UNIVERSIDADE FEDERAL DE RORAIMA – UFRR

DEPARTAMENTO DE CIÊNCIAS DA COMPUTAÇÃO – DCC

SISTEMAS OPERACIONAIS I – DCC403

Desenvolvimento e Simulação de um Gerenciador de Boot Inspirado no GRUB com QEMU

Agosto de 2024

Boa Vista, Roraima

UNIVERSIDADE FEDERAL DE RORAIMA – UFRR

DEPARTAMENTO DE CIÊNCIAS DA COMPUTAÇÃO – DCC

SISTEMAS OPERACIONAIS I – DCC403

Alunos: Ranier Sales Veras e Anderson Silva

Agosto de 2024

Boa Vista, Roraima

RESUMO

Este projeto tem como objetivo desenvolver e simular um gerenciador de boot personalizado, inspirado no GRUB, utilizando linguagem Assembly e a ferramenta de emulação QEMU. A proposta consiste em criar uma estrutura de boot simples, capaz de exibir um menu textual, receber entrada do usuário e carregar diferentes kernels escritos em Assembly. O sistema é empacotado em uma imagem de disco simulada e executado com QEMU, permitindo testes completos do processo de boot, desde a BIOS até a execução do kernel.

A arquitetura do projeto está dividida em três componentes principais:

1. **Bootloader:** O primeiro estágio (MBR) é responsável por carregar um segundo estágio mais robusto, que pode exibir mensagens ou menus e preparar o carregamento do kernel.
2. **Kernel simples**: Um pequeno sistema em Assembly, com funções básicas para teste e exibição de mensagens na tela.
3. **Ambiente de execução com QEMU**: O sistema é empacotado em uma imagem de disco simulada e executado com a ferramenta QEMU, permitindo testes de boot como se fosse um sistema real.

O projeto envolve também o uso de ferramentas como NASM, LD, dd e make, demonstrando o funcionamento de baixo nível de um sistema de inicialização, desde o carregamento inicial da BIOS até a execução de um código C no modo protegido.

A simulação com QEMU permite validar todo o processo, proporcionando uma visão prática do funcionamento interno de um sistema de boot e do papel dos bootloaders como o GRUB.

IMPLEMENTAÇÃO

Antes de rodar o sistema no QEMU, o projeto é **compilado e montado** com a ajuda do Makefile.

Bootloader: O bootloader desenvolvido em Assembly é carregado diretamente pela BIOS, operando em modo real (16 bits) e ocupando os primeiros 512 bytes do disco (MBR). Após configurar o ambiente mínimo (registradores de segmento, pilha e modo de vídeo), ele exibe um menu de seleção na tela e aguarda a entrada do usuário.

```
; ================================================================
; mbr.asm - Bootloader Master com gerenciamento de kernel
; ================================================================

org 0x7C00
bits 16

start:
    ; Salva o drive de boot passado pela BIOS em DL
    mov [boot_drive], dl

    ; Inicializa os registradores de segmento
    cli
    xor ax, ax
    mov es, ax
    mov ds, ax
    mov ss, ax
    mov sp, 0x7C00
    sti

    ; Limpa a tela
    mov ah, 0x00
    mov al, 0x03
    int 0x10

    ; Exibe o menu
    mov si, msg_title
    call print
    mov si, opt1
    call print
    mov si, opt2
    call print
    mov si, opt3
    call print

menu_loop:
    mov ah, 0x00
    int 0x16

    cmp al, '1'
    je load_kernel1
    cmp al, '2'
    je load_kernel2
    cmp al, '3'
    je load_kernel3

    jmp menu_loop
```

Com base na tecla pressionada (1, 2 ou 3), o bootloader carrega diferentes setores do disco, correspondentes a três kernels distintos, e transfere o controle para o kernel selecionado, iniciando sua execução na memória.

```
; ====================================================================
; Rotinas de Carregamento CHS para setores próximos
; ====================================================================

load_kernel1:
    mov ah, 0x02
    mov al, 10
    mov ch, 0
    mov cl, 3
    mov dh, 0
    mov dl, [boot_drive]
    mov bx, 0x1000
    mov es, bx
    xor bx, bx
    int 0x13
    jc disk_error
    jmp 0x1000:0000

load_kernel2:
    mov ah, 0x02
    mov al, 10
    mov ch, 0
    mov cl, 4
    mov dh, 0
    mov dl, [boot_drive]
    mov bx, 0x1000
    mov es, bx
    xor bx, bx
    int 0x13
    jc disk_error
    jmp 0x1000:0000

load_kernel3:
    mov ah, 0x02
    mov al, 10
    mov ch, 0
    mov cl, 5
    mov dh, 0
    mov dl, [boot_drive]
    mov bx, 0x1000
    mov es, bx
    xor bx, bx
    int 0x13
    jc disk_error
    jmp 0x1000:0000
```

```
disk_error:
    mov si, msg_fail
    call print
    hlt

print:
    mov ah, 0x0E
.print_loop:
    lodsb
    or al, al
    jz .done
    int 0x10
    jmp .print_loop
.done:
    ret

boot_drive  db 0

msg_title   db 'Selecione o Kernel para boot:', 0x0D, 0x0A, 0
opt1        db '1) Kernel Padrao (CHS C0 H0 S2)', 0x0D, 0x0A, 0
opt2        db '2) Kernel Alternativo (CHS C0 H0 S3)', 0x0D, 0x0A, 0
opt3        db '3) Kernel Diagnostico (CHS C0 H0 S4)', 0x0D, 0x0A, 0
msg_fail    db 'Erro ao carregar o kernel.', 0x0D, 0x0A, 0

; Assinatura de boot
times 510-($-$$) db 0
dw 0xAA55
```

