura 13 – Definição da GDT usada pelo *bootloader* do BergOS

```
68
69  GDT:
70    dw .end - .begin - 1
71    dd .begin
72    .begin:
73      .null: db 0x00, 0x00, 0x00, 0x00, 0x00, 0b00000000, 0b00000000, 0x00
74      .code: db 0xff, 0xff, 0x00, 0x00, 0x00, 0b10011010, 0b11001111, 0x00
75      .data: db 0xff, 0xff, 0x00, 0x00, 0x00, 0b10010010, 0b11001111, 0x00
76  GDT.end:
77
```

Fonte: Elaborado pelo autor (2025)

O rótulo GDT demarca o início das estruturas referentes a GDT. Na linha 70, a pseudo-instrução *dw* é usada para definir o valor de 16 *bits* que representa o tamanho da GDT. Um cálculo com endereços é feito para obter o tamanho da GDT subtraído por 1. Enquanto a linha 71 usa a pseudo-instrução *dd* para um valor de 32 *bits* que representa o endereço da GDT.

O rótulo local *.begin* marca o início da GDT propriamente dita. Como BergOS não faz uso de segmentação, por ser um mecanismo obsoleto, e nem faz separação

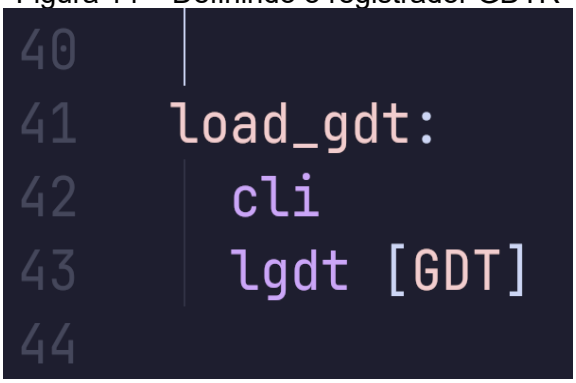
do espaço de *kernel* e espaço de usuário, os segmentos definidos na GDT são bem simples.

A GDT usada pelo *bootloader* do BergOS define apenas dois segmentos válidos: um para código e outro para dados. A única diferença entre eles é um *bit* que determina qual é o segmento de código e qual é o segmento de dados. De resto, ambos têm as mesmas características: endereço base igual a 0, limite do segmento igual a 0xFFFFF (tamanho máximo), nível de privilégio igual a 0 (mais privilegiado).

Como o primeiro descritor de uma GDT não é usado, o rótulo local *.null* preenche essa entrada com zeros. Já os rótulos locais *.code* e *.data* definem os descritores de segmento de código e de dados, respectivamente.

Após a definição dessa estrutura, o registrador GDTR pode finalmente ser carregado com o endereço dela. Isso é feito no rótulo *load_gdt* que foi para onde o programa do *bootloader* saltou após carregar o *kernel*. A Figura 14 mostra o rótulo *load_gdt.*

Figura 14 – Definindo o registrador GDTR

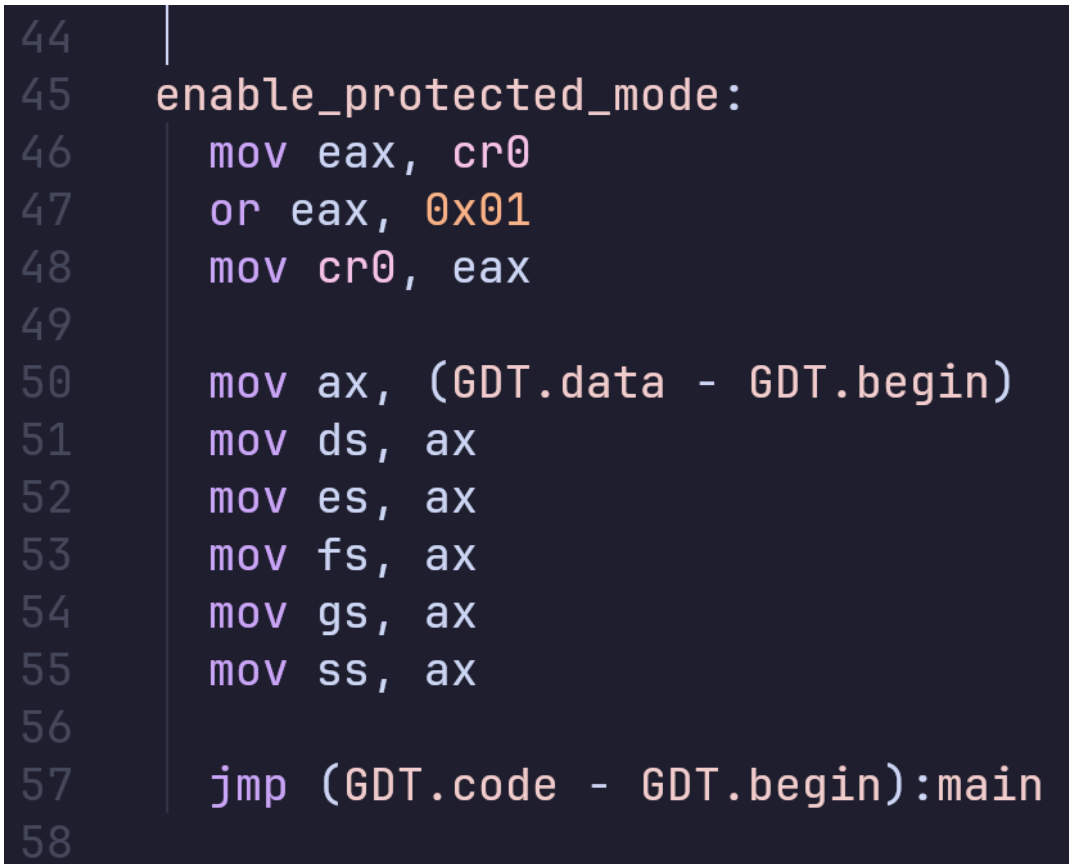


Fonte: Elaborado pelo autor (2025)

Primeiramente, as interrupções são desligadas com a instrução *cli*, depois a estrutura é carregada para o registrador GDTR através da instrução *lgdt*. Desligar as interrupções serve tanto para evitar comportamentos indesejados quanto para cumprir o contrato entre o *bootloader* e o *kernel* onde o primeiro deve entregar o controle da máquina para o segundo com as interrupções desligadas.

A Figura 15 mostra o rótulo *enable_protected_mode*, onde o processador finalmente é posto em modo protegido.

Figura 15 – Colocando o processador em modo protegido e passando o controle para o kernel



```
44
45  enable_protected_mode:
46      mov eax, cr0
47      or eax, 0x01
48      mov cr0, eax
49
50      mov ax, (GDT.data - GDT.begin)
51      mov ds, ax
52      mov es, ax
53      mov fs, ax
54      mov gs, ax
55      mov ss, ax
56
57      jmp (GDT.code - GDT.begin):main
58
```

Fonte: Elaborado pelo autor (2025)

Nas linhas 46, 47 e 48, o valor do registrador de controle *CR0* é copiado para EAX (a versão estendida de 32 *bits* de AX), uma instrução *or*, que representa a operação “ou inclusivo”, é usada para ativar o primeiro *bit* do registrador para, na operação seguinte, colocar o valor de volta em CR0. O primeiro *bit* de CR0 determina se o processador está em modo protegido ou não, portanto, ao ativá-lo, o processador está definitivamente em modo protegido.

A linha 50 faz um cálculo para obter o índice do segmento de dados, definido na GDT, para, nas linhas 51 a 55, fazer os registradores de segmento DS, ES, FS, GS e SS usarem o mesmo segmento de dados.

A linha 57 é importante por duas razões: a primeira é que ela altera o segmento de código, a segunda é que ela é responsável por realizar o salto para o *kernel* do BergOS. O valor de CS não pode ser alterado com uma instrução *mov* como os outros. Portanto, para alterar o segmento de código é necessário alguma instrução de desvio de fluxo. A instrução *jmp* permite alterar o valor de CS especificando o valor do segmento antes do endereço em si.

O segmento pode ser um dos registradores de segmento ou um valor imediato. A linha 57 faz um *jmp* para o símbolo *main*, que é a função principal do *kernel* do BergOS, alterando o valor de CS para o índice do segmento de código, definido na GDT.

A função de entrada do BergOS, *main*, é definida no arquivo *./kernel/main.c*, como mostra a Figura 16.

Figura 16 – Função *main* do kernel do BergOS

```
1   #include "kernel.h"
2   #include "tty.h"
3
4   void main(void) {
5     kernel_initialize();
6
7     tty_initialize();
8
9     tty_printf("Hello, world!\n");
10    tty_printf("I am BergOS!\n");
11
12    kernel_halt();
13  }
```

Fonte: Elaborado pelo autor (2025)

É uma boa prática em desenvolvimento de sistemas operacionais criar uma camada de código que é mais baixo nível, com rotinas que dependem de recursos específicos de uma arquitetura, e uma camada mais alto nível que tenta ser o mais adequadamente independente. Assim, caso os desenvolvedores queiram portar o sistema para uma outra arquitetura, apenas as rotinas de baixo nível precisariam ser reescritas. Isso promove uma programação baseada em interfaces, onde o código de alto nível invoca rotinas de baixo nível sem se importar com a implementação delas.

No repositório do BergOS, todo código da camada de alto nível está no diretório *./kernel/*, e todo código da camada de baixo nível está no diretório *./arch/*. Dentro de *./arch*, há outros diretórios, cada um se referindo à implementação de uma arquitetura específica. Por exemplo, a implementação para arquitetura *i386*, a estudada neste trabalho, está em *./arch/i386/*.

A linha 5 chama a função de baixo nível *kernel_initialize*, que é definida no cabeçalho *./kernel/include/kernel.h*. Ela é responsável por fazer quaisquer configurações e inicializações necessárias para o funcionamento do *kernel*. Sua implementação está em *./arch/i386/kernel.c*. No caso da implementação para *i386,* a função configura uma GDT para ser usada pelo *kernel*, já que a definida pelo *bootloader* foi apenas uma necessidade para colocar o processador em modo protegido. Apesar da GDT configurada por *kernel_initialize* ser, atualmente, idêntica à do *bootloader*, é uma boa prática torná-las independentes, já que caso futuramente o BergOS venha a ter separação entre espaço de *kernel* e espaço de usuário, a funcionalidade poderia ser implementada facilmente sem fazer com que o *bootloader* perca sua simplicidade.

Na linha 7, a função de baixo nível *tty_initialize* é chamada. Ela é responsável por inicializar o emulador de terminal, que será usado pelo *kernel* para realizar saída de dados.

As linhas 9 e 10 usam a função *tty_printf* para escrever *strings* na tela que, juntas, formam a mensagem “*Hello, world! I am BergOS*”. Por fim, na linha 12, como a função *main* não deve retornar, é chamada a função de baixo nível *kernel_halt* que para a execução do processador.

A função *main* apresenta todo o comportamento visível do *kernel* do BergOS para o usuário que executa o sistema. A Figura 17 mostra a execução do BergOS no emulador QEMU.

Figura 17 – Execução de BergOS

Fonte: Elaborado pelo autor (2025)

O próximo capítulo apresenta o padrão VGA, como ele pode ser usado para se comunicar com o dispositivo de vídeo e termina com um exame detalhado do *driver* de VGA usado pelo BergOS.

4 DRIVER DE VGA

Computação gráfica é um tópico complexo e extenso que se tornou fundamental na computação. Ferramentas modernas, como Cuda, facilitam o trabalho dos programadores fornecendo camadas de abstração. Porém, décadas atrás os programadores não tinham esse luxo e eram obrigados a lidar com interfaces espartanas e problemas de portabilidade.

Um avanço importante foi feito com a introdução do padrão VGA, que, segundo Wilson "[…] é um padrão de exibição de vídeo e um tipo de conexão amplamente utilizado na indústria de computadores há décadas. Introduzido pela IBM em 1987, o VGA rapidamente se tornou o padrão gráfico para PCs e lançou as bases para os monitores de computador modernos." (2024, tradução nossa).

Apesar de ser um padrão antigo, ele ainda é suportado pela maioria dos dispositivos de vídeo modernos. Portanto é uma boa ideia ter um *driver* simples de VGA para ter suporte a vídeo logo no estágio inicial de desenvolvimento de um sistema operacional, com a segurança de que provavelmente ele funcionará em qualquer hardware.

Mesmo que o padrão VGA permita uma resolução de 640 x 480 e tenha suporte a 256 cores (WILSON, 2024), nenhum desses recursos é usado no *driver* do BergOS. Na verdade, o padrão VGA tem suporte a vários modos de vídeo, incluindo os antigos modos que surgiram nos PCs da IBM anteriores ao PS/2, onde o padrão VGA foi introduzido. A Figura 18 mostra os modos de vídeo disponíveis para um IBM PS/2.

