

Como criar um Sistema Operacional | Unix

27 DE AGOSTO DE 2015

Índice

1. Introdução à arquitetura x86 e sobre nosso SO
2. Configuração do ambiente de desenvolvimento
3. Primeiro boot com GRUB
4. Backbone do SO e runtime C++
5. Classes de base para o gerenciamento da arquitetura x86
6. GDT
7. IDT e interrupções
8. Teoria: memória física e virtual
9. Gerenciamento de memória: física e virtual

Autor: *Samy Pesse*

Traduzido por: *Ygor Máximo*

Introdução

Esse é um livro digital sobre como escrever um sistema operacional em C/C++ a partir do zero.

Aviso: Esse repositório é uma alteração do meu antigo curso. Foi escrito muitos anos atrás **como um dos meus primeiros projetos quando eu estava no Ensino Médio**, e eu continuo melhorando algumas partes. O curso original estava em francês e eu não sou um nativo em inglês. Eu irei continuar e aprimorar esse curso no meu tempo livre.

Livro: Uma versão online está disponível em <http://samypesse.gitbooks.io/how-to-create-an-operating-system/> (PDF, Mobi e ePub). Eles são gerados utilizando o **GitBook**.

Código Fonte: Todo o código fonte do sistema será armazenado no diretório **src**. Cada passo irá conter links para os diferentes arquivos relacionados.

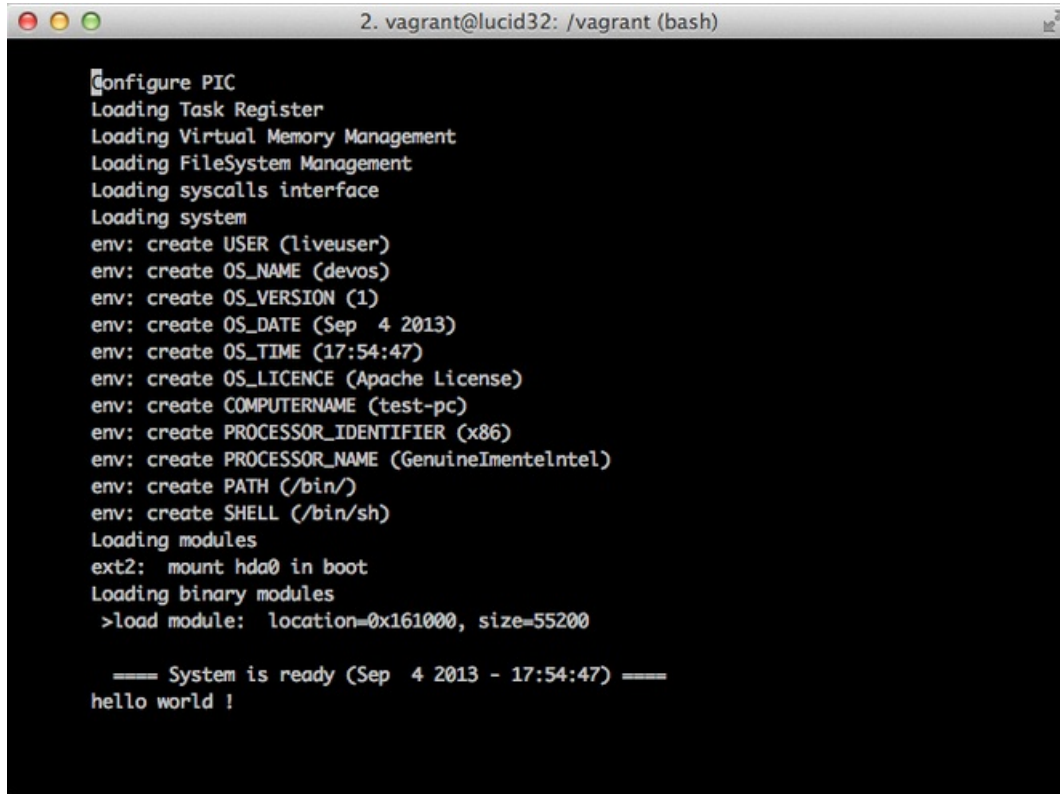
Contribuições: Esse curso está aberto à contribuições, sinta-se livre para informar erros criando issues ou corrigindo diretamente os erros com pull-requests.

Perguntas: Sinta-se livre para fazer qualquer pergunta adicionando issues. Por favor, não me

mande e-mails.

Você pode me seguir no Twitter [@SamyPesse](#) ou me ajudar no [Flattr](#) ou [Gittip](#).

Que tipo de SO estaremos construindo?



```
2. vagrant@lucid32: /vagrant (bash)
Configure PIC
Loading Task Register
Loading Virtual Memory Management
Loading FileSystem Management
Loading syscalls interface
Loading system
env: create USER (liveuser)
env: create OS_NAME (devos)
env: create OS_VERSION (1)
env: create OS_DATE (Sep  4 2013)
env: create OS_TIME (17:54:47)
env: create OS_LICENCE (Apache License)
env: create COMPUTERNAME (test-pc)
env: create PROCESSOR_IDENTIFIER (x86)
env: create PROCESSOR_NAME (GenuineIntelIntel)
env: create PATH (/bin/)
env: create SHELL (/bin/sh)
Loading modules
ext2: mount hda0 in boot
Loading binary modules
>load module: location=0x161000, size=55200

==== System is ready (Sep  4 2013 - 17:54:47) ====
hello world !
```

Introdução à arquitetura x86 e sobre nosso SO

O termo x86 denota uma família de arquiteturas de conjunto de instruções compatíveis com versões anteriores com base no 8086 CPU Intel.

A arquitetura x86 é a arquitetura de conjunto de instruções mais comum desde sua introdução em 1981 para o PC IBM. Uma grande quantidade de software, incluindo sistemas operacionais (SO's) como DOS, Windows, Linux, BSD, Solaris e Mac OS X, funcionam com hardware baseado na arquitetura x86.

Neste curso, nós não iremos desenvolver um sistema operacional para a arquitetura x86-64 mas para x86-32, graças a compatibilidade com versões anteriores, nosso SO será compatível com nossos novos PCs (mas tenha cuidado se você quer testá-lo em sua máquina real).

Nosso sistema operacional

O objetivo é construir um sistema operacional baseado em UNIX muito simples em C++, não apenas uma “prova de conceito”. O SO deve ser capaz de dar boot, iniciar uma shell e ser extensível.

O SO será construído para a arquitetura x86, rodando sobre 32 bits e compatível com PCs IBM.