`mbr.asm`: Bootloader principal com menu de seleção.

Kernel 1, 2 e 3: Os kernels foram escritos inteiramente em Assembly, operando em modo real (16 bits), e são carregados diretamente pelo bootloader conforme a escolha do usuário no menu.

O kernel principal (Kernel 1) apresenta um prompt de comandos que permite interação básica com o usuário. Ao digitar a palavra "poweroff", o kernel envia um comando via porta I/O (0x604) que solicita o desligamento da máquina virtual no QEMU. Caso outro texto seja digitado, uma mensagem de erro é exibida e o prompt é reiniciado. Essa estrutura demonstra a execução de código diretamente após o boot, simulando funcionalidades mínimas de um sistema operacional.

```
; ------------------------------------------------------------------
; kernel1.asm - Kernel interativo para desligar com 'poweroff'
; ------------------------------------------------------------------

bits 16
org 0x0

; ------------------------------------------------------------------
; Ponto de Entrada do Kernel
; ------------------------------------------------------------------
_start:
    ; Configura os registradores de segmento para o segmento de código
    mov ax, 0x1000
    mov ds, ax
    mov es, ax
    mov ss, ax
    mov sp, 0xFFFE

    ; Limpa a tela
    mov ax, 0x03
    int 0x10

    ; Exibe a mensagem de inicialização
    mov si, kernel_message
    call print_string

; ------------------------------------------------------------------
; Loop Principal do Kernel
; ------------------------------------------------------------------
main_loop:
    ; Exibe o prompt '>'
    mov si, prompt_message
    call print_string

    ; Define o buffer de entrada e le a linha
    mov di, input_buffer
    call read_line

    ; Compara a entrada com o comando 'poweroff'
    mov si, poweroff_command
    mov di, input_buffer
    call compare_strings

    cmp ax, 0
    jne .not_poweroff

    ; Se as strings sao iguais, desliga o sistema
    call shutdown_qemu

.not_poweroff:
    mov si, invalid_command_message
    call print_string

    ; Volta para o loop principal
    jmp main_loop

```

```
; ------------------------------------------------------------------
; Rotina de Leitura de Linha
; Lê do teclado, exibe os caracteres na tela e armazena em um buffer
; ------------------------------------------------------------------
read_line:
    mov cx, 255 ; Limite de caracteres para o buffer
.read_loop:
    mov ah, 0x00 ; Funcao BIOS: ler caractere
    int 0x16     ; Espera por um caractere

    cmp al, 0x0D ; Verifica se a tecla Enter foi pressionada
    je .done_read

    cmp al, 0x08 ; Verifica se a tecla Backspace foi pressionada
    je .handle_backspace

    mov ah, 0x0E ; Funcao BIOS: exibir caractere
    int 0x10

    mov [di], al
    inc di
    loop .read_loop

    jmp .read_loop

.handle_backspace:
    cmp di, input_buffer
    je .read_loop

    dec di
    mov byte [di], 0

    mov ah, 0x0E
    mov al, 0x08 ; Caractere Backspace
    int 0x10

    mov al, ' '
    int 0x10

    mov al, 0x08
    int 0x10

    jmp .read_loop

.done_read:
    mov byte [di], 0 ; Termina a string com um byte nulo
    mov si, newline
    call print_string
    ret
```

```
    ; ---------------------------------------------------------------
    ; Rotina de Comparação de Strings
    ; Compara a string em DS:SI com a string em ES:DI
    ; Retorna AX = 0 se forem iguais, != 0 se forem diferentes
    ; ---------------------------------------------------------------
compare_strings:
    push si
    push di
    push cx
    xor ax, ax
.compare_loop:
    mov al, [si]
    mov bl, [di]
    cmp al, bl
    jne .not_equal
    cmp al, 0
    je .equal_check
    inc si
    inc di
    jmp .compare_loop
.not_equal:
    mov ax, 1
    jmp .done_compare
.equal_check:
    cmp bl, 0
    jne .not_equal
.done_compare:
    pop cx
    pop di
    pop si
    ret

    ; ---------------------------------------------------------------
    ; Funções Utilitárias e de Desligamento
    ; ---------------------------------------------------------------
print_string:
    mov ah, 0x0E
.print_loop:
    lodsb
    or al, al
    jz .done
    int 0x10
    jmp .print_loop
.done:
    ret

shutdown_qemu:
    mov dx, 0x604
    mov ax, 0x2000
    out dx, ax
    jmp $ ; Fallback loop

    ; ---------------------------------------------------------------
    ; Dados
    ; ---------------------------------------------------------------
kernel_message          db 'Kernel 1 carregado com sucesso!', 0x0D, 0x0A, 0
prompt_message          db '> ', 0
poweroff_command        db 'poweroff', 0
invalid_command_message db 'Comando invalido!', 0x0D, 0x0A, 0
newline                 db 0x0D, 0x0A, 0
input_buffer            times 256 db 0
```