Figura 18 – Modos de vídeo disponíveis

Mode (Hex)	Type	Maximum Colors	Alpha Format	Buffer Start
0, 1	A/N	16	40x25	B8000
2, 3	A/N	16	80x25	B8000
4, 5	APA	4	40x25	B8000
6	APA	2	80x25	B8000
7	A/N	Mono	80x25	B0000
8	APA	16	20x25	B0000
9	APA	16	40x25	B0000
A	APA	4	80x25	B0000
B, C	—Reserved—			
D	APA	16	40x25	A0000
E	APA	16	80x25	A0000
F	APA	Mono	80x25	A0000
10	APA	16	80x25	A0000
11	APA	2	80x30	A0000
12	APA	16	80x30	A0000
13	APA	256	40x25	A0000

APA — All Points Addressable (Graphics)
A/N — Alphanumeric (Text)

Fonte: IBM (1987)

Os modos de vídeo variam entre *All Points Addressable* (APA), que permitem uma manipulação gráfica através de pixels, e *Alphanumeric* (A/N), onde a manipulação gráfica ocorre através de caracteres. O *driver* de VGA do BergOS utiliza o modo 3, que permite a escrita de caracteres em uma matriz 80 x 25 com suporte a 16 cores.

Ainda que a maioria das implementações do BIOS já inicialize com o modo de vídeo 3, o *bootloader* do BergOS garante que a máquina esteja nesse modo utilizando a função BIOS *Set Mode*. Isso é feito antes do *kernel* ser carregado, como mostra a Figura 19.

Figura 19 – Definindo o modo de vídeo para 3

```
17
18    set_video_mode:
19        mov ah, 0x00
20        mov al, 0x03
21        int 0x10
22
```

Fonte: Elaborado pelo autor (2025)

O valor 0, que representa a função *Set Mode* nos serviços de vídeo, é posto em AH. Logo em seguida o valor 3 é posto em AL, o modo de vídeo desejado, e então uma interrupção de vídeo é invocada, garantindo que o modo de vídeo seja 3.

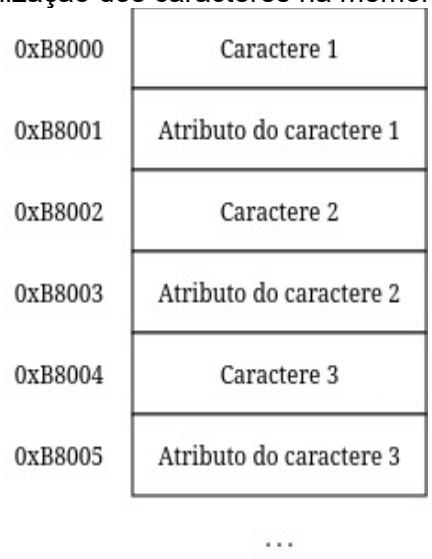
O interessante do padrão VGA é que ele usa E/S mapeada na memória, onde para escrever um caractere na tela em algum modo alfanumérico, basta colocar o código do caractere em um endereço de memória comum. O endereço de memória mapeado depende do modo de vídeo utilizado. No modo 3, os endereços vão de 0xB8000 a 0xBFFFF (FERRARO, 1994, p. 181).

De acordo com Ferraro “Nos modos alfanuméricos, os códigos que representam o caractere e o atributo do caractere são armazenados na memória. Um único byte é dedicado a cada código de caractere, permitindo o acesso a 256

caracteres. Um único byte também é dedicado ao atributo do caractere.” (FERRARO, 1994, 181).

O byte do caractere é um endereço par e o atributo desse caractere é o endereço ímpar seguinte. A Figura 20 ilustra essa ideia.

Figura 20 – Organização dos caracteres na memória no padrão VGA

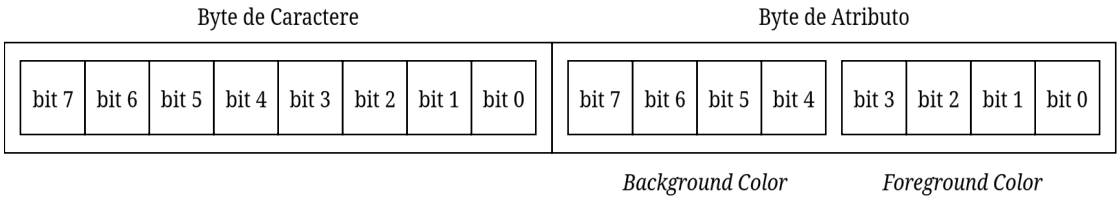


Fonte: Elaborado pelo autor (2025)

Os modos alfanuméricos também possuem um sistema de páginas. No entanto, o *driver* do BergOS não utiliza esse sistema, escrevendo apenas na página 0. Sendo assim, há espaço para 80 x 25 (2000) caracteres no emulador de terminal.

O byte de atributo pode ser dividido em um par de 4 *bits* cada. Os 4 *bits* menos significativos são usados para determinar a *foreground color* (cor do caractere), e os 4 *bits* mais significativos são usados para determinar a *background color* (cor de fundo). A Figura 21 mostra a representação de um *byte* de atributo juntamente a um *byte* de caractere.

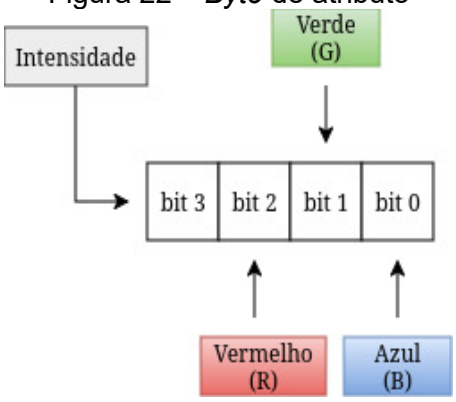
Figura 21 – *Byte* de caractere e *byte* de atributo



Fonte: Elaborado pelo autor (2025)

Como uma cor é definida por 4 *bits*, esse modo de vídeo tem 2^4 (16) cores disponíveis. As cores são definidas através de um padrão RGB. A Figura 22 mostra um *byte* de atributo demarcando esse padrão.

Figura 22 – *Byte* de atributo



Fonte: Elaborado pelo autor (2025)

A Tabela 4 mostra todas as combinações possíveis de cores.

Tabela 4 – Combinações de cores RGB possíveis em 4 *bits*

Código RGB	Cor
0000	Preto
0001	Azul
0010	Verde
0011	Ciano
0100	Vermelho
0101	Magenta
0110	Marrom
0111	Branco
1000	Cinza
1001	Azul claro
1010	Verde claro
1011	Ciano claro
1100	Vermelho claro
1101	Magenta claro
1110	Amarelo (Marrom claro)
1111	Branco brilhante

Fonte: Elaborado pelo autor (2025).

Em alguns casos, os *bits* 3 e 7 do *byte* de atributo assumem significados especiais. No entanto, essas funcionalidades não são consideradas na implementação do BergOS.

4.1 Definição da interface do *driver* de VGA

A interface do *driver* de VGA é definida no arquivo de cabeçalho *./arch/i386/video/vga/vga.h*. A Figura 23 mostra o conteúdo desse arquivo.

Figura 23 – Definição da interface do *driver* de VGA

```
1  #ifndef VGA_H
2  #define VGA_H
3  |
4  #define VGA_MAXY 25
5  #define VGA_MAXX 80
6  |
7  typedef enum {
8  |  VGA_COLOR_BLACK         = 0b0000,
9  |  VGA_COLOR_BLUE          = 0b0001,
10 |  VGA_COLOR_GREEN         = 0b0010,
11 |  VGA_COLOR_CYAN          = 0b0011,
12 |  VGA_COLOR_RED           = 0b0100,
13 |  VGA_COLOR_MAGENTA       = 0b0101,
14 |  VGA_COLOR_BROWN         = 0b0110,
15 |  VGA_COLOR_WHITE         = 0b0111,
16 |  VGA_COLOR_GRAY          = 0b1000,
17 |  VGA_COLOR_LIGHT_BLUE    = 0b1001,
18 |  VGA_COLOR_LIGHT_GREEN   = 0b1010,
19 |  VGA_COLOR_LIGHT_CYAN    = 0b1011,
20 |  VGA_COLOR_LIGHT_RED     = 0b1100,
21 |  VGA_COLOR_LIGHT_MAGENTA = 0b1101,
22 |  VGA_COLOR_YELLOW        = 0b1110,
23 |  VGA_COLOR_BRIGHT_WHITE  = 0b1111,
24 } VGAColor;
25 |
26 int vga_write(int index, char character, VGAColor foreground, VGAColor background);
27 int vga_read(int index, char *character, VGAColor *foreground, VGAColor *background);
28 |
29 #endif
```

Fonte: Elaborado pelo autor (2025)

As linhas 4 e 5 definem macros que se referem às dimensões da tela. Conforme mencionado, no modo de vídeo utilizado há uma matriz de 80 x 25. Portanto, a constante *VGA_MAXY* é definida como 25 e a constante *VGA_MAXX* é definida como 80.

A interface também define o tipo *enum VGAColor*. Esse *enum* possui constantes que representam todas as cores disponíveis, as quais podem ser utilizadas como argumentos para as rotinas do *driver* sempre que uma cor precisar ser especificada.

Na linguagem C, *enums* funcionam essencialmente como syntactic *sugars* para um *int*. Isso significa que qualquer valor *int* válido é um *VGAColor* válido, mesmo que não corresponda a nenhuma das constantes de cor definidas.

Portanto, a definição do tipo *VGAColor* não proporciona segurança de tipos, já que um valor inválido (diferente das constantes de cor pré-definidas) pode ser atribuído a uma variável desse tipo. Isso faz com que as rotinas do *driver* tenham que validar os argumentos passados.

Ainda assim, a definição de *VGAColor* é vantajosa por tornar a interface mais autoexplicativa. O programador, ao se deparar com uma rotina com um parâmetro do tipo *VGAColor*, entende que deve usar uma das constantes de cor definidas, melhorando a usabilidade da interface.

A linha 26 declara a rotina *vga_write* para escrita de caracteres. Ela retorna um valor diferente de zero em caso de erro e tem os seguintes parâmetros:

- O índice (posição de memória) onde o caractere será escrito.
- O caractere que será escrito.
- A *foreground color* (cor do caractere).
- A *background color* (cor de fundo).

A linha 27 declara a rotina *vga_read* para recuperar informações de um caractere. Ela retorna um valor diferente de zero em caso de erro e tem os seguintes parâmetros:

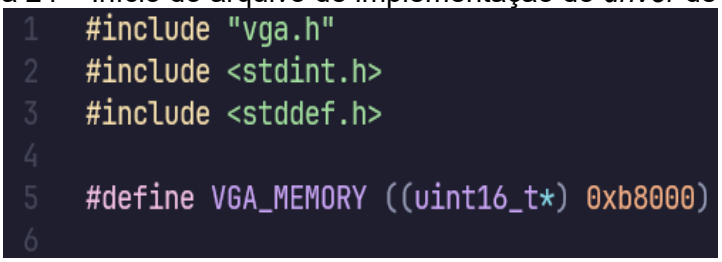
- O índice (posição de memória) do caractere a ser lido.
- Um ponteiro para armazenar o caractere daquele índice.

- Um ponteiro para armazenar a *foreground color* (cor do caractere) do caractere daquele índice.
- Um ponteiro para retornar a *background color* (cor de fundo) do caractere daquele índice.

4.2 Implementação da interface do *driver* de VGA

A implementação da interface do *driver* de VGA está no arquivo *./arch/i386/video/vga/vga.c*. A Figura 24 mostra o início do arquivo.

Figura 24 – Início do arquivo de implementação do *driver* de VGA



Fonte: Elaborado pelo autor (2025)

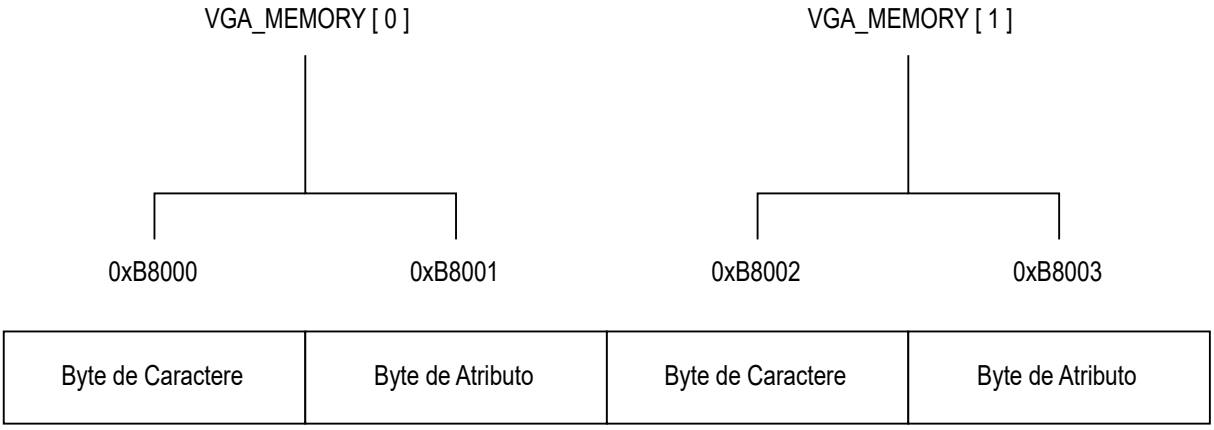
A linha 1 inclui o arquivo de cabeçalho que contém a definição da interface do *driver*. As linhas 2 e 3 incluem cabeçalhos necessários para acessar recursos que serão utilizados na implementação do *driver*, como a constante *NULL* e tipos inteiros de tamanho especificado, como *uint16_t*.