Especificações:

- Código em C++
- Arquitetura x86, 32 bit
- Boot com GRUB
- Tipo de sistema modular para drivers
- Tipo de estilo UNIX
- Multitasking
- ELF executável no userland
- Módulos (acessível no userland usando /dev/...):
 - Discos IDE
 - Partições DOS
 - Clock
 - EXT2 (somente leitura)
 - Bochs VBE
- Userland:
 - API Posix
 - LibC
 - Pode rodar uma shell ou alguns executáveis (e.g., lua)

Configuração do ambiente de desenvolvimento

O primeiro passo é configurar um bom e viável ambiente de desenvolvimento. Usando Vagrant e VirtualBox, você será capaz de compilar e testar seu SO a partir do seu computador, sendo ele Linux, Windows ou Mac.

Instalar Vagrant

O Vagrant é um software livre de código aberto para criar e configurar ambientes de desenvolvimento virtual. Ele pode ser considerado um wrapper em torno do VirtualBox.

Vagrant irá nos ajudar a criar um ambiente de desenvolvimento virtual limpo em qualquer sistema que você esteja usando. O primeiro passo é fazer download e instalar o Vagrant para seu sistema no [site oficial](#).

Instalar VirtualBox

Oracle VM VirtualBox é um pacote de software de virtualização para computadores com arquitetura baseada em x86 e AMD64/Intel-64.

O Vagrant precisa do VirtualBox para funcionar, faça o download e instale no seu sistema no [site oficial](#).

Inicie e teste seu ambiente de desenvolvimento

Uma vez que o Vagrant e o VirtualBox estejam instalados, você precisa baixar a imagem do Ubuntu lucid32 para o Vagrant.

```
vagrant box add lucid32 http://files.vagrantup.com/lucid32.box
```

Uma vez que a imagem do lucid32 estiver pronta, nós precisamos determinar nosso ambiente de desenvolvimento usando Vagrantfile, crie um arquivo chamado Vagrantfile. Esse arquivo determina quais pré-requisitos nosso ambiente precisa: nasm, make, build-essential, grub e qemu.

Inicie usando:

```
vagrant up
```

Agora você pode acessar seu box usando ssh para conectar ao virtualbox usando:

```
vagrant ssh
```

O diretório contendo o *Vagrantfile* será montado por padrão no diretório */vagrant* do Ubuntu Lucid32:

```
cd /vagrant
```

Construa e teste nosso sistema operacional

O arquivo **Makefile** determina algumas regras básicas para a construção do kernel, o usuário `libc` e alguns programas do userland.

Construa:

```
make all
```

Teste nosso sistema operacional com *qemu*:

```
make run
```

A documentação para o *qemu* está disponível em [QEMU Emulator Documentation](#).

Você pode sair do emulador usando a tecla de atalho *Ctrl* + *A*.

Primeiro boot com GRUB

Como o boot funciona?

Quando um computador baseado em x86 é ligado, ele começa um caminho complexo para chegar ao estágio em que o controle é transferido à nossa rotina “main” do kernel (*kmain()*). Para este curso, nós vamos somente considerar o método de inicialização da BIOS e não seu sucessor (UEFI).

A sequência de Boot da BIOS é: detecção de memória RAM ~> Inicialização/Detecção de Hardware ~> Sequência de Boot.

O passo mais importante para nós é a “Sequência de Boot/Inicialização”, onde a BIOS já está pronta com sua inicialização e tenta transferir o controle para o próximo estágio do processo de *bootloader*.

Durante a “Sequência de Inicialização”, a BIOS irá tentar definir um “dispositivo de inicialização” (e.g. floppy disk, hard-disk, CD, dispositivo USB flash memory ou rede). Nosso sistema operacional irá, em primeira instância, inicializar a partir do hard-disk/HD (mas será possível inicializar a partir de um CD ou um dispositivo USB no futuro). Um dispositivo é considerado inicializável se o setor de inicialização conter a assinatura válida de bytes `0x55` e `0xAA` nos deslocamentos 511 e 512, respectivamente (chamados os bytes mágicos do Master Boot Record (MBR)). Essa assinatura é representada (em binário) como `0b1010101001010101`. O padrão de bit alternado foi idealizado para ser uma proteção contra certas falhas (drive ou controlador). Se esse padrão for ilegível ou `0x00`, o dispositivo não é considerado inicializável).

A BIOS pesquisa fisicamente por um dispositivo de inicialização carregando os primeiros 512 bytes do setor de inicialização de cada dispositivo na memória física, começando pelo endereço `0x7C00` (1 KiB abaixo da marca de 32 KiB). Quando a assinatura de bytes válida é detectada, a BIOS transfere o controle para o endereço de memória `0x7c00` (a partir de uma instrução de salto) a fim de executar o código do setor de inicialização.

Durante todo este processo, a CPU foi executada em modo real 16-bit (o estado padrão para CPUs x86, a fim de manter a compatibilidade com versões anteriores). Para executar as instruções de 32 bits dentro do nosso kernel, um gerenciador de inicialização é necessário para mudar a CPU para o Modo Protegido.

O que é GRUB?

O GNU GRUB (abreviação para GNU Grand Unified Bootloader) é um pacote carregador de inicialização do Projeto GNU. GRUB é a implementação de referência da especificação Multiboot da Free Software Foundation (FSF) que fornece ao usuário a escolha de iniciar um dos múltiplos sistemas operacionais instalados em um computador ou selecionar uma configuração de kernel específica disponível em partições de um sistema operacional em particular.

Para facilitar, GRUB é a primeira coisa que é iniciada pela máquina (um carregador de inicialização) e que irá simplificar o carregamento do nosso kernel armazenado no HD.

Por que estamos usando GRUB?

- GRUB é muito simples de usar
- Ele torna muito simples o carregamento de kernels de 32 bits sem precisar do código de 16 bits
- Multiboot com Linux, Windows e outros
- Facilita o carregamento de módulos externos na memória

Como usar GRUB?

O GRUB usa a especificação Multiboot, o binário executável deve ser de 32 bits e deve conter um cabeçalho especial (cabeçalho de multiboot) nos seus 8192 primeiros bytes. Nosso kernel será um arquivo executável ELF (“Executable and Linkable Format”, um formato padrão comum para executáveis na maioria dos sistemas UNIX).

A primeira sequência de inicialização do nosso kernel está escrita em Assembly: `start.asm` e nós

usamos um arquivo linkador para definir a estrutura do nosso executável: [linker.ld](#).

Esse processo de inicialização também inicia alguns dos nossos runtime C++, ele será descrito no próximo capítulo.