`kernel1.asm`, `kernel2.asm`, `kernel3.asm`: Kernels simples com funcionalidades básicas.

MakeFile: O Makefile automatiza todo o processo de compilação e montagem do projeto. Ele organiza os diretórios de build, compila o bootloader (mbr.asm) e os três kernels em arquivos binários (.bin), e gera uma imagem de disco (disk.img) com os arquivos posicionados corretamente nos setores esperados. O Makefile também utiliza o utilitário dd para criar e manipular a imagem do disco de forma controlada, garantindo que o sistema possa ser executado no QEMU de forma reprodutível e confiável. (`Makefile`: Automatiza compilação e criação da imagem de disco.)

```
# Diretórios
SRC_DIR := kernel
BOOT_DIR := bootloader
BUILD_DIR := build
DISK_DIR := disk

# Arquivos de origem
MBR_ASM := $(BOOT_DIR)/mbr.asm
KERNEL1_ASM := $(SRC_DIR)/kernel1.asm
KERNEL2_ASM := $(SRC_DIR)/kernel2.asm
KERNEL3_ASM := $(SRC_DIR)/kernel3.asm

# Arquivos de saída
MBR_BIN := $(BUILD_DIR)/mbr.bin
KERNEL1_BIN := $(BUILD_DIR)/kernel1.bin
KERNEL2_BIN := $(BUILD_DIR)/kernel2.bin
KERNEL3_BIN := $(BUILD_DIR)/kernel3.bin
DISK_IMG := $(DISK_DIR)/disk.img

# Criação dos diretórios de build, se não existirem
$(BUILD_DIR):
	mkdir -p $(BUILD_DIR)
$(DISK_DIR):
	mkdir -p $(DISK_DIR)

# Comando padrão
default: $(DISK_IMG)

# Criação da imagem de disco
$(DISK_IMG): $(MBR_BIN) $(KERNEL1_BIN) $(KERNEL2_BIN) $(KERNEL3_BIN) | $(DISK_DIR)
	@echo "Criando imagem de disco..."
	dd if=/dev/zero of=$(DISK_IMG) bs=512 count=2880
	dd if=$(MBR_BIN) of=$(DISK_IMG) bs=512 seek=0 conv=notrunc
	dd if=$(KERNEL1_BIN) of=$(DISK_IMG) bs=512 seek=2 conv=notrunc
	dd if=$(KERNEL2_BIN) of=$(DISK_IMG) bs=512 seek=3 conv=notrunc
	dd if=$(KERNEL3_BIN) of=$(DISK_IMG) bs=512 seek=4 conv=notrunc

# Compilação do MBR
$(MBR_BIN): $(MBR_ASM) | $(BUILD_DIR)
	@echo "Compilando MBR..."
	nasm -f bin -o $@ $<

# Compilação do Kernel em Assembly
$(KERNEL1_BIN): $(KERNEL1_ASM) | $(BUILD_DIR)
	@echo "Compilando Kernel 1..."
	nasm -f bin -o $@ $<

$(KERNEL2_BIN): $(KERNEL2_ASM) | $(BUILD_DIR)
	@echo "Compilando Kernel 2..."
	nasm -f bin -o $@ $<

$(KERNEL3_BIN): $(KERNEL3_ASM) | $(BUILD_DIR)
	@echo "Compilando Kernel 3..."
	nasm -f bin -o $@ $<

# Limpeza
clean:
	@echo "Limpando arquivos..."
	rm -f $(BUILD_DIR)/*.bin $(DISK_IMG)
```

Conclusão

O desenvolvimento deste projeto proporcionou uma imersão prática nos conceitos fundamentais do processo de boot de sistemas operacionais, desde o carregamento inicial pelo bootloader até a execução de um kernel funcional. A utilização do QEMU como plataforma de simulação permitiu testar e validar cada etapa de forma controlada e acessível, enquanto a inspiração no GRUB guiou a estruturação de um gerenciador de boot simples, mas funcional.

A implementação em Assembly reforçou a compreensão de baixo nível sobre o funcionamento da arquitetura x86 e a interação direta com o hardware via BIOS. Ao final, o projeto cumpre seu objetivo didático ao demonstrar, de forma enxuta e eficaz, como um sistema pode ser carregado e executado a partir do zero, sem depender de sistemas operacionais pré-existentes ou ferramentas complexas.