A linha 5 define a macro *VGA_MEMORY* como um ponteiro do tipo *uint16_t* (inteiro sem sinal de 16 *bits*) que aponta para o endereço de memória 0xB8000, local onde se inicia o *buffer* de memória mapeada para vídeo no modo texto.

Esta definição permite utilizar a sintaxe de *arrays* da linguagem C para manipular diretamente o *buffer* de vídeo. Cada posição do *array* acessa uma palavra de 16 *bits* (2 *bytes*) que contém tanto o caractere quanto seus atributos de cor na memória VGA. Dessa forma, operações de leitura e escrita no *buffer* tornam-se mais intuitivas. A Figura 25 ilustra visualmente este conceito, mostrando como cada elemento do *array* corresponde a uma posição específica na tela, armazenando em

uma única palavra de 16 *bits* o código do caractere (*byte* menos significativo) e seus atributos (*byte* mais significativo).

Figura 25 – Acesso do *byte* de caractere e *byte* de atributo através de indexação de *array*



Fonte: Elaborado pelo autor (2025)

4.2.1 Implementação da rotina *vga_write*

A Figura 26 mostra a implementação da rotina *vga_write*.

Figura 26 – Implementação da rotina *vga_write*

```
6
7   int vga_write(int index, char character, VGAColor foreground, VGAColor background) {
8     if (index < 0 || index ⩾ VGA_MAXY * VGA_MAXX) {
9       return 1;
10    }
11
12    VGA_MEMORY[index] = (uint16_t) character | foreground << 8 | (background << 12);
13    return 0;
14  }
15
```

Fonte: Elaborado pelo autor (2025)

Na linha 8 é realizada uma verificação do índice passado como parâmetro. Caso o valor seja menor que zero ou maior ou igual ao limite do *buffer* de vídeo (calculado como 25 linhas por 80 colunas), a função retorna 1. Esse valor, diferente de zero, indica a ocorrência de um erro.

A instrução da linha 12 efetua a escrita do caractere na memória de vídeo, composição que demanda a correta formatação do dado a ser armazenado. A composição é feita através de uma série de operações *bit* a *bit* para construir um *uint16_t*, que forma a unidade fundamental esperada pelo controlador VGA, com o *byte* menos significativo sendo o caractere e o *byte* mais significativo sendo o atributo. A estrutura deste dado é composta da seguinte forma:

- **Composição do caractere:** Os 8 *bits* menos significativos (posições de 0 a 7) são preenchidos diretamente por *character*, após uma conversão explícita para o tipo uint16_t. Esta etapa assegura a correta interpretação do caractere pelo vídeo.

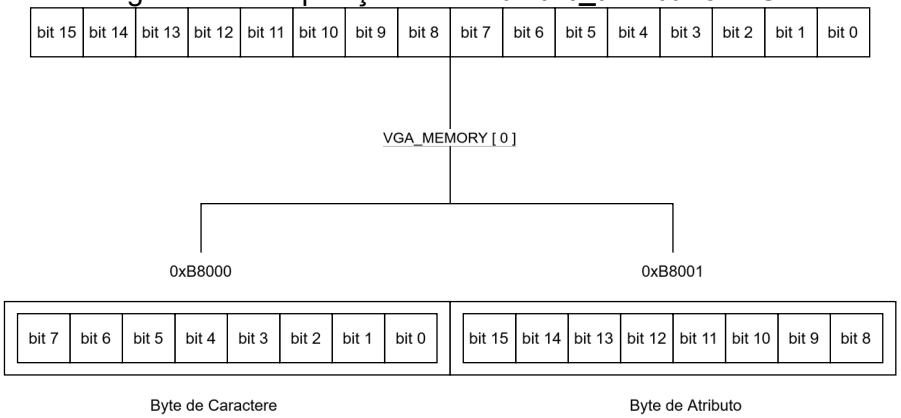
- **Composição da *foreground color*:** Os 4 *bits* subsequentes (posições de 8 a 11) armazenam o código da *foreground color* (cor do caractere). O operador de deslocamento à esquerda (“<< 8”) posiciona este valor nos 4 *bits* menos significativos do *byte* de atributo.

- **Composição da *background color*:** Os 4 *bits* mais significativos (posições de 12 a 15) são reservados para o código da *background color* (cor de fundo). O deslocamento de 12 posições (“<< 12”) garante seu posicionamento nos 4 *bits* mais significativos do *byte* de atributo.

A operação de OU *bit* a *bit* (|) é então utilizada para fundir esses três elementos distintos (caractere, *foreground color* e *background color*) em um único valor de 16 *bits*. Por fim, este valor composto é atribuído à posição de memória solicitada (“*VGA_MEMORY[index]*”), o que resulta na renderização visual do caractere no monitor.

A abordagem de tratar o *buffer* de memória VGA como um *array* de *uint16_t* demonstra-se vantajosa devido à característica *little-endian* da arquitetura x86. Neste padrão, os *bytes* menos significativos são armazenados nas posições de memória iniciais, o que resulta no posicionamento correto do *byte* de caractere seguido pelo *byte* de atributo no formato exigido pelo controlador VGA. Esta disposição é ilustrada na Figura 27.

Figura 27 – Disposição de um *uint16_t* no *buffer* VGA



Fonte: Elaborado pelo autor (2025)

Por fim, a função encerra com o retorno 0, indicando que não houve erros.

4.2.2 Implementação da rotina *vga_read*

A Figura 28 mostra a implementação da rotina *vga_read*.

Figura 28 – Implementação da rotina *vga_read*

```
15
16  int vga_read(int index, char *character, VGAColor *foreground, VGAColor *background) {
17    if (index < 0 || index ⩾ VGA_MAXY * VGA_MAXX) {
18      return 1;
19    }
20
21    if (character ≠ NULL) {
22      *character = (char) VGA_MEMORY[index];
23    }
24    if (foreground ≠ NULL) {
25      *foreground = (VGAColor) VGA_MEMORY[index] >> 8 & 0xf;
26    }
27    if (background ≠ NULL) {
28      *background = (VGAColor) VGA_MEMORY[index] >> 12 & 0xf;
29    }
30
31    return 0;
32  }
```

Fonte: Elaborado pelo autor (2025)

Após a verificação inicial dos limites do índice, na linha 17, a função *vga_read* procede com a extração e decodificação do dado armazenado na posição especificada do *buffer* VGA. Esta operação é inversa à realizada por *vga_write*,

desmontando o valor *uint16_t* em seus componentes originais através de três estruturas condicionais independentes.

Cada condicional verifica se o ponteiro recebido como parâmetro é diferente de *NULL* antes de acessá-lo. Dessa forma, a implementação permite que o chamador recupere seletivamente apenas os componentes de interesse, fornecendo *NULL* para os parâmetros irrelevantes, o que confere flexibilidade à interface da função.

O processo de decodificação ocorre da seguinte forma:

- **Extração do caractere:** O primeiro condicional recupera diretamente o byte menos significativo através de um *cast* para *char*, que corresponde ao código do caractere armazenado nos 8 *bits* menos significativos do *uint16_t*.
- **Extração da *foreground color*:** O segundo condicional realiza o deslocamento à direita de 8 posições (“>> 8”) para posicionar os 4 *bits* da cor do caractere nos *bits* menos significativos, aplicando em seguida uma operação *E bit* a *bit* (“& 0xF”) para isolar exclusivamente estes 4 *bits* e descartar quaisquer outros valores residuais.
- **Extração da *background color*:** O terceiro condicional executa um deslocamento à direita de 12 posições (“>> 12”) para trazer os 4 *bits* da cor de fundo para as posições menos significativas, igualmente aplicando a operação *E bit* a *bit* para garantir que apenas os 4 *bits* relevantes sejam preservados.

Por fim, a função encerra com o retorno 0, indicando que não houve erros.

O próximo capítulo faz uma análise do programa do emulador de terminal do BergOS, e como ele faz uso do *driver* de VGA, descrito neste capítulo, para fazer a saída de dados.

5 EMULADOR DE TERMINAL

Um emulador de terminal é um programa que emula os antigos dispositivos terminais usados para a entrada e saída de dados com o usuário. BergOS implementa um emulador de terminal simples, fornecendo uma interface ao programador para que ele possa escrever caracteres, escrever strings, formatar strings e escrever números inteiros.

5.1 Definição da interface do emulador de terminal

A interface do emulador de terminal é definida na camada de alto nível, no arquivo *./kernel/include/tty.h*. A Figura 29 mostra o conteúdo do arquivo.

Figura 29 – Definição da interface do emulador de terminal

```
#ifndef TTY_H
#define TTY_H

int tty_initialize(void);
int tty_clear(void);

int tty_maxy(void);
int tty_maxx(void);

int tty_gety(void);
int tty_getx(void);

int tty_sety(int y);
int tty_setx(int x);

int tty_iscrlf(void);
int tty_setcrlf(int status);

int tty_putchar(int ch);
int tty_printf(const char *format, ...);

#endif
```

Fonte: Elaborado pelo autor (2025)

A interface declara as seguintes rotinas para manipulação do emulador de terminal:

- ***tty_initialize*:** Inicializa o programa do emulador de terminal. Deve ser chamada antes de qualquer outra. Após uma execução bem-sucedida, todo o conteúdo da tela será apagado, o cursor estará na posição inicial e o terminal estará em modo *LF*. Retorna um valor diferente de zero em caso de erro.

- ***tty_clear*:** Apaga o conteúdo do terminal e volta o cursor para posição inicial. Retorna um valor diferente de zero em caso de erro.

- ***tty_maxy*:** Retorna o valor máximo do eixo y.

- ***tty_maxx*:** Retorna o valor máximo do eixo x.

- ***tty_gety*:** Retorna a posição atual do cursor no eixo y.

- ***tty_getx*:** Retorna a posição atual do cursor no eixo x.

- ***tty_sety*:** Define a posição atual do cursor no eixo y. Retorna um valor diferente de zero em caso de erro.

- ***tty_setx*:** Define a posição atual do cursor no eixo x. Retorna um valor diferente de zero em caso de erro.

- ***tty_iscrlf*:** Retorna um valor diferente de zero caso o modo *CRLF* esteja ativado ou um zero caso esteja desativado.

- **tty_setcrlf:** Ativa o modo *CRLF* se receber um valor diferente de zero ou desativa caso o contrário.

- ***tty_putchar*:** Imprime o caractere recebido como argumento e avança o cursor. Retorna um valor diferente de zero em caso de erro.

- ***tty_printf*:** Semelhante a *printf* da biblioteca padrão da linguagem C. Imprime uma *string*, aplicando os argumentos adicionais aos códigos de formato presentes na *string* e avança o cursor. Caso o código de formato seja inválido ou não corresponda a um argumento válido o comportamento é indefinido. Os códigos de formato são:

    - **“%c”**: Imprime um caractere.

- **“%s”**: Imprime uma *string*.
- **“%d”**: Imprime um inteiro com sinal.
- **“%u”**: Imprime um inteiro sem sinal.
- **“%x”**: Imprime um inteiro sem sinal em formato hexadecimal.
- **“%%”**: Imprime o caractere “%”.

O cursor determina a posição onde será escrito o próximo caractere. Sua posição é atualizada automaticamente por qualquer rotina de impressão de caracteres, mas também pode ser definida manualmente por meio das rotinas *tty_sety* e *tty_setx* ou através de caracteres de controle.

O emulador de terminal reconhece os caracteres de controle *Carriage Return* (CR, representado por ‘\r’ na linguagem C), e *Line Feed* (LF, representado por ‘\n’ na linguagem C). Conceitualmente, *o* CR move o cursor para o início da linha e o LF avança o cursor para a linha seguinte. Contudo, o comportamento efetivo desses caracteres é determinado pelo modo de operação do terminal.

Após a inicialização do terminal com a função *tty_initialize*, o terminal opera no modo LF. Nesta configuração, cada ocorrência do caractere LF não apenas avança o cursor para a próxima linha, mas também o reposiciona para o início dela.

Opcionalmente, o terminal pode operar no modo CRLF, onde é necessária a sequência completa de ambos os caracteres (“\r\n”) para efetuar um avanço de linha completo. Neste modo, o caractere LF ('\n') executa apenas o avanço vertical para a próxima linha, enquanto o CR ('\r') é responsável pelo retorno do cursor ao início da linha horizontal.

A Figura 30 mostra como seria a mensagem de saudações do BergOS caso o modo CRLF fosse ativado.