Estrutura do cabeçalho de multiboot:

```
struct multiboot_info {  
    u32 flags;  
    u32 low_mem;
```

```
1      struct multiboot_info {  
2          u32 flags;  
3          u32 low_mem;  
4          u32 high_mem;  
5          u32 boot_device;  
6          u32 cmdline;  
7          u32 mods_count;  
8          u32 mods_addr;  
9          struct {  
10             u32 num;  
11             u32 size;  
12             u32 addr;  
13             u32 shndx;  
14         } elf_sec;  
15         unsigned long mmap_length;  
16         unsigned long mmap_addr;  
17         unsigned long drives_length;  
18         unsigned long drives_addr;  
19         unsigned long config_table;  
20         unsigned long boot_loader_name;  
21         unsigned long apm_table;  
22         unsigned long vbe_control_info;  
23         unsigned long vbe_mode_info;  
24         unsigned long vbe_mode;  
25         unsigned long vbe_interface_seg;  
26         unsigned long vbe_interface_off;  
27         unsigned long vbe_interface_len;  
28     };
```

Você pode usar o comando *mbchk kernel.elf* para validar seu arquivo kernel.elf contra o padrão de multiboot. Você pode também usar o comando *nm -n kernel.elf* para validar o deslocamento dos diferentes objetos no binário ELF.

Crie uma imagem de disco para nosso kernel e GRUB

O script [diskimage.sh](#) irá gerar uma imagem de disco que pode ser usada pelo QEMU.

O primeiro passo é criar uma imagem de disco (c.img) usando qemu-img

```
qemu-img create c.img 2M
```

Agora nós precisamos particionar o disco usando fdisk:

```
fdisk ./c.img
```

```
# Switch to Expert commands
```

```
1      fdisk ./c.img
2
3      # Switch to Expert commands
4      &gt; x
5
6      # Change number of cylinders (1-1048576)
7      &gt; c
8      &gt; 4
9
10     # Change number of heads (1-256, default 16):
11     &gt; h
12     &gt; 16
13
14     # Change number of sectors/track (1-63, default 63)
15     &gt; s
16     &gt; 63
17
18     # Return to main menu
19     &gt; r
20
21     # Add a new partition
22     &gt; n
23
24     # Choose primary partition
25     &gt; p
26
27     # Choose partition number
28     &gt; 1
29
30     # Choose first sector (1-4, default 1)
31     &gt; 1
32
33     # Choose last sector, +cylinders or +size{K,M,G} (1-4, default 4)
34     &gt; 4
35
36     # Toggle bootable flag
37     &gt; a
38
39     # Choose first partition for bootable flag
40     &gt; 1
41
42     # Write table to disk and exit
43     &gt; w
```

Agora nós precisamos unir a partição criada ao dispositivo de loop (que permite um arquivo ser acessado como um dispositivo de bloco) usando `losetup`. O deslocamento da partição é passado como argumento e calculado usando: **deslocamento= setor_inicial * bytes_por_setor**

Usando `fdisk -l -u c.img`, você tem: $63 * 512 = 32256$.

```
losetup -o 32256 /dev/loop1 ./c.img
```

Nós criamos um sistema de arquivos EXT2 neste novo dispositivo usando:

```
mke2fs /dev/loop1
```

Nós copiamos nossos arquivos em um disco montado:

```
mount /dev/loop1 /mnt/
cp -R bootdisk/* /mnt/
```


`umount /mnt/`

Instale o GRUB no disco:

```
grub -device-map=/dev/null &&& EOF
device (hd0) ./c.img
geometry (hd0) 4 16 63
```

```
1      grub -device-map=/dev/null &&& EOF
2      device (hd0) ./c.img
3      geometry (hd0) 4 16 63
4      root (hd0,0)
5      setup (hd0)
6      quit
7      EOF
```

E, finalmente, nós desmontamos o dispositivo de loop:

`losetup -d /dev/loop1`

Veja também

- [GNU GRUB na Wikipedia](#)
- [Especificação Multiboot](#)

Backbone do SO e runtime C++

Runtime C++ do kernel

Um kernel pode ser programado em C++, é muito semelhante a fazer um kernel em C, exceto que há algumas armadilhas que você tem que levar em conta (suporte runtime, constructors, ...).

O compilador irá supor que todo o suporte de runtime C++ necessário está disponível por padrão, mas como não estamos linkando em `libsupc++` em seu kernel do C++, nós precisamos adicionar algumas funções básicas que podem ser encontrados no arquivo `CXX.CC`.

Aviso: Os operadores *new* e *delete* não podem ser usados antes da memória virtual e paginação terem sido inicializados.

Funções básicas C/C++

O código do kernel não pode usar funções das bibliotecas padrão então nós precisamos adicionar algumas funções básicas para o gerenciamento de memória e strings:

```
void itoa(char *buf, unsigned long int n, int base);
void * memset(char *dst, char src, int n);
void * memcpy(char *dst, char *src, int n);
```

```
1 void itoa(char *buf, unsigned long int n, int base);
2 void * memset(char *dst, char src, int n);
3 void * memcpy(char *dst, char *src, int n);
4 int strlen(char *s);
5 int strcmp(const char *dst, char *src);
6 int strcpy(char *dst, const char *src);
7 void strcat(void *dest, const void *src);
8 char *strncpy(char *destString, const char *sourceString, int maxLength);
9 int strncmp( const char* s1, const char* s2, int c );
```

Essas funções são definidas em [string.cc](#), [memory.cc](#), [itoa.cc](#)

Tipos C

Durante o próximo passo, nós iremos usar diferentes tipos no nosso código. Na maioria dos tipos, nós usaremos tipos unsigned (todos os bits são usados para armazenar o número inteiro. Em tipos signed, um bit é usado para sinalizar o sinal)

```
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned int u32;
```

```
1 typedef unsigned char u8;
2 typedef unsigned short u16;
3 typedef unsigned int u32;
4 typedef unsigned long long u64;
5
6 typedef signed char s8;
7 typedef signed short s16;
8 typedef signed int s32;
9 typedef signed long long s64;
```

Compile nosso kernel

Compilar um kernel não é a mesma coisa que compilar um executável linux, nós não podemos usar uma biblioteca padrão e não se deve ter dependências para o sistema.

Nosso arquivo **Makefile** irá determinar o processo para compilar e linkar nosso kernel.

Para arquitetura x86, os seguintes argumentos serão usados para gcc/g++/ld:

```
# Linker
LD=ld
LDFLAGS=-melf_i386 -static -L ./ -T ./arch/$(ARCH)/linker.ld
```

```
1 # Linker
2 LD=ld
```

```
3 LDFLAG= -melf_i386 -static -L ./ -T ./arch/$(ARCH)/linker.ld
4
5 # C++ compiler
6 SC=g++
7 FLAG= $(INCDIR) -g -O2 -w -trigraphs -fno-builtin -fno-exceptions -fno-stack-protector -O0 -m32 -fno-rtti -nostdlib -nodefaultlibs
8
9 # Assembly compiler
10 ASM=nasm
11 ASMFLAG=-f elf -o
```

Classes de base para o gerenciamento da arquitetura

Agora que sabemos como compilar nosso kernel C++ e inicializar o binário usando GRUB, nós podemos começar a fazer algumas coisas legais em C/C++.