Figura 30 – Mensagem de saudações do BergOS com o terminal operando em CRLF

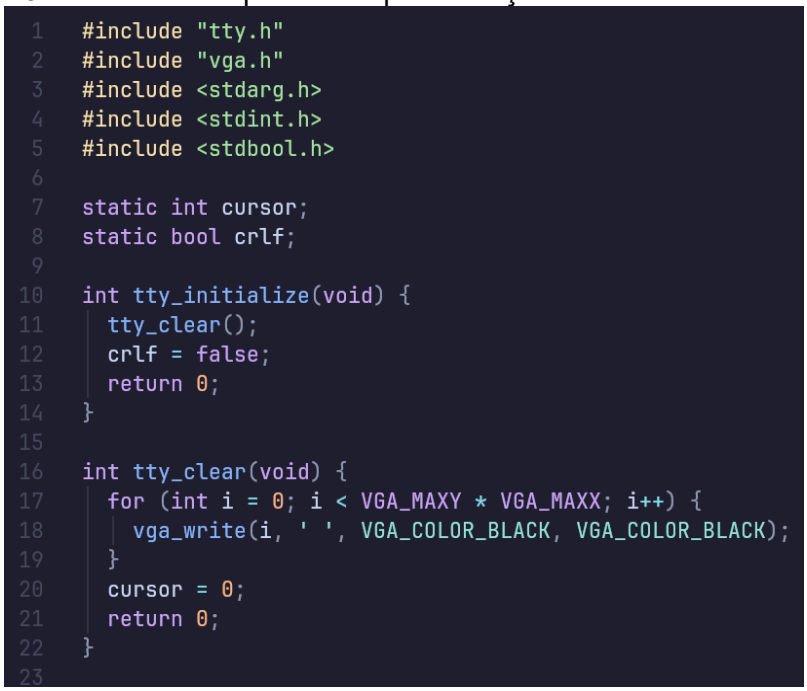
Fonte: Elaborado pelo autor (2025)

Quando a escrita no terminal atinge o limite inferior da tela, o sistema executa uma operação de rolagem vertical. Este mecanismo consiste em “puxar” todo o conteúdo exibido para cima, onde cada linha é movida para a posição imediatamente superior. Especificamente, o conteúdo original da primeira linha é descartado, o da segunda linha passa a ocupar a primeira, o da terceira linha move-se para a segunda, e este processo se repete sequencialmente até que a última linha do terminal seja liberada para receber novos caracteres.

5.2 Implementação da interface do emulador de terminal

A implementação da interface do emulador de terminal está no arquivo *./arch/i386/tty.c*. A Figura 31 mostra o início do arquivo de implementação.

Figura 31 – Início do arquivo de implementação do emulador de terminal



```c
#include "tty.h"
#include "vga.h"
#include <stdarg.h>
#include <stdint.h>
#include <stdbool.h>

static int cursor;
static bool crlf;

int tty_initialize(void) {
  tty_clear();
  crlf = false;
  return 0;
}

int tty_clear(void) {
  for (int i = 0; i < VGA_MAXY * VGA_MAXX; i++) {
    vga_write(i, ' ', VGA_COLOR_BLACK, VGA_COLOR_BLACK);
  }
  cursor = 0;
  return 0;
}

```

Fonte: Elaborado pelo autor (2025)

Nas linhas 1 a 5, são incluídos arquivos de cabeçalho que fornecem os recursos necessários para a implementação, entre os quais a interface do *driver* de VGA.

Na linha 7, é declarada uma variável global do tipo *int* acessível em todo o arquivo, que será utilizada para controlar a posição do cursor. Na linha 8, é definida uma variável global do tipo *bool* que indica se o modo de operação CRLF está ativado.

Em C, a palavra-chave *static* assume significados distintos conforme o contexto. Por padrão, identificadores de escopo de arquivo possuem vinculação externa, podendo ser referenciados por outros arquivos durante o processo de linkagem. É por conta desse mecanismo, por exemplo, que a função *main* do *kernel* é visível para o *bootloader*.

No entanto, nem sempre é desejável expor um identificador. Para evitar que tais identificadores sejam acessados externamente, utiliza-se a palavra-chave *static*. Quando aplicada a variáveis ou funções em escopo de arquivo, *static* altera sua vinculação para interna, limitando sua visibilidade exclusivamente ao arquivo onde foram definidas. Dessa forma, as variáveis *cursor* e *crlf*, bem como quaisquer funções

auxiliares do emulador de terminal, podem ser declaradas como *static* para restringir seu acesso apenas à implementação local.

Esta abordagem oferece dois benefícios fundamentais: primeiro, promove o encapsulamento ao ocultar os detalhes de implementação que não fazem parte da interface pública; segundo, previne possíveis conflitos de nomes durante a linkagem, já que identificadores com vinculação interna não são visíveis para outros arquivos objeto.

No contexto do desenvolvimento de *kernels*, esse controle de visibilidade é particularmente importante, pois permite organizar o código em módulos coesos com interfaces bem definidas, reduzindo o acoplamento entre componentes e facilitando a manutenção do sistema.

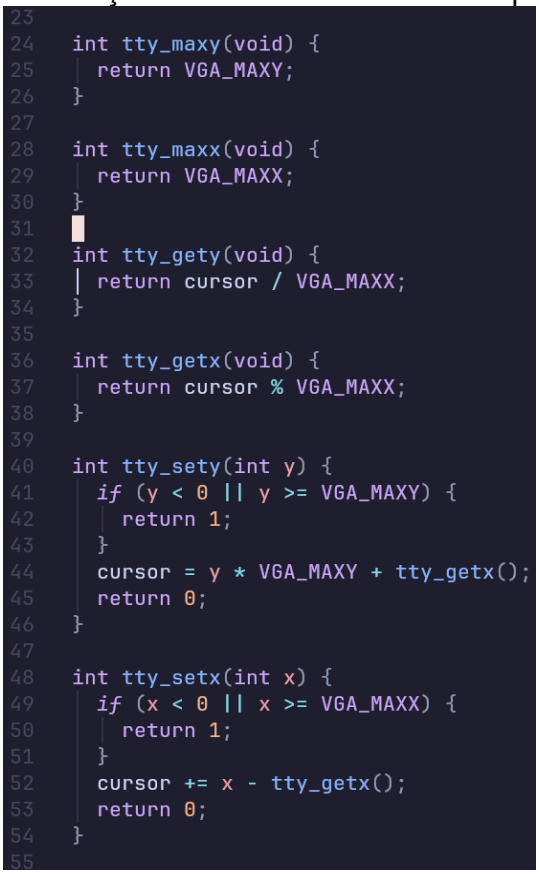
Nas linhas 16 a 22 está a implementação de *tty_clear*. Primeiramente, um *for loop* é feito para iterar sobre todas as posições do *buffer* de vídeo, cuja dimensão total é determinada pelo produto de *VGA_MAXY* e *VGA_MAXX*. Para cada posição, a função invoca *vga_write* com o caractere de espaço e os atributos de cor que definem o preto tanto para *foreground color* quanto para *background color*, removendo todo o conteúdo da tela. Após isso, na linha 20 o cursor é posto na posição inicial. A função retorna zero indicando que não houve erros.

Nas linhas 10 a 14, a implementação de *tty_initialize* começa invocando *tty_clear*, que apaga o conteúdo da tela e põe o cursor na posição inicial, desabilita o modo CRLF e encerra sua execução retornando zero para sinalizar a não ocorrência de erros.

5.2.1 Implementação das rotinas relacionadas a posição do cursor

A Figura 32 mostra a implementação das rotinas relacionadas à posição do cursor.

Figura 32 – Implementação das rotinas relacionadas a posição do cursor



```
23
24  int tty_maxy(void) {
25    return VGA_MAXY;
26  }
27
28  int tty_maxx(void) {
29    return VGA_MAXX;
30  }
31
32  int tty_gety(void) {
33    return cursor / VGA_MAXX;
34  }
35
36  int tty_getx(void) {
37    return cursor % VGA_MAXX;
38  }
39
40  int tty_sety(int y) {
41    if (y < 0 || y >= VGA_MAXY) {
42      return 1;
43    }
44    cursor = y * VGA_MAXY + tty_getx();
45    return 0;
46  }
47
48  int tty_setx(int x) {
49    if (x < 0 || x >= VGA_MAXX) {
50      return 1;
51    }
52    cursor += x - tty_getx();
53    return 0;
54  }
55
```

Fonte: Elaborado pelo autor (2025)

Nas linhas 24 a 30 são implementadas as funções *tty_maxy* e *tty_maxx*, que retornam os valores das macros *VGA_MAXY* e *VGA_MAXX* respectivamente. Estas macros, definidas na interface do *driver* de VGA, representam os limites máximos do terminal, indicando a última posição válida nos eixos y e x, respectivamente.

Nas linhas 32 a 34, a função *tty_gety* retorna a posição do cursor no eixo y com a divisão da variável global *cursor* pela macro *VGA_MAXX*. A operação aproveita o truncamento na divisão de inteiros, onde a parte fracionária do resultado é descartada, para produzir o índice da posição vertical do cursor.

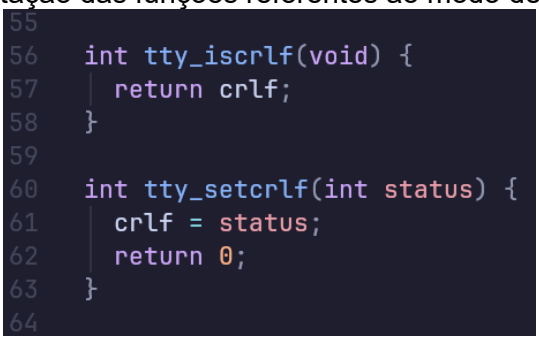
Complementarmente, nas linhas 36 a 38 a função *tty_getx* retorna a posição do cursor no eixo x através da operação módulo entre *cursor* e *VGA_MAXX*. Esta operação produz o resto da divisão entre os valores, que corresponde precisamente à posição horizontal do cursor.

Nas linhas 40 a 54 estão as implementações das funções *tty_sety* e *tty_setx*, que definem as coordenadas vertical e horizontal do cursor, respectivamente. Ambas as funções verificam se os valores recebidos como parâmetros estão dentro dos limites do terminal. Caso a posição seja inválida, as funções encerram sua execução retornando o valor 1 para indicar a ocorrência de um erro. Para atualizar a posição vertical, *tty_sety* recalcula o valor de *cursor* combinando a nova coordenada y com a posição horizontal corrente, enquanto *tty_setx* ajusta coordenada horizontal preservando a linha atual. Ambas as funções retornam zero ao fim de sua execução para indicar a não ocorrência de erros.

5.2.2 Implementação das rotinas relacionadas a escrita de caracteres

A Figura 33 mostra as implementações das funções *tty_iscrlf* e *tty_setcrlf*, onde a primeira retorna o valor da variável *crlf*, e a segunda usa o valor recebido como parâmetro para redefinir o valor de *crlf*, onde zero é falso e qualquer valor diferente de zero é verdadeiro.

Figura 33 – Implementação das funções referentes ao modo de operação do terminal



Fonte: Elaborado pelo autor (2025)

Para executar a operação de rolagem vertical, a função auxiliar *scroll* é definida conforme mostra a Figura 34.

Figura 34 – Função auxiliar de rolagem vertical

```
64
65  static void scroll(void) {
66    int i;
67
68    for (i = VGA_MAXX; i < VGA_MAXY * VGA_MAXX; i++) {
69      char character;
70      VGAColor foreground;
71      VGAColor background;
72
73      vga_read(i, &character, &foreground, &background);
74      vga_write(i - VGA_MAXX, character, foreground, background);
75    }
76
77    for (i -= VGA_MAXX; i < VGA_MAXX * VGA_MAXY; i++) {
78      vga_write(i, ' ', VGA_COLOR_BLACK, VGA_COLOR_BLACK);
79    }
80
81    cursor -= VGA_MAXX;
82  }
83
```

Fonte: Elaborado pelo autor (2025)

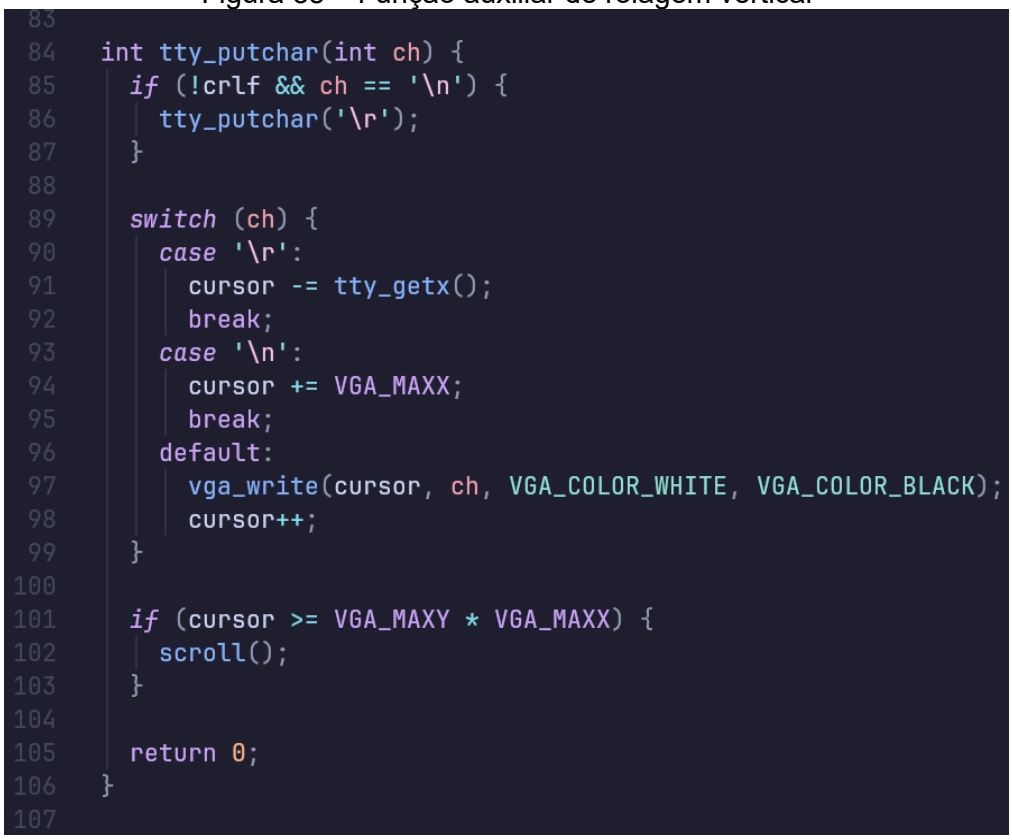
Primeiramente, um *for loop* inicia uma iteração a partir da segunda linha e vai até a última posição válida do terminal. A cada iteração, a função *vga_read* é usada para obter e armazenar as informações do caractere na posição correspondente ao contador *i* para que então essas variáveis sejam usadas como argumentos para a função *vga_write*, que será responsável por escrever o caractere e seus atributos na linha superior. O resultado é que ao fim do *loop*, o conteúdo de todas as linhas tenha sido copiado para as linhas imediatamente superiores.

Nas linhas 77 e 79 um outro *for loop* é feito, dessa vez iterando somente da posição horizontal inicial da última linha até a posição final. A cada iteração, a função *vga_write* é usada para escrever um caractere de espaço com o fundo preto, com o objetivo de apagar o conteúdo da última linha do terminal, que antes do *loop* estava igual ao da penúltima linha.

Por fim, é subtraído *VGA_MAXX* da posição do cursor para reposicioná-lo na linha superior à que ele estava.

A Figura 35 demonstra a implementação da função *tty_putchar*.

Figura 35 – Função auxiliar de rolagem vertical



```
83
84  int tty_putchar(int ch) {
85    if (!crlf && ch == '\n') {
86      tty_putchar('\r');
87    }
88
89    switch (ch) {
90      case '\r':
91        cursor -= tty_getx();
92        break;
93      case '\n':
94        cursor += VGA_MAXX;
95        break;
96      default:
97        vga_write(cursor, ch, VGA_COLOR_WHITE, VGA_COLOR_BLACK);
98        cursor++;
99    }
100
101   if (cursor >= VGA_MAXY * VGA_MAXX) {
102     scroll();
103   }
104
105   return 0;
106 }
107
```

Fonte: Elaborado pelo autor (2025)

Inicialmente, nas linhas 85 a 87, é verificado se o modo CRLF está desativado e o caractere a ser renderizado é um LF (‘\n’). Caso a condição seja verdadeira, *tty_putchar* chama a si mesma recursivamente para pôr um CR (‘\r’) antes de LF.

As linhas 89 a 98 usam a estrutura *switch* para processar diferencialmente caracteres de controle e caracteres comuns. O bloco *switch* define três comportamentos distintos baseados no caractere recebido como parâmetro:

- Para o caractere CR (‘\r’), a posição é atualizada com uma operação de subtração que remove o deslocamento horizontal corrente para voltar o cursor ao início da linha.
- Para o caractere de LF (‘\n’), o cursor avança para a linha seguinte através de uma uma operação que adiciona *VGA_MAXX* (tamanho de uma linha) à posição atual.

- No caso padrão (caracteres comuns), a função *vga_write* é invocada para renderizar o caractere na posição atual do cursor, com uma cor branca para ele e uma cor preta para o fundo, seguido do incremento da posição do cursor.

Após o processamento do caractere, nas linhas 101 a 103, a função invoca *scroll* para realizar a rolagem vertical caso a operação de escrita tenha feito o cursor ultrapassar os limites do terminal, e termina sua execução retornando zero para indicar a não ocorrência de erros.

5.2.3 Implementação das rotinas relacionadas a formatação de strings

A Figura 36 mostra a definição das funções auxiliares *puts* e *putint*.

Figura 36 – Definiçaão das funções auxiliares *puts* e *putint*

```
107
108  static void puts(const char *s) {
109    while (*s) {
110      tty_putchar(*s++);
111    }
112  }
113
114  static void putint(uint32_t num, bool is_negative, int base) {
115    static const char DIGITS[] = "0123456789abcdef";
116
117    char stack[sizeof(num) * 8 + 1];
118    int stack_top = 0;
119
120    do {
121      stack[stack_top++] = DIGITS[num % base];
122      num /= base;
123    } while (num > 0);
124
125    if (is_negative) {
126      stack[stack_top++] = '-';
127    }
128
129    while (stack_top > 0) {
130      tty_putchar(stack[--stack_top]);
131    }
132  }
133
```

Fonte: Elaborado pelo autor (2025)

A função *puts* tem a finalidade de imprimir uma *string* de caracteres. Sua implementação é bem simples, ela “varre” a *string* que recebeu como parâmetro usando a função *tty_putchar* para imprimir todos os seus caracteres.

Já a função *putint* tem o objetivo de imprimir um número inteiro. Sua implementação é mais elaborada e exige uma análise mais atenta. A função recebe três parâmetros:

- ***num***: Um inteiro do tipo *uint32_t* que será impresso.
- ***is_negative***: Um valor *booleano* que determina se a função deve imprimir o número acompanhado de um sinal de negatividade.
- ***base***: A base na qual o número será impresso, podendo ir de 2 até 16.

No topo da definição da função, está a declaração de uma *string* constante identificada por *DIGITS*, que contém todos os dígitos que podem ser demandados nas bases suportadas.

Apesar de apenas bases de 2 a 16 serem suportadas, nenhuma validação é feita para verificar se o valor de *base* atende a essa condição. Isso não é o ideal, porém como a função é usada como um auxiliar e não é acessível fora do arquivo, optou-se por confiar cegamente no valor passado.

Na linha 117, é declarado um *array* de caracteres denominado *stack*, que funciona como uma pilha para armazenar os dígitos resultantes da conversão numérica. Seu tamanho é calculado pela expressão “sizeof(num) * 8 + 1”, onde “sizeof(num) * 8” representa a quantidade de *bits* da variável *num*. Como a base binária é a que tem a representação numérica mais longa possível, isso garante que qualquer valor esteja dentro dos limites do *array*, enquanto o acréscimo de uma posição adicional serve para um possível sinal de negatividade. A linha 118 declara uma variável inteira para servir de ponteiro para o topo da pilha.

Nas linhas 120 a 123, um *loop do-while* utiliza uma técnica clássica de conversão numérica com divisões sucessivas. A cada iteração, o resto da divisão de *num* por *base* é utilizado como índice para acessar o caractere correspondente no *array DIGITS*, sendo armazenado na pilha com posterior incremento do ponteiro *stack_top*. Em seguida, o valor de *num* é atualizado pelo quociente inteiro da divisão. Este processo repete-se enquanto o valor de *num* permanecer maior que zero, garantindo que ao fim do *loop*, a pilha contenha todos os caracteres que representam o número na base especificada.

Por fim, nas linhas 129 a 131, após a inserção do caractere de negatividade no topo da pilha se necessário, a estrutura de repetição *while* é usada para recuperar os caracteres da pilha na ordem inversa à sua inserção. Com o decremento sucessivo

de *stack_top*, cada elemento é removido do topo da pilha e enviado para saída via *tty_putchar*, garantindo que a representação numérica final seja exibida na orientação correta.

As funções *puts* e *putint* serão, primariamente, usadas como auxiliares de *tty_printf*, que, como demonstra a Figura 37, possui uma implementação complexa.

Figura 37 – Implementação de *tty_printf*

```
137
138  int tty_printf(const char *format, ...) {
139      va_list args;
140      va_start(args, format);
141
142      while (*format) {
143          if (*format != '%') {
144              tty_putchar(*format++);
145              continue;
146          }
147
148          format++;
149
150          switch (*format) {
151              case 'c':
152                  tty_putchar(va_arg(args, int));
153                  break;
154              case 's':
155                  puts(va_arg(args, char*));
156                  break;
157              case 'd': {
158                  int num = va_arg(args, int);
159                  bool is_negative = num < 0;
160                  num = is_negative ? -num : num;
161                  putint((uint32_t) num, is_negative, 10);
162              } break;
163              case 'u': {
164                  int num = va_arg(args, unsigned int);
165                  putint((uint32_t) num, false, 10);
166              } break;
167              case 'x': {
168                  int num = va_arg(args, unsigned int);
169                  putint((uint32_t) num, false, 16);
170              } break;
171              case '%':
172                  tty_putchar('%');
173                  break;
174          }
175
176          format++;
177      }
178
179      return 0;
180  }
```

Fonte: Elaborado pelo autor (2025)

A função utiliza os recursos da biblioteca *stdarg.h* para implementar o mecanismo de argumentos variáveis. Na linha 139, é declarado uma variável do tipo *va_list*, que será responsável por armazenar o estado de iteração sobre os parâmetros adicionais.

Posteriormente, na linha 140, esta variável é inicializada com o uso da macro *va_start*, que requer dois parâmetros: a variável *va_list* previamente declarada e o último parâmetro nomeado da função. Esta inicialização estabelece o ponto de partida para a leitura dos argumentos variáveis, que agora podem ser obtidos com o uso da macro *va_arg*.

Nas linhas 142 a 177, há um longo bloco *while* que itera sobre todos os caracteres da *string* de formato e os processa apropriadamente. Primeiramente, as linhas 143 a 146 verificam se o caractere da iteração é diferente de “%”. Em caso afirmativo, a função *tty_putchar* é invocada para realizar a escrita, o ponteiro da *string* de formato é incrementado e a instrução *continue* é usada para avançar para a próxima iteração.

Quando o caractere for igual a “%”, a função incrementa o ponteiro da *string* de formato e entra em um bloco *switch*, que determinará a formatação apropriada através da análise do próximo caractere, que, junto a “%”, forma um código de formato. Na maioria dos casos, a macro *va_arg* será usada para obter o dado a ser formatado na lista de argumentos variáveis.

A análise e processamento do código de formato é feita da seguinte forma:

- **“%c”**: Manda o caractere obtido na lista de argumentos variáveis para ser processado pela função *tty_putchar*.
- **“%s”**: Manda a *string* obtida na lista de argumento variáveis para ser processada pela função auxiliar *puts*.
- **“%d”**: Obtém um inteiro com sinal na lista de argumentos variáveis, cria a variável *is_negative* e atribui o resultado de um teste *booleano* que indica se o valor é negativo ou não, obtém o valor absoluto do número e invoca a função auxiliar *putint* para processá-lo em base decimal, fazendo a devida conversão

para *uin32_t* e passando a variável *is_negative* como argumento para a função saber se deve imprimi-lo acompanhado de um sinal de negatividade ou não.

- **“%u”**: Obtém um inteiro sem sinal na lista de argumentos variáveis e o passa para a função *putint* processá-lo como um número em base decimal.
- **“%x”**: Obtém um inteiro sem sinal na lista de argumentos variáveis e o passa para a função *putint* processá-lo como um número em base hexadecimal.
- **“%%”**: Invoca *tty_putchar* para imprimir o caractere “%”.

Após o fim do bloco *switch*, o ponteiro da *string* de formato é incrementado e segue-se para a próxima iteração. Quando finalmente a *string* chegar ao fim e o bloco *while* encerrar sua execução, a função termina retornando o valor zero para indicar a não ocorrência de erros.

6 CONSIDERAÇÕES FINAIS

O presente trabalho teve como objetivo principal aplicar conceitos teóricos referentes a sistemas operacionais na construção de *kernels* para a arquitetura x86 usando linguagem C e *assembly*, através da análise minuciosa de um *kernel* chamado BergOS.

Todas as partes principais do BergOS foram apresentadas. O *bootloader*, que foi um importante laboratório para a análise de mecanismos importantes da arquitetura x86 como a GDT. O *kernel* em si, que usa rotinas definidas em interfaces abstratas para escrever uma mensagem de saudação na tela e parar sua execução.

Também foi abordado o *driver* de VGA, que serviu como o exemplo prático do conceito de E/S mapeada na memória, e seu uso na implementação da interface do emulador de terminal. Com a implementação do emulador de terminal, tanto o *kernel* quanto os futuros programas aplicativos do BergOS têm uma interface simples e agradável para escrever caracteres no monitor do usuário. Estando assim, livre das complexidades de um *driver* de vídeo e das especificidades de um *hardware.* Com isso, foi possível observar um exemplo real da abstração fornecida pelos sistemas operacionais.

Portanto, este trabalho faz sua contribuição ao se aprofundar na conexão inerente entre sistema operacional e *hardware*. Ilustrando essa conexão através de uma longa análise das características de uma arquitetura específica e demonstrando, através do BergOS, como elas são usadas para construir abstrações.