Imprimindo na tela console

Nós iremos usar o modo padrão VGA (03h) para mostrar algum texto para o usuário. A tela pode ser diretamente acessada usando a memória de vídeos em `0xB8000`. A resolução da tela é de 80×25 e cada caracter na tela é determinado por 2 bytes: um para o código de caracteres e um para o estilo. Isso significa que o tamanho total da memória de vídeo é 4000B (80B25B2B).

Na classe IO (`io.cc`):

- **x, y**: define a posição do cursor na tela
- **real_screen**: define o ponteiro da memória de vídeo
- **putc(char c)**: mostra um único caracter na tela e gerencia a posição do cursor
- **printf(char* s, ...)**: mostra uma string

Nós adicionamos um método **putc** à **classe IO** para colocar um caracter na tela e atualizar a posição (x, y).

```
/* put a byte on screen */
void io::putc(char c){
    kattr = 0x07;
```

```
1      /* put a byte on screen */
2      void io::putc(char c){
3          kattr = 0x07;
4          unsigned char *video;
5          video = (unsigned char *) (real_screen + 2 * x + 160 * y);
6          // newline
7          if (c == '\n') {
8              x = 0;
9              y++;
10             // back space
11         } else if (c == '\b') {
12             if (x) {
13                 *(video + 1) = 0x0;
14                 x--;
15             }
16         }
```

```

16      // horizontal tab
17      } else if (c == '\t') {
18          x = x + 8 - (x % 8);
19      // carriage return
20      } else if (c == '\r') {
21          x = 0;
22      } else {
23          *video = c;
24          *(video + 1) = kattr;
25
26          x++;
27          if (x > 79) {
28              x = 0;
29              y++;
30          }
31      }
32      if (y > 24)
33          scrollup(y - 24);
34  }

```

Nós também adicionamos um método muito útil e conhecido: **printf**

```

/* put a string in screen */
void lo::print(const char *s, ...){
    va_list ap;

```

```

1      /* put a string in screen */
2      void lo::print(const char *s, ...){
3          va_list ap;
4
5          char buf16;
6          int i, j, size, buflen, neg;
7
8          unsigned char c;
9          int ival;
10         unsigned int uival;
11
12         va_start(ap, s);
13
14         while ((c = *s++)) {
15             size = 0;
16             neg = 0;
17
18             if (c == 0)
19                 break;
20             else if (c == '%') {
21                 c = *s++;
22                 if (c &gt;= '0' &amp; c <= '9') {
23                     size = c - '0';
24                     c = *s++;
25                 }
26
27                 if (c == 'd') {
28                     ival = va_arg(ap, int);
29                     if (ival < 0) {
30                         uival = 0 - ival;
31                         neg++;
32                     } else
33                         uival = ival;
34                     itoa(buf, uival, 10);
35
36                     buflen = strlen(buf);
37                     if (buflen &lt; size) for (i = size, j = buflen; i &gt;= 0;
38                         i--, j--)
39                         bufi =
40                             (j &gt;=
41                             0) ? bufj : '0';
42
43                     if (neg)
44                         print("%s", buf);
45                     else
46                         print(buf);
47                 }
48             else if (c == 'u') {
49                 uival = va_arg(ap, int);
50                 itoa(buf, uival, 10);
51
52                 buflen = strlen(buf);
53                 if (buflen &lt; size) for (i = size, j = buflen; i &gt;= 0;

```

```

54         i--, j--)
55         bufi =
56         (j &gt;=
57         0) ? bufj : '0';
58
59         print(buf);
60     } else if (c == 'x' || c == 'X') {
61         uival = va_arg(ap, int);
62         itoa(buf, uival, 16);
63
64         buflen = strlen(buf);
65         if (buflen &lt; size) for (i = size, j = buflen; i &gt;= 0;
66             i--, j--)
67             bufi =
68             (j &gt;=
69             0) ? bufj : '0';
70
71         print("0x%s", buf);
72     } else if (c == 'p') {
73         uival = va_arg(ap, int);
74         itoa(buf, uival, 16);
75         size = 8;
76
77         buflen = strlen(buf);
78         if (buflen &lt; size) for (i = size, j = buflen; i &gt;= 0;
79             i--, j--)
80             bufi =
81             (j &gt;=
82             0) ? bufj : '0';
83
84         print("0x%s", buf);
85     } else if (c == 's') {
86         print((char *) va_arg(ap, int));
87     }
88     } else
89         putc(c);
90 }
91
92 return;
93 }

```

Interface Assembly

Uma grande quantidade de instruções estão disponíveis em Assembly mas não há equivalente em C (como cli, sti, in e out), então nós precisamos de uma interface para essas instruções.

Em C, nós podemos incluir Assembly usando a diretiva “asm()”, gcc usa *gas* para compilar o assembly.

Aviso: *gas* usa a sintaxe AT&T.

```

/* output byte */
void lo::outb(u32 ad, u8 v){
    asmv("outb %%al, %%dx" :: "d" (ad), "a" (v));;
}

```

```

1      /* output byte */
2      void lo::outb(u32 ad, u8 v){
3          asmv("outb %%al, %%dx" :: "d" (ad), "a" (v));;
4      }
5      /* output word */
6      void lo::outw(u32 ad, u16 v){
7          asmv("outw %%ax, %%dx" :: "d" (ad), "a" (v));
8      }
9      /* output word */
10     void lo::outl(u32 ad, u32 v){
11         asmv("outl %%eax, %%dx" :: "d" (ad), "a" (v));
12     }
13     /* input byte */
14     u8 lo::inb(u32 ad){

```

```
15         u8 _v; \
16         asmv("inb %%dx, %%al" : "=a" (_v) : "d" (ad)); \
17         return _v;
18     }
19     /* input word */
20     u16 lo::inw(u32 ad){
21         u16 _v; \
22         asmv("inw %%dx, %%ax" : "=a" (_v) : "d" (ad)); \
23         return _v;
24     }
25     /* input word */
26     u32 lo::inl(u32 ad){
27         u32 _v; \
28         asmv("inl %%dx, %%eax" : "=a" (_v) : "d" (ad)); \
29         return _v;
30     }
```

GDT

Graças ao GRUB, seu kernel não está mais em Modo Real (real-mode), mas já em **modo protegido**, esse modo nos permite usar todas as possibilidades do microprocessador como gerenciamento da memória virtual, paginação e um multitasking seguro.