Apesar deste trabalho fornecer uma base sólida para a compreensão de como sistemas operacionais são programados e funcionam na prática, ainda há limitações que servem de gancho para trabalhos futuros:

- **Separação de espaço de *kernel* e espaço de usuário**: Um conceito extremamente importante em qualquer sistema operacional moderno. Trabalhos futuros devem explorar como paginação e os anéis de proteção são usados na arquitetura x86 para implementar essa separação.

- **Interrupções**: Mesmo que interrupções tenham sido apresentadas neste trabalho, não houve nenhum exame profundo que fizesse justiça à importância

desse tópico. Estudos posteriores devem se aprofundar nos mecanismos de interrupção da arquitetura x86, como a IDT e os controladores de interrupção programáveis: PIC e APIC.

- **Entrada de dados com teclado**: Permitir que o usuário entre dados a partir de um dispositivo de entrada como um teclado é o primeiro passo para um sistema operacional interativo. Seria proveitosa uma pesquisa que se aprofunde na implementação de *drivers* de teclado que lide diretamente com *scan codes*, *typematic* e interrupções.

- **Processos**: Provavelmente a abstração mais importante fornecida pelos sistemas operacionais. Trabalhos futuros devem explorar formas de se implementar processos, bem como protegê-los de adulteração por parte de outros processos. O uso de interrupções na programação de escalonadores é fundamental.

Conclui-se, portanto, que o estudo de sistemas operacionais não deve ser dissociado do estudo de arquitetura de computadores, e seu funcionamento só pode ser plenamente entendido quando se leva em conta o *hardware* para o qual ele está sendo programado, e, nesse sentido, BergOS se mostrou um laboratório frutífero para a compreensão dos conceitos teóricos, devido à sua natureza simples e didática.

REFERÊNCIAS

BERGANTON, Lucas. **BergOS**. 2025. Disponível em: https://github.com/lberganton/bergos/tree/tcc. Acesso em: 23 nov. 2025.

DODGE, Catherine; IRVINE, Cynthia; NGUYEN, Thuy. A Study of Initialization in Linux and OpenBSD. **ACM SIGOPS Operating Systems Review**, v. 39, n. 2, p. 79–93, abr. 2005.

FERRARO, Richard F. **Programmer's Guide to the EGA, VGA, and Super VGA Cards**. [S.l.]: Addison Wesley, 1994.

FREE SOFTWARE FOUNDATION. **GNU Make**. 2023. Disponível em: https://www.gnu.org/software/make/. Acesso em: 23 nov. 2025.

INTEL CORPORATION. **Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4**. [S. l.: s. n.], 2025. Disponível em: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Acesso em: 10 set. 2025.

INTERNATIONAL BUSINESS MACHINES CORPORATION (IBM). **Personal System/2 and Personal Computer BIOS Interface Technical Reference**. 1. ed. 1987. Disponível em: https://bitsavers.trailing-edge.com/pdf/ibm/pc/ps2/PS2_and_PC_BIOS_Interface_Technical_Reference_Apr87.pdf. Acesso em: 20 nov. 2025.

MAZIERO, Carlos Alberto. **Sistemas Operacionais: Conceitos e Mecanismos**. Curitiba: DINF - UFPR, 2019. *E-book*. Disponível em: https://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=socm:socm-livro.pdf. Acesso em: 20 nov. 2025.

PHOENIX TECHNOLOGIES. **BIOS Enhanced Disk Especification**. Version 1.1, 1995. Disponível em: https://wiki.sensi.org/download/doc/ata_edd_11.pdf. Acesso em: 20 nov. 2025.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer; GAGNE, Greg. **Fundamentos de sistemas operacionais**. Tradução de: Aldir José Coelho Correa da Silva. 9. ed. Rio de Janeiro: LTC, 2015. *E-book*.

STALLINGS, William. **Arquitetura e organização de computadores**. Tradução de: Sérgio Nascimento. 10. ed. São Paulo: Pearson Education do Brasil, 2017.

TANENBAUM, Andrew S.; BOS, Herbert. **Sistemas operacionais modernos**. Tradução de: Daniel Vieira; Jorge Ritter. 4. ed. São Paulo: Pearson Education do Brasi, 2016.

WILSON, Michelle. **What is VGA? A comprehensive guide to Video Graphics Array**. HP, 2024. Disponível em: https://www.hp.com/us-en/shop/tech-takes/what-is-vga-comprehensive-guide-video-graphics-array. Acesso em: 16 nov. 2025.

ZHIRKOV, Igor. **Programação em Baixo Nível: C, Assembly e execução de programas na arquitetura Intel 64**. Tradução de: Lúcia A. Kinoshita. 1. ed. São Paulo: Novatec Editora Ltda, 2018. *E-book*.

APÊNDICE A – PROCESSO DE COMPILAÇÃO DO BERGOS

O processo de compilação de um *kernel* difere do de em um projeto convencional. No geral, os desenvolvedores estão acostumados com uma compilação que envolve apenas buscar pelos arquivos de código-fonte escritos em uma linguagem de programação, enviá-los ao compilador e esperar que o binário gerado seja imediatamente executável, sem passos extras. Porém, um *kernel*, bem como qualquer programa destinado a execução direta por um processador, é um projeto sensível com relação à fase de compilação, pois, diferentemente de um projeto de *software* comum, o binário final deve ser construído cuidadosamente.

Os binários pré-compilados de um compilador disponíveis em uma plataforma são feitos para gerar binários compatíveis com ela. O GCC, disponível para *download* no repositório público de uma distribuição Linux, por exemplo, foi compilado para que o binário gerado esteja no formato ELF, enquanto o MinGW (porte do GCC para sistemas Windows), para que o binário gerado esteja no formato PE. Esses formatos são feitos para serem processados por um sistema operacional. Eles não apenas possuem código de máquina, mas outros dados e informações que serão utilizados pelo sistema para carregá-lo na memória e pô-lo em execução. Isso se torna um problema para o processo de compilação de um *kernel*, pois o usuário provavelmente tem um compilador que espera gerar código que dependa de um sistema operacional e de uma arquitetura específica, enquanto um *kernel* não pode depender de um sistema operacional e pode ter como alvo uma arquitetura diferente daquela executada pelo usuário.

O BergOS tem que contornar essa dificuldade, pois ele é feito para ser executado sobre um processador de 32 *bits*, enquanto a maioria dos sistemas, hoje em dia, são feitos para executar em processadores de 64 *bits*. Portanto, o compilador que um usuário de Linux terá disponível, por exemplo, irá, a princípio, gerar código no formato ELF para 64 *bits*, diferente do binário “cru” de 32 *bits* esperado pelo BergOS.

Para contornar esse problema há duas soluções principais: compilar o próprio compilador, para que ele gere um binário compatível com a arquitetura desejada sem depender de um sistema operacional; ou usar o compilador disponível na plataforma,

mas usar muitas *flags* de compilação para forçar a geração de um binário no formato desejado.

A primeira solução é mais elegante, recomendada para projetos grandes e tende a dar menos problemas, pois se especifica exatamente o que deve ser gerado, permitindo, inclusive, que usuários que não estejam executando um sistema x86 gerem códigos para ele. Porém, a segunda solução é tentadora devido à sua simplicidade, afinal, nenhuma etapa extra é necessária além de instalar o compilador já disponível na plataforma do usuário. O BergOS segue com a segunda opção, justamente para facilitar que o usuário teste o sistema, sem exigir o trabalho extra de compilar um compilador apenas para este fim.

Outro problema comum no processo de compilação de *kernels* é que eles geralmente são escritos em mais de uma linguagem de programação. O *kernel* Linux, por exemplo, é programado em *assembly*, C e Rust. Isso não apenas aumenta as dependências do projeto, como aumenta a complexidade de sua compilação, pois agora terá de se pensar em um jeito delas se comunicarem de alguma forma.

Para atingir esse fim, um conceito importante precisa ser analisado: os símbolos, uma das informações mais úteis armazenadas em um arquivo objeto. Um símbolo nada mais é do que um endereço nomeado. Esses símbolos são armazenados no arquivo objeto em um local chamado tabela de símbolos. Um símbolo pode ser exportado para ser usado por outros arquivos objetos. Também pode ser marcado para ser resolvido no processo de linkagem, permitindo assim que o arquivo-fonte interaja com símbolos declarados em outros arquivos-fonte. No fim, o trabalho do *linker* é fazer justamente o que seu nome diz, ligar todos os arquivos objeto em um único arquivo final, fazendo cada símbolo presente nos arquivos de entrada se referir a um único endereço.

A forma como os símbolos são tratados depende do compilador e da linguagem de programação. Em *assembly*, as coisas são mais intuitivas, já que todo rótulo, a princípio, se torna um símbolo. Em linguagem C, uma função pode facilmente ser convertida em um símbolo de mesmo nome. Por padrão, toda função é um símbolo que será exportado, ou seja, será visível para outros arquivos objeto, onde o *linker*, ao encontrar referências a esse símbolo em outros arquivos objeto, resolverá para que no

arquivo gerado eles se refiram ao endereço da função correspondente. Quando a função é declarada como *static*, isso diz ao compilador que o símbolo (identificador daquela função) não deve ser exportado, ou seja, outros arquivos objeto não devem ser capazes de acessá-lo.

Para ser capaz de usar as capacidades do *linker* e poder compartilhar símbolos entre o código-fonte, cada arquivo é compilado, unitariamente, para o formato ELF32 (a versão de 32 *bits* do formato ELF), para, na fase de linkagem, esses arquivos objeto isolados serem unidos para formarem um único binário “cru”.

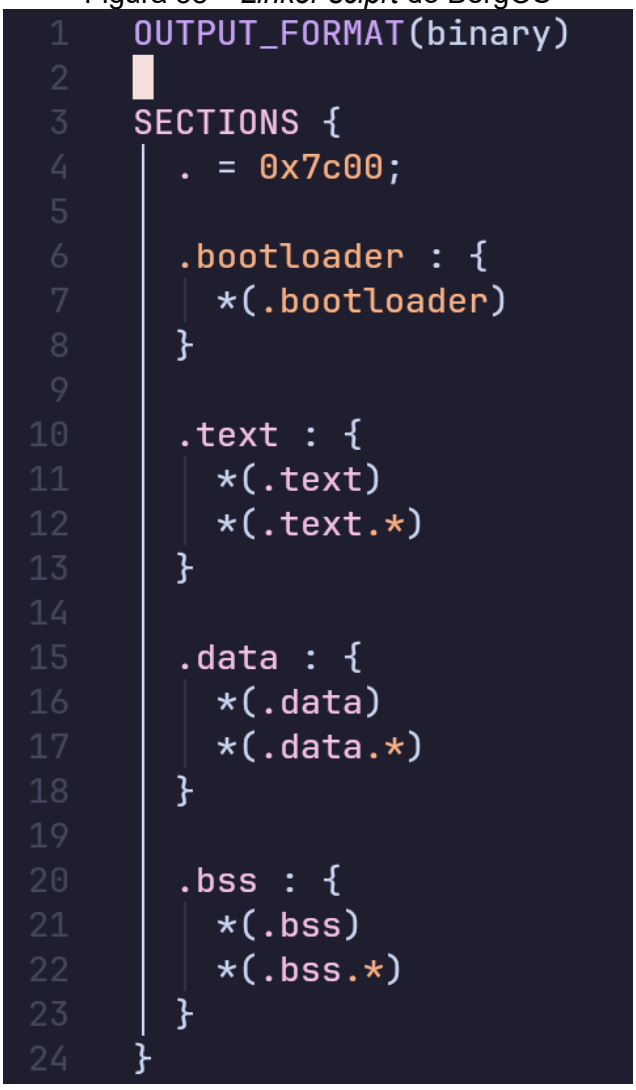
A.1 Linker *script*

Como já estabelecido, no desenvolvimento de *kernels* o formato do binário final é extremamente importante. Isso inclui a forma em que o código e os dados são dispostos nele. O maior exemplo disso é o caso do *bootloader*, que, como abordado no capítulo 3, deve estar no primeiro setor de um dispositivo para que ele possa ser reconhecido. Isso traz a necessidade de posicioná-lo bem no início do binário.

Para especificar a forma do binário final, juntamente com a posição exata dos códigos e dos dados, é possível fornecer um *linker script* para o *linker* do GCC. O *linker script* é uma ferramenta poderosa, mas relativamente pouco usada já que não há tanta necessidade de especificar o formato do binário executável em alto nível. Porém, em baixo nível ela se torna indispensável.

O *linker script* de BergOS está no arquivo *./linker.ld*. A Figura 38 mostra o conteúdo desse arquivo.

Figura 38 – *Linker sciprt* do BergOS



```
1   OUTPUT_FORMAT(binary)
2
3   SECTIONS {
4     . = 0x7c00;
5
6     .bootloader : {
7       *(.bootloader)
8     }
9
10    .text : {
11      *(.text)
12      *(.text.*)
13    }
14
15    .data : {
16      *(.data)
17      *(.data.*)
18    }
19
20    .bss : {
21      *(.bss)
22      *(.bss.*)
23    }
24  }
```

Fonte: Elaborado pelo autor (2025)

A primeira linha define que o arquivo a ser gerado é um binário “cru”, ou seja, deve conter apenas código executável. Isso impede o *linker* de produzir formatos que não são imediatamente executáveis pelo processador, como ELF. Nas linhas 3 a 24, há um longo bloco chamado *SECTIONS;* é nesse bloco onde a disposição do código e dos dados pode ser manualmente definida.

Primeiramente, na linha 4, é especificado que o código deve tratar seu primeiro endereço como sendo 0x7C00. Isso é necessário, pois é nesse endereço de memória que o *bootloader* será carregado. Isso faz com que o *linker* resolva as referências a endereços para corresponder a essa base. Por exemplo, se o código

objeto fizer um salto para o endereço 0x0010, com essa declaração, o *linker* fará com que, no binário final, o salto ocorra para 0x7C10.

Depois disso, as declarações seguintes especificam a posição exata na qual as seções dos arquivos objeto devem estar no binário final. Os arquivos ELF possuem algumas seções padrão, dentre elas está *.text*, *.data* e *.bss*. A primeira é usada para código executável; a segunda para dados inicializados e a última para dados não inicializados. A ordem destas no binário final do BergOS não é tão importante. Porém, há a necessidade de o código do *bootloader* estar imediatamente no início do arquivo.

Como foi analisado na seção 3.2 do capítulo 3, o código do *bootloader* foi posto em uma seção personalizada chamada *.bootloader*. Com esse truque, se torna fácil colocar o código do *bootloader* no início do binário, bastando apenas pôr a seção *.bootloader* antes das outras.

A.2 GNU Make

O BergOS utiliza o GNU Make para automatizar o processo de compilação. O GNU Make é uma ferramenta popular no mundo Linux, principalmente em projetos envolvendo C e *assembly*. Segundo a Free Software Foundation “O GNU Make é uma ferramenta que controla a geração de executáveis e outros arquivos não-fonte de um programa a partir dos arquivos-fonte do programa.” (FREE SOFTWARE FOUNDATION, 2023, tradução nossa).

A popularidade da ferramente se deve muito ao fato dela ter a simplicidade de um *shell*, mas possuir funcionalidades que auxiliam o processo de *build*. Um exemplo disso é a capacidade do Make de reconhecer quais arquivos preciso ser recompilados.

> O Make determina automaticamente quais arquivos precisam ser atualizados, com base nos arquivos de origem que foram alterados. Ele também determina automaticamente a ordem correta para atualizar os arquivos, caso um arquivo não-fonte dependa de outro arquivo não-fonte. Como resultado, se você alterar alguns arquivos de origem e executar o Make, ele não precisará recompilar todo o seu programa. Ele atualizará apenas os arquivos não-fonte que dependem direta ou indiretamente dos arquivos de origem que você alterou (FREE SOFTWARE FOUNDATION, 2023, tradução nossa).

Os *scripts* de *build* são feitos a partir de um arquivo chamado *Makefile*, que “[…] lista cada um dos arquivos não-fonte e como computá-los a partir de outros arquivos. Ao escrever um programa, você deve escrever um Makefile para ele, para que seja

possível usar o Make para compilar e instalar o programa.” (FREE SOFTWARE FOUNDATION, 2023, tradução nossa).

A parte mais importante de um Makefile são as *rules*. São elas que determinam como gerar os arquivos desejados.

> Uma *rule* no arquivo Makefile informa ao Make como executar uma série de comandos para gerar um arquivo de destino a partir de arquivos de origem. Ela também especifica uma lista de dependências do arquivo de destino. Essa lista deve incluir todos os arquivos (sejam arquivos de origem ou outros arquivos de destino) que são usados como entradas para os comandos na *rule* (FREE SOFTWARE FOUNDATION, 2023, tradução nossa).

O Makefile do BergOS está localizado em *./Makefile*. Ele contém *rules* que vão desde compilar o kernel a executá-lo no emulador QEMU.

A Figura 39 mostra o início do Makefile.

Figura 39 – Início do Makefile do BergOS

```
1  # Architecture
2  ARCH ?= i386
3
4  # Directories
5  ARCH_DIR := arch/$(ARCH)
6  KERNEL_DIR := kernel
7  BUILD_DIR := build
8
9  # Output
10 OUTPUT := $(BUILD_DIR)/bergos.img
11 OUTPUT_SIZE := 64k
12
```

Fonte: Elaborado pelo autor (2025)

Nesse trecho, algumas variáveis importantes são declaradas. Na linha 2, a variável *ARCH* é definida apenas se ela já não tiver valor. Essa variável se refere a arquitetura alvo para a qual o BergOS será compilado. Como BergOS pode vir a suportar outras arquiteturas, é importante fornecer um meio para o usuário escolher para qual arquitetura ele quer compilar. Caso o usuário queira compilar o BergOS para x86_64, por exemplo, ele pode definir a variável *ARCH* no momento de invocar o

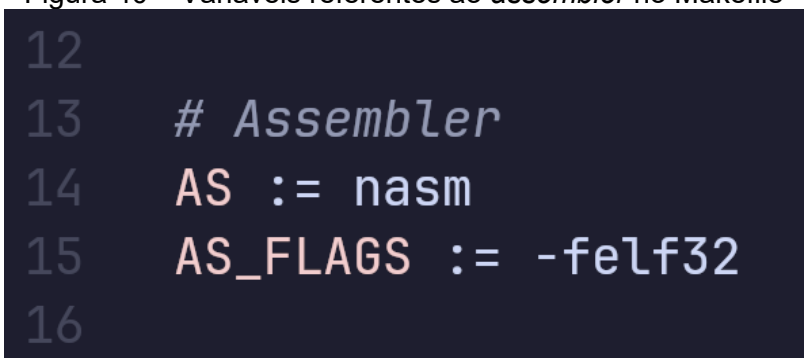
Make, com “make ARCH=x86_64”. O operador de atribuição condicional (“?=”) é útil nesse contexto, pois seleciona a arquitetura I386 como padrão caso o usuário não defina explicitamente outra.

Nas linhas 5 a 7, são declaradas variáveis referentes aos diretórios do projeto. *ARCH_DIR* se refere ao diretório que contém os códigos da camada de baixo nível (dependente de arquitetura), *KERNEL_DIR* se refere ao diretório com os códigos da camada de alto nível (independente de arquitetura) *e BUILD_DIR* se trata do diretório onde os arquivos objeto e o *kernel* compilado serão colocados.

Na linha 10, a variável *OUTPUT* é usada para identificar o caminho onde o binário do BergOS compilado será posto. Na linha 11, a variável *OUTPUT_SIZE* se refere ao tamanho do binário do BergOS. O processo de compilação forçará esse tamanho, mesmo que a compilação resulte em um arquivo muito menor que esse.

A Figura 40 mostra as variáveis referentes ao *assembler*.

Figura 40 – Variáveis referentes ao *assembler* no Makefile



Fonte: Elaborado pelo autor (2025)

A váriavel *AS* demarca o NASM como *assembler* e a variável *AS_FLAGS* será usada para conter as *flags* de montagem que serão passadas para o NASM. Apenas a flag “-felf32” é usada, que significa que o NASM deve gerar um arquivo objeto no formato ELF32 (a versão 32 *bits* do formato ELF).

A Figura 41 mostra as variáveis referentes compilador C.

Figura 41 – Variáveis referentes ao compilador C

```
16
17  # C Compiler
18  CC := gcc
19  CC_FLAGS := -std=gnu99 -m32 -Wall -Wextra -nostdlib -ffreestanding -fno-pic -fno-stack-protector -
    mno-sse -mno-sse2 -mno-mmx
20  CC_INCLUDES := $(addprefix -I,$(dir $(shell find $(ARCH_DIR) $(KERNEL_DIR) -type f -name '*.h')))
21
```

Fonte: Elaborado pelo autor (2025)

A variável *CC* é inicializada com *gcc*, o compilador que será usado. A variável *CC_INCLUDES* será usada como *flag* de compilação para que um código fonte C possa incluir arquivos de cabeçalho presentes no projeto. A atribuição é feita através de um *script* que busca por todos os diretórios que contém arquivos que terminam com ".h".

Já a parte mais importante está em *CC_FLAGS*. Como o BergOS pode ser compilado por um compilador comum, há a necessidade de se usar muitas *flags* de compilação para forçar a geração do binário no formato desejado.

As *flags* de compilação usadas são:

- **"-std=gnu99"**: Faz o compilador usar o padrão *gnu99*, baseado no padrão *c99* mas com expansões de gramática estabelecidas pelo projeto GNU. Essas expansões são úteis em desenvolvimento de *kernels*, principalmente por fornecerem mecanismos de *inline assembly*, uma forma de escrever código *assembly* diretamente em linguagem C.
- **"-m32"**: Força o GCC a gerar código de 32 *bits*.
- **"-Wall"**: Não é estritamente necessário, mas gera avisos úteis em tempo de compilação.
- **"-Wextra"**: Também não é necessário, mas fornece outros alertas em tempo de compilação.
- **"-nostdlib"**: Extremamente importante. O GCC, por padrão, faz a *linkagem* do código com a biblioteca padrão C. Isso é útil em alto nível, porém em baixo nível

a maioria dos recursos da biblioteca padrão C não estão disponíveis. Para evitar esse problema, esta *flag* diz para o GCC não fazer essa linkagem.

- **“-ffreestanding”**: Diz para o GCC que o código rodará em ambiente *freestanding*. Este é um termo do padrão da linguagem C para se referir a ambientes onde a biblioteca padrão da linguagem C não está totalmente disponível.
- **“-fno-pic”**: Faz o GCC gerar código que use endereços absolutos.
- **“-fno-stack-protector”**: Desativa o mecanismo de proteção de pilha, já que este depende de recursos do sistema operacional.
- **“-mno-sse”**: Diz para o GCC não gerar código que use instruções SSE. Essa é uma extensão da arquitetura x86 que não está disponível no 80386.
- **“-mno-sse2”**: Diz para o GCC não gerar código que use instruções SSE2. Essa é uma extensão da arquitetura x86 que não está disponível no 80386.
- **“-mno-mmx”**: Diz para o GCC não gerar código que use instruções MMX. Essa é uma extensão da arquitetura x86 que não está disponível no 80386.

A Figura 42 mostra as variáveis referentes ao *linker*.

Figura 42 – Variáveis referentes ao *linker*

```
21
22  # Linker
23  LINKER_SCRIPT := linker.ld
24  LINKER_FLAGS := -T $(LINKER_SCRIPT)
25
```

Fonte: Elaborado pelo autor (2025)

A variável *LINKER_SCRIPT* referencia o arquivo de *linker script*, e a variável *LINKER_FLAGS* define as *flags* de linkagem a serem utilizadas pelo linker.

Especificamente, essa *flag* tem a função de instruir o *linker* a utilizar o *script* de linkagem especificado na variável *LINKER_SCRIPT*.

A Figura 43 mostra as variáveis relacionadas aos arquivos de código-fonte e aos arquivos de código-objeto.

Figura 43 – Variáveis referentes a código-fonte e código-objeto

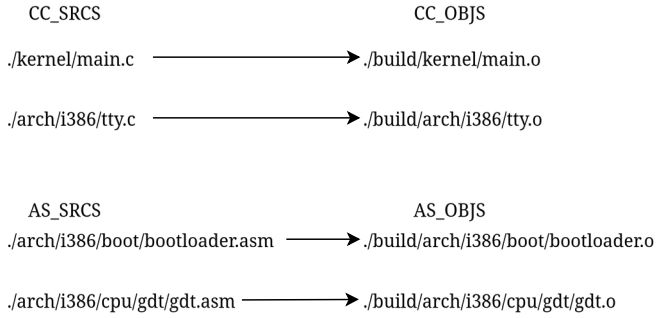
```
29
30  # Sources
31  CC_SRCS := $(shell find $(ARCH_DIR) $(KERNEL_DIR) -type f -name '*.c')
32  AS_SRCS := $(shell find $(ARCH_DIR) -type f -name '*.asm')
33
34  CC_OBJS := $(patsubst %.c,%.c.o,$(addprefix $(BUILD_DIR)/,$(CC_SRCS)))
35  AS_OBJS := $(patsubst %.asm,%.asm.o,$(addprefix $(BUILD_DIR)/,$(AS_SRCS)))
36
```

Fonte: Elaborado pelo autor (2025)

A variável *CC_SRCS* utiliza o comando *find* do *shell* para localizar recursivamente todos os arquivos com extensão ".c" nos diretórios *ARCH_DIR* (dependente de arquitetura) e *KERNEL_DIR* (independente de arquitetura). Similarmente, *AS_SRCS* coleta arquivos *assembly* que terminam com ".asm" exclusivamente do diretório *ARCH_DIR*.

As variáveis *CC_OBJS* e *AS_OBJS* mapeiam os arquivos fonte para seus respectivos arquivos objetos no diretório de *build*. Arquivos ".c" são transformados em ".c.o" e arquivos ".asm" são transformados em ".asm.o". A Figura 44 ilustra esse mapeamento.