O que é GDT:

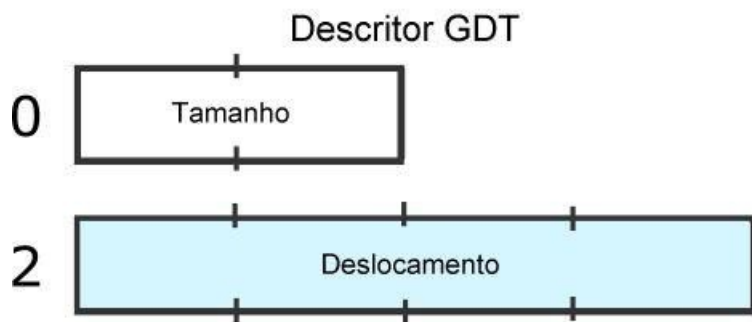
O **GDT** (“Global Descriptor Table”) é uma estrutura de dados usada para definir as diferentes áreas da memória: o endereço base, o tamanho e privilégios de acesso como execução e escrita. Essas áreas da memória são chamadas “segmentos”.

Nós iremos usar o GDT para determinar diferentes segmentos da memória.

- “code”: código do kernel, usado para armazenar o código binário executável
- “data”: dados do kernel
- “stack”: pilha do kernel, usada para armazenar a chamada da pilha durante a execução do kernel
- “ucode”: código do usuário, usado para armazenar o código binário executável para o programa do usuário
- “udata”: dados do programa do usuário
- “ustack”: pilha do usuário, usada para armazenar a chamada da pilha durante a execução do userland

Como carregar nosso GDT?

O GRUB inicializa um GDT mas este GDT não corresponde ao nosso kernel. O GDT é carregado usando a instrução assembly LGDT. Ela espera a localização de uma descrição de estrutura do GDT:



E a estrutura C:

```
struct gdt {
    u16 limite;
    u32 base;
```

```
1 struct gdt {
2     u16 limite;
3     u32 base;
4 } __attribute__((packed));
```

Aviso: a diretiva `__attribute__((packed))` sinaliza para o gcc que a estrutura deve usar a menor quantidade de memória possível. Sem essa diretiva, o gcc inclui alguns bytes para otimizar o alinhamento da memória e o acesso durante a execução.

Agora precisamos definir nossa tabela GDT e então carregá-la usando LGDT. A tabela GDT pode ser armazenada em qualquer lugar que quisermos na memória, seu endereço deve apenas ser sinalizado para o processo usando o registro GDTR.

A tabela GDT é composta por segmentos com a seguinte estrutura:



E a estrutura C:

```
struct gdt {
    u16 lim0_15;
    u16 base0_15;
```

```
1 struct gdt {
2     u16 lim0_15;
3     u16 base0_15;
```

```
4         u8 base16_23;
5         u8 acces;
6         u8 lim16_19:4;
7         u8 other:4;
8         u8 base24_31;
9     } __attribute__((packed));
```

Como determinar nosso GDT?

Agora nós precisamos definir nosso GDT na memória e finalmente carregá-la usando o registro GDTR.

Nós iremos armazenar nosso GDT no seguinte endereço:

```
#define GDTBASE 0x00000800
```

A função **init_gdt_desc** em x86.cc inicializa um descritor de segmento GDT.

```
void init_gdt_desc(u32 base, u32 limite, u8 acces, u8 other, struct gtdesc *desc)
{
    desc->lim0_15 = (limite & 0xffff);

1    void init_gdt_desc(u32 base, u32 limite, u8 acces, u8 other, struct gtdesc *desc)
2    {
3        desc->lim0_15 = (limite & 0xffff);
4        desc->base0_15 = (base & 0xffff);
5        desc->base16_23 = (base & 0xffff0000) >> 16;
6        desc->acces = acces;
7        desc->lim16_19 = (limite & 0xffff0000) >> 16;
8        desc->other = (other & 0xf);
9        desc->base24_31 = (base & 0xffff00000) >> 24;
```

E a função **init_gdt** inicializa o GDT, algumas partes da função abaixo irão ser explicadas mais tarde e são usadas para multitasking.

```
void init_gdt(void)
{
    default_tss.debug_flag = 0x00;

1    void init_gdt(void)
2    {
3        default_tss.debug_flag = 0x00;
4        default_tss.io_map = 0x00;
5        default_tss.esp0 = 0x1FFF0;
6        default_tss.ss0 = 0x18;
7
8        /* initialize gdt segments */
9        init_gdt_desc(0x0, 0x0, 0x0, 0x0, &kgdt0);
10       init_gdt_desc(0x0, 0xFFFFF, 0x9B, 0x0D, &kgdt1); /* code */
11       init_gdt_desc(0x0, 0xFFFFF, 0x93, 0x0D, &kgdt2); /* data */
12       init_gdt_desc(0x0, 0x0, 0x97, 0x0D, &kgdt3); /* stack */
13
14       init_gdt_desc(0x0, 0xFFFFF, 0xFF, 0x0D, &kgdt4); /* ucode */
15       init_gdt_desc(0x0, 0xFFFFF, 0xF3, 0x0D, &kgdt5); /* udata */
16       init_gdt_desc(0x0, 0x0, 0xF7, 0x0D, &kgdt6); /* ustack */
17
18       init_gdt_desc((u32) & default_tss, 0x67, 0xE9, 0x00, &kgdt7); /* descripteur de tss */
19
```



```

20      /* initialize the gdt structure */
21      kgdtr.limite = GDTSIZE * 8;
22      kgdtr.base = GDTBASE;
23
24      /* copy the gdt to its memory area */
25      memcpy((char *) kgdtr.base, (char *) kgdt, kgdtr.limite);
26
27      /* load the gdt registry */
28      asm("lgdtl (kgdtr)");
29
30      /* initiliaz the segments */
31      asm("    movw $0x10, %ax  \n \
32          movw %ax, %ds  \n \
33          movw %ax, %es  \n \
34          movw %ax, %fs  \n \
35          movw %ax, %gs  \n \
36          ljmp $0x08, $next  \n \
37          next:  \n");
38  }

```

IDT e interrupções

Uma interrupção é um sinal para o processador emitido pelo hardware ou software indicando um evento que precisa de atenção imediata.

Há três tipos de interrupções:

- **Interrupções de Hardware:** são enviadas para o processador a partir de um dispositivo externo (teclado, mouse, HD, ...). Interrupções de Hardware foram introduzidas como uma forma de reduzir o desperdício do valioso tempo do processador em loops de captação, esperando por eventos externos.
- **Interrupções de Software:** são iniciadas voluntariamente pelo software. É usada para gerenciar chamadas de sistema.
- **Exceptions/Exceções:** são usadas para erros ou eventos ocorrendo durante a execução de um programas no qual eles são excepcionais o suficiente que não podem ser tratadas pelo programa em si (divisão por zero, falha de paginação, ...)

O exemplo de teclado:

Quando o usuário pressiona um botão no teclado, o controlador do teclado irá sinalizar uma interrupção para o controlador de interrupção (Interrupt Controller). Se a interrupção não for mascarada, o controlador irá sinalizar a interrupção para o processador, o processador irá executar uma rotina para gerenciar a interrupção (botão pressionado ou botão liberado), essa rotina pode, por exemplo, pegar o botão pressionado do controlador do teclado e imprimir o respectivo caracter na tela. Uma vez que a rotina de processamento de caracteres é concluída, o trabalho interrompido pode ser retomado.

O que é o PIC?

O PIC (Programmable Interrupt Controller) é um dispositivo que é usado para combinar várias fontes

de interrupções em uma ou mais linhas de CPU, permitindo níveis de prioridade a ser atribuído a suas saídas de interrupção. Quando o dispositivo tem múltiplas saídas de interrupção para declarar, ele as declara na ordem de suas relativas prioridades.

O PIC mais conhecido é o 8259A, cada 8259A pode manuseiar 8 dispositivos mas a maioria dos computadores têm dois controladores: um mestre e um escravo, isso permite o computador gerenciar interrupções a partir de 14 dispositivos.

Neste capítulo, nós precisaremos programar esse controlador para inicializar e mascarar interrupções.

O que é o IDT?

O IDT (Interrupt Descriptor Table) é uma estrutura de dados usada pela arquitetura x86 para implementar uma tabela de vetor de interrupção. O IDT é usado pelo processador para determinar a resposta correta para interrupções e exceções.

Nosso kernel irá usar o IDT para definir as diferentes funções a serem executadas quando uma interrupção ocorrer.

Como o GDT, o IDT é carregado usando a instrução assembly LIDTL. Ele aguarda a localização de uma descrição de estrutura do IDT:

```
struct idtr {
    u16 limite;
    u32 base;
};
```

1 struct idtr {
2 u16 limite;
3 u32 base;
4 } __attribute__((packed));

A tabela IDT é composta de segmentos IDT com a seguinte estrutura:

```
struct idtdesc {
    u16 offset0_15;
    u16 select;
};
```

1 struct idtdesc {
2 u16 offset0_15;
3 u16 select;
4 u16 type;
5 u16 offset16_31;
6 } __attribute__((packed));

Aviso: a diretiva `_attribute__((packed))` sinaliza ao GCC que a estrutura deve usar a menor quantidade de memória possível. Sem essa diretiva, o GCC inclui alguns bytes para otimizar o alinhamento de memória e o acesso durante a execução.

Agora nós precisamos definir nossa tabela IDT e então carregá-la usando a LIDTL. A tabela IDT pode ser armazenada no lugar que quisermos na memória, seu endereço deve apenas ser sinalizado para o processo usando o registro IDTR.

Aqui está a tabela de interrupções comuns (Interrupções Maskable de hardware são chamados de IRQ):

IRQ	Descrição
0	Programmable Interrupt Timer Interrupt
1	Keyboard Interrupt
2	Cascade (usado internamente por dois PICs)
3	COM2 (se habilitado)
4	COM1 (se habilitado)
5	LPT2 (se habilitado)
6	Floppy Disk
7	LPT1
8	CMOS real-time clock (se habilitado)
9	Free for peripherals / legacy SCSI / NIC
10	Free for peripherals / SCSI / NIC
11	Free for peripherals / SCSI / NIC
12	PS2 Mouse
13	FPU / Coprocessor / Inter-processor
14	Primary ATA Hard Disk
15	Secondary ATA Hard Disk

Como inicializar as interrupções?

Esse é um simples método para definir um segmento IDT:

```
void init_idt_desc(u16 select, u32 offset, u16 type, struct idtdesc *desc)
{
    desc->offset0_15 = (offset & 0xffff);

1      void init_idt_desc(u16 select, u32 offset, u16 type, struct idtdesc *desc)
2      {
3          desc->offset0_15 = (offset & 0xffff);
4          desc->select = select;
5          desc->type = type;
6          desc->offset16_31 = (offset & 0xffff0000) >> 16;
7          return;
8      }
```

E agora nós podemos inicializar as interrupções:

```
#define IDTBASE 0x00000000
#define IDTSIZE 0xFF
idtr kidtr;

1      #define IDTBASE 0x00000000
2      #define IDTSIZE 0xFF
3      idtr kidtr;
```

```
void init_idt(void)
{
    /* Init irq */

1      void init_idt(void)
2      {
3          /* Init irq */
4          int i;
5          for (i = 0; i < IDTSIZE; i++) init_idt_desc(0x08, (u32) _asm_schedule, INTGATE, &akidt); // /* Vectors 0 -&gt; 31 are for
6          exceptions */
7          init_idt_desc(0x08, (u32) _asm_exc_GP, INTGATE, &kidt13); /* #GP */
8          init_idt_desc(0x08, (u32) _asm_exc_PF, INTGATE, &kidt14); /* #PF */
9
10         init_idt_desc(0x08, (u32) _asm_schedule, INTGATE, &kidt32);
11         init_idt_desc(0x08, (u32) _asm_int_1, INTGATE, &kidt33);
12
13         init_idt_desc(0x08, (u32) _asm_syscalls, TRAPGATE, &kidt48);
14         init_idt_desc(0x08, (u32) _asm_syscalls, TRAPGATE, &kidt128); //48
15
16         kidtr.limite = IDTSIZE * 8;
17         kidtr.base = IDTBASE;
18
19
20         /* Copy the IDT to the memory */
21         memcpy((char *) kidtr.base, (char *) kidt, kidtr.limite);
22
23         /* Load the IDTR registry */
24         asm("lidtl (%0);", &kidtr);
}
```

Depois da inicialização do nosso IDT, nós precisamos ativar as interrupções configurando o PIC. A seguinte função irá configurar os dois PICs escrevendo em seus registros internos usando portas de

saída do processador *io.outb*. Nós configuramos os PICs usando as portas:

- Master (Mestre) PIC: 0x20 e 0x21
- Slave (Escravo) PIC: 0xA0 e 0xA1

Para um PIC, há dois tipos de registros:

- ICW (Initialization Command Word): reinicia o controlador
- OCW (Operation Control Word): configure o controlador uma vez inicializado (usado para mascarar/desmascarar as interrupções)

```
void init_pic(void)
{
    /* Initialization of ICW1 */
```

```
1      void init_pic(void)
2      {
3          /* Initialization of ICW1 */
4          io.outb(0x20, 0x11);
5          io.outb(0xA0, 0x11);
6
7          /* Initialization of ICW2 */
8          io.outb(0x21, 0x20); /* start vector = 32 */
9          io.outb(0xA1, 0x70); /* start vector = 96 */
10
11         /* Initialization of ICW3 */
12         io.outb(0x21, 0x04);
13         io.outb(0xA1, 0x02);
14
15         /* Initialization of ICW4 */
16         io.outb(0x21, 0x01);
17         io.outb(0xA1, 0x01);
18
19         /* mask interrupts */
20         io.outb(0x21, 0x00);
21         io.outb(0xA1, 0x00);
22     }
```

Configurações de detalhe PIC ICW

Os registros têm que ser configurados em ordem.