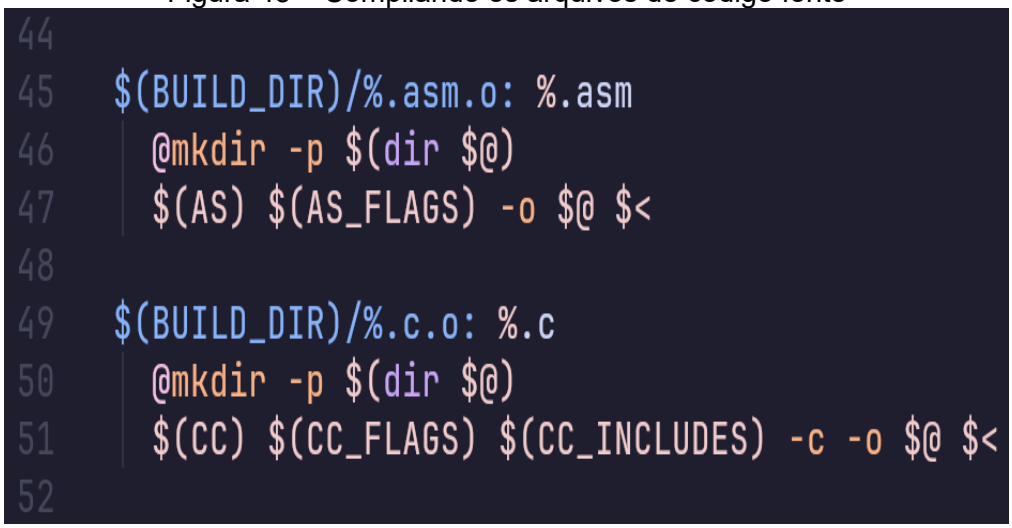
Figura 44 – Mapeamento de arquivos fonte em arquivos objeto



Fonte: Elaborado pelo autor (2025)

A Figura 45 mostra as *rules* responsáveis por compilar os arquivos de código.

Figura 45 – Compilando os arquivos de código fonte



```
44
45  $(BUILD_DIR)/%.asm.o: %.asm
46      @mkdir -p $(dir $@)
47      $(AS) $(AS_FLAGS) -o $@ $<
48
49  $(BUILD_DIR)/%.c.o: %.c
50      @mkdir -p $(dir $@)
51      $(CC) $(CC_FLAGS) $(CC_INCLUDES) -c -o $@ $<
52
```

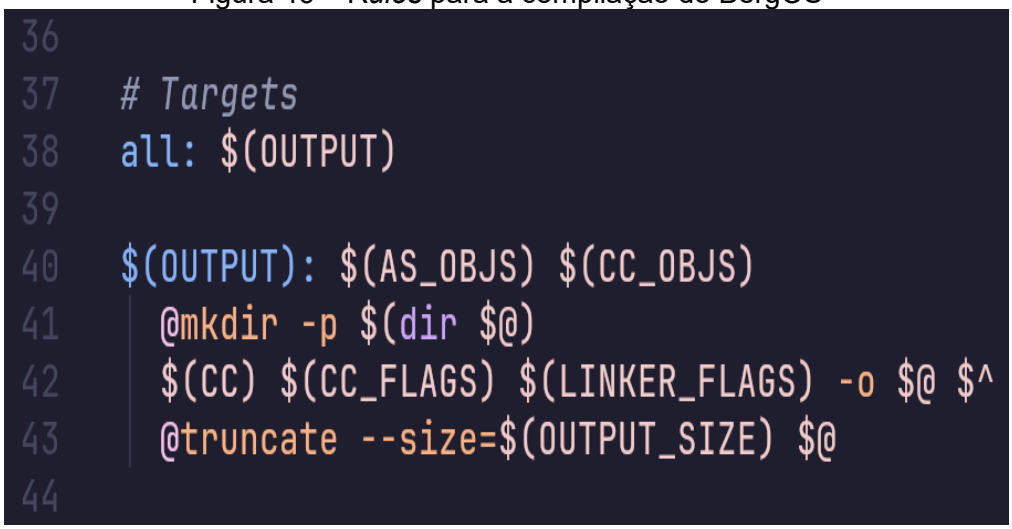
Fonte: Elaborado pelo autor (2025)

A *rule* definida na linha 45 especifica como construir objetos *assembly*. O curinga “%” captura o nome base do arquivo. A linha 46 cria silenciosamente o diretório de destino necessário para o arquivo objeto, onde “$@” expande para o nome do alvo. Em seguida, a linha 47 invoca o *assembler* com as *flags* apropriadas, onde “$<” representa o primeiro pré-requisito (arquivo terminado com “.asm”), gerando o objeto especificado.

Analogamente, a *rule* definida na linha 49 gerencia a compilação de arquivos C. Após criar o diretório, na linha 51, o código é compilado com as *flags* e diretórios de inclusão apropriados. A *flag* “-c” é importante, pois ela indica para o compilador gerar apenas o arquivo objeto, sem passar pelo *linker*.

Por fim, ao compilar todo o código fonte, a imagem do BergOS será gerada. A Figura 46 mostra as *rules* necessárias para isso.

Figura 46 – R*ules* para a compilação do BergOS



Fonte: Elaborado pelo autor (2025)

A *rule all* constitui o *target* padrão, sendo executada automaticamente quando o Make é invocado sem argumentos. Ela tem como dependência *OUTPUT*, o que faz sua execução produzir a imagem do BergOS. A *rule* definida na linha 40 é a responsável por fazer a geração do binário final. Após a criação do diretório, o GCC é invocado, mas desta vez para agir como *linker*. Tanto as *flags* usadas para compilar código C quanto as *flags* do *linker* são fornecidas. Desta vez, os arquivos de entrada são especificados através de "$^", que expande para todas as dependências (arquivos objeto).

Na linha 43, o comando de *shell truncate* é usado para ajustar o tamanho da imagem gerada para a especificada na variável *OUTPUT_SIZE*.

A.3 Compilando e executando o BergOS

O projeto BergOS possui três dependências: GCC, NASM e GNU Make. O GCC é usado para compilar códigos C e para fazer a linkagem; o NASM é usado para montar códigos *assembly* e o GNU Make é usado para os *scripts* de *build*. Será necessário um sistema Unix-like para realizar a compilação. Em sistemas Windows, é possível conseguir um ambiente Unix com WSL ou Cygwin.

Para compilar o BergOS, basta invocar o Make em um *shell*. Se nenhum erro ocorrer, a imagem do BergOS compilada estará em *./build/bergos.img*. A Figura 47 mostra a execução do Makefile para compilar o BergOS.

Figura 47 – Compilando o BergOS

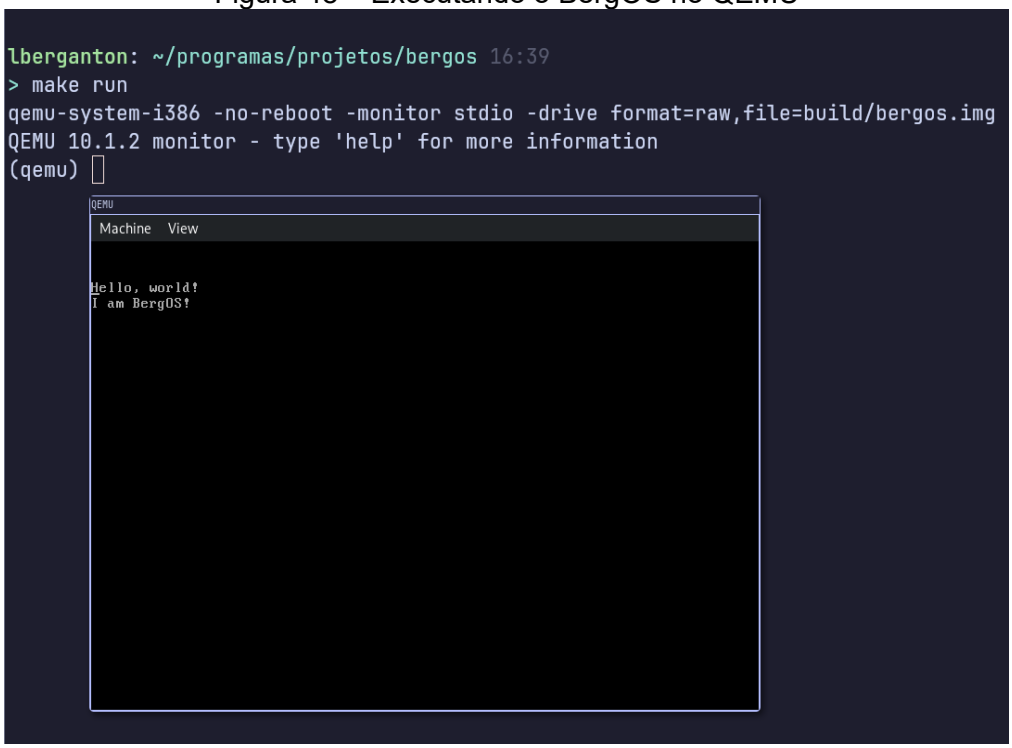
```
lberganton: ~/programas/projetos/bergos 16:39
> make
nasm -felf32 -o build/arch/i386/cpu/idt/idt.asm.o arch/i386/cpu/idt/idt.asm
nasm -felf32 -o build/arch/i386/cpu/gdt/gdt.asm.o arch/i386/cpu/gdt/gdt.asm
nasm -felf32 -o build/arch/i386/boot/bootloader.asm.o arch/i386/boot/bootloader.asm
gcc -std=gnu99 -m32 -Wall -Wextra -nostdlib -ffreestanding -fno-pic -fno-stack-protector -mno-sse -mno-sse2 -mno-mmx -Iarch/i386/cpu/ -Iarch/i386/
cpu/idt/ -Iarch/i386/cpu/gdt/ -Iarch/i386/video/vga/ -Ikernel/include/ -Ikernel/include/ -c -o build/arch/i386/cpu/idt/idt.c.o arch/i386/cpu/idt/i
dt.c
gcc -std=gnu99 -m32 -Wall -Wextra -nostdlib -ffreestanding -fno-pic -fno-stack-protector -mno-sse -mno-sse2 -mno-mmx -Iarch/i386/cpu/ -Iarch/i386/
cpu/idt/ -Iarch/i386/cpu/gdt/ -Iarch/i386/video/vga/ -Ikernel/include/ -Ikernel/include/ -c -o build/arch/i386/tty.c.o arch/i386/tty.c
gcc -std=gnu99 -m32 -Wall -Wextra -nostdlib -ffreestanding -fno-pic -fno-stack-protector -mno-sse -mno-sse2 -mno-mmx -Iarch/i386/cpu/ -Iarch/i386/
cpu/idt/ -Iarch/i386/cpu/gdt/ -Iarch/i386/video/vga/ -Ikernel/include/ -Ikernel/include/ -c -o build/arch/i386/video/vga/vga.c.o arch/i386/video/v
ga/vga.c
gcc -std=gnu99 -m32 -Wall -Wextra -nostdlib -ffreestanding -fno-pic -fno-stack-protector -mno-sse -mno-sse2 -mno-mmx -Iarch/i386/cpu/ -Iarch/i386/
cpu/idt/ -Iarch/i386/cpu/gdt/ -Iarch/i386/video/vga/ -Ikernel/include/ -Ikernel/include/ -c -o build/arch/i386/kernel.c.o arch/i386/kernel.c
gcc -std=gnu99 -m32 -Wall -Wextra -nostdlib -ffreestanding -fno-pic -fno-stack-protector -mno-sse -mno-sse2 -mno-mmx -Iarch/i386/cpu/ -Iarch/i386/
cpu/idt/ -Iarch/i386/cpu/gdt/ -Iarch/i386/video/vga/ -Ikernel/include/ -Ikernel/include/ -c -o build/kernel/main.c.o kernel/main.c
gcc -std=gnu99 -m32 -Wall -Wextra -nostdlib -ffreestanding -fno-pic -fno-stack-protector -mno-sse -mno-sse2 -mno-mmx -T linker.ld -o build/bergos.
img build/arch/i386/cpu/idt/idt.asm.o build/arch/i386/cpu/gdt/gdt.asm.o build/arch/i386/boot/bootloader.asm.o build/arch/i386/cpu/idt/idt.c.o buil
d/arch/i386/tty.c.o build/arch/i386/video/vga/vga.c.o build/arch/i386/kernel.c.o build/kernel/main.c.o

lberganton: ~/programas/projetos/bergos 16:39
>
```

Fonte: Elaborado pelo autor (2025)

Para executar o BergOS, o usuário pode optar por um emulador de IA-32. O QEMU é uma boa opção, pois é simples, multiplataforma e foi o principal ambiente usado no desenvolvimento do BergOS. O Makefile do projeto fornece um *target* chamado *run* para executar a imagem do BergOS compilada no QEMU.

A Figura 48 mostra a execução do BergOS no QEMU através do *target run* do Makefile.

Figura 48 – Executando o BergOS no QEMU



Fonte: Elaborado pelo autor (2025)

Por outro lado, é possível executar o BergOS em uma máquina real, desde que o processador seja compatível com a arquitetura IA-32. A Figura 49 mostra o BergOS executando sobre uma máquina real, com um processador Intel Core i7-7700 e uma placa-mãe MS-7A15.

Figura 49 – Executando o BergOS em uma máquina real

Fonte: Elaborado pelo autor (2025)