ICW (porta 0x20 / porta 0xA0)

```
|0|0|0|1|x|0|x|x|
| | +--- com ICW4 (1) ou sem (0)
| +----- um controlador (1), ou cascade (0)
```

```
1      |0|0|0|1|x|0|x|x|
2      | | +--- com ICW4 (1) ou sem (0)
3      | +----- um controlador (1), ou cascade (0)
4      +----- desencadeando por nível (level) (1) ou pela borda (edge) (0)
```

ICW (porta 0x21 / porta 0xA1)

|x|x|x|x|x|0|0|0|

||||

+----- endereço de base para vetores de interrupção

1

|x|x|x|x|x|0|0|0|

2

||||

3

+----- endereço de base para vetores de interrupção

ICW (porta 0x21 / porta 0xA1)

Para o master:

|x|x|x|x|x|x|x|x|

|||||

+----- controlador slave conectado a porta sim (1), ou não (0)

1

|x|x|x|x|x|x|x|x|

2

|||||

3

+----- controlador slave conectado a porta sim (1), ou não (0)

Para o slave:

|0|0|0|0|0|x|x|x| pour l'esclave

|||

+----- ID slave que é igual à porta master

1

|0|0|0|0|0|x|x|x| pour l'esclave

2

|||

3

+----- ID slave que é igual à porta master

ICW4 (porta 0x21 / porta 0xA1)

É usado para definir em qual modo o controlador deve funcionar.

|0|0|0|x|x|x|x|1|

||| +----- modo "fim de interrupção automático" AEOL (1)

|| +----- modo buffered slave (0) ou master (1)

1

|0|0|0|x|x|x|x|1|

2

||| +----- modo "fim de interrupção automático" AEOL (1)

3

|| +----- modo buffered slave (0) ou master (1)

4

| +----- modo buffered (1)

5

+----- modo "totalmente encaixados" (1)

Por que segmentos IDT deslocam nossas funções ASM?

Você deve ter notado que quando eu estou inicializando nossos segmentos IDT, eu estou usando deslocamentos para segmentar o código em Assembly. As diferentes funções são definidas em [x86int.asm](#) e são do seguinte esquema:

```
%macro SAVE_REGS 0
    pushad
    push ds
```

```
1          %macro SAVE_REGS 0
2
3          pushad
4          push ds
5          push es
6          push fs
7          push gs
8          push ebx
9          mov bx,0x10
10         mov ds,bx
11         pop ebx
12     %endmacro
13
14
15         %macro RESTORE_REGS 0
16
17         pop gs
18         pop fs
19         pop es
20         pop ds
21         popad
22     %endmacro
23
24
25         %macro INTERRUPT 1
26
27         global _asm_int_%1
28         _asm_int_%1:
29             SAVE_REGS
30             push %1
31             call isr_default_int
32             pop eax    ;;a enlever sinon
33             mov al,0x20
34             out 0x20,al
35             RESTORE_REGS
36             iret
37     %endmacro
```

Essas macros serão usadas para determinar o segmento de interrupção que irá prevenir a corrupção dos diferentes registros, será muito útil para o multitasking.

Teoria: memória física e virtual

No capítulo relacionado ao GDT, vimos que o uso de segmentação de um endereço de memória física é calculado usando um seletor de segmento e um deslocamento.

Neste capítulo, vamos implementar a paginação que irá traduzir um endereço linear de segmentação em um endereço físico.

Por que precisamos de paginação?

A paginação irá possibilitar que nosso kernel:

- use o HD como uma memória e não ser limitada pelo limite de memória RAM da máquina
- tenha um espaço de memória único para cada processo
- permita e não permita espaço de memória de uma forma dinâmica

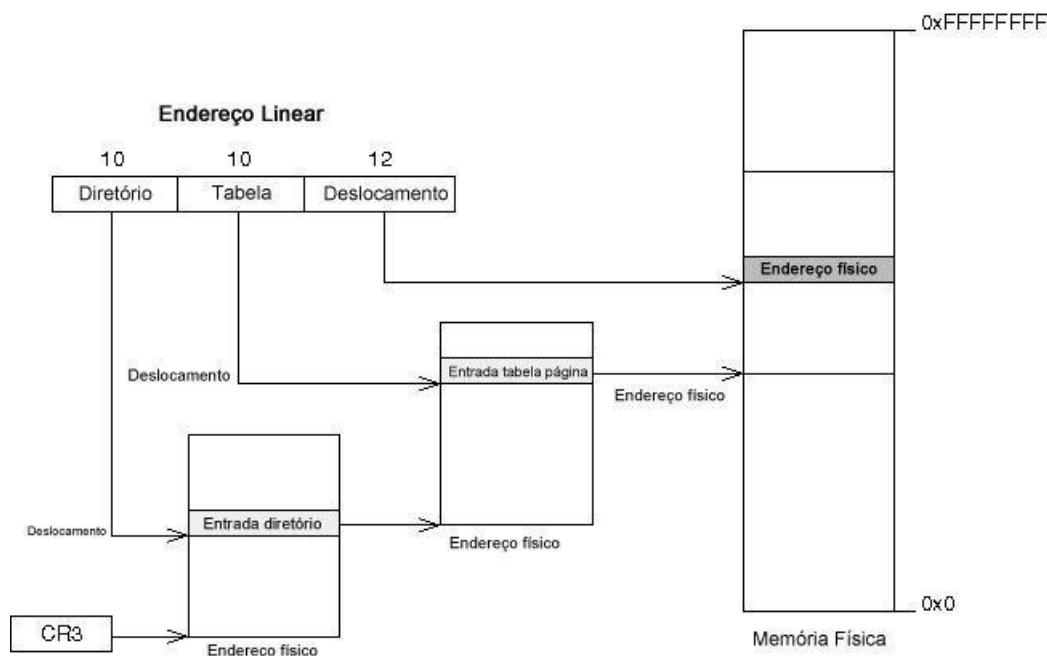
Em um sistema paginado, cada processo pode executar em sua própria área de 4GB de memória, sem qualquer chance de afetar a memória ou kernel de um outro processo. Ele simplifica o multitasking.

Memória física	Processo A		Processo B	
	Páginas da tabela	Memória virtual	Páginas da tabela	Memória virtual
00x H E L L	00x 00	00x H E L L	00x 03	00x H A V E
01x R L D !	01x 02	01x O W O	01x 05	01x L O T
02x O W O	02x 01	02x R L D !	02x 06	02x S O F
03x H A V E	03x n. a.	03x #####	03x 04	03x F U N
04x F U N	04x n. a.	04x #####	04x n. a.	04x #####
05x L O T	05x 07	05x ; -)	05x 07	05x ; -)
06x S O F				
07x ; -)				

Como isso funciona?

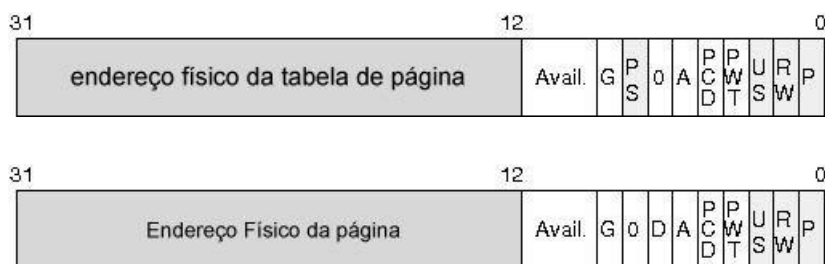
A tradução de um endereço linear em um endereço físico é feita em várias etapas:

1. O processador usa o registro *CR3* para saber o endereço físico do diretório de páginas.
2. Os primeiros 10 bits do endereço linear representam um deslocamento (entre 0 e 1023), indicando uma entrada no diretório de páginas. Essa entrada contém o endereço físico de uma tabela de páginas.
3. Os próximos 10 bits do endereço linear representam outro deslocamento, indicando uma entrada na tabela de páginas. Essa entrada está indicando uma página 4ko.
4. Os últimos 12 bits do endereço linear representam um deslocamento (entre 0 e 4095), que indicam a posição na página 4ko.



Formato para diretório e tabela de páginas

Os dois tipos de entradas (diretório e tabela) parecem o mesmo. Apenas os campos em cinza serão utilizados no nosso sistema operacional.



- *P*: indica se a página ou tabela está na memória física
- *R/W*: indica se a página ou tabela é acessível por escrita (igual à 1)
- *U/S*: é igual à 1 para permitir o acesso à tarefas não preferidas
- *A*: indica se a página ou tabela foi acessada
- *D*: (somente tabela de páginas) indica se a página foi escrita
- *PS*: (somente diretório de páginas) indica o tamanho das páginas:
 - 0 = 4kb
 - 1 = 4mb

Nota: endereços físicos nos diretórios ou tabelas de páginas são escritos usando 20 bits porque esses endereços estão alinhados em 4kb, então os últimos 12 bits devem ser iguais a 0.

- Um diretório ou tabela de páginas usam $1024 * 4 = 4096$ bytes = 4k
- Uma tabela de páginas podem endereçar $1024 * 4k = 4$ Mb
- Um diretporio de páginas pode endereçar $1024 (1024 \text{ 4k}) = 4$ Gb

Como habilitar a paginação?

Para habilitar a paginação, só precisamos definir o bit 31 do registro *CR0* para 1:

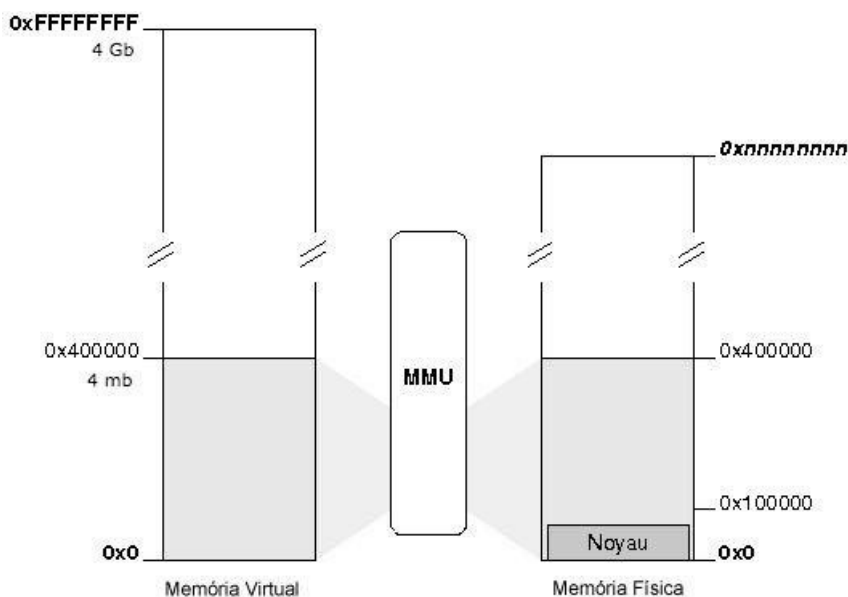
```
asm(" mov %%cr0, %%eax; \  
or %1, %%eax; \  
mov %%eax, %%cr0" \  
:: "r"(0x80000000));
```

```
1      asm(" mov %%cr0, %%eax; \  
2      or %1, %%eax; \  
3      mov %%eax, %%cr0" \  
4      :: "r"(0x80000000));
```

Mas antes, nós precisamos inicializar nosso diretório de páginas com pelo menos uma tabela de páginas.

Mapeamento de identidade

Com o modelo de mapeamento de identidade, a página será aplicada para o kernel apenas quando os primeiros 4 MB de memória virtual coincidir com os primeiros 4 MB de memória física:



Este modelo é simples: a primeira página de memória virtual coincide com a primeira página na memória física, a segunda página coincide com a segunda página na memória física e assim vai ...

Gerenciamento de memória: física e virtual

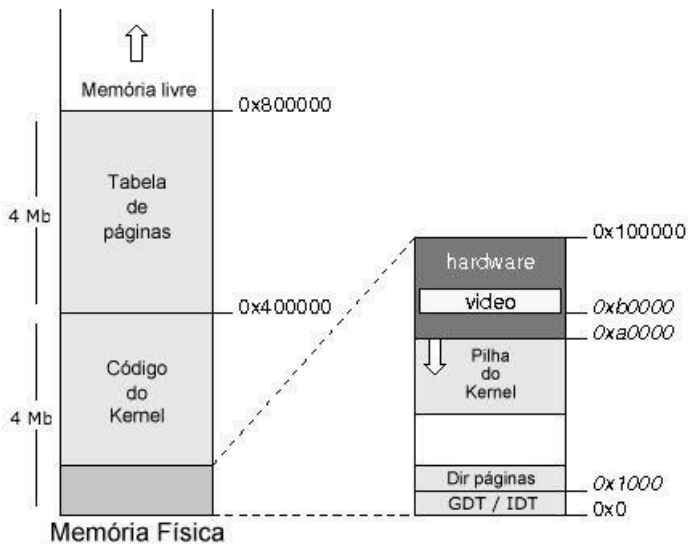
O kernel conhece o tamanho da memória física disponível graças ao **GRUB**.

Na nossa implementação, os primeiros 8 megabytes da memória física será reservada para uso pelo

kernel e irá conter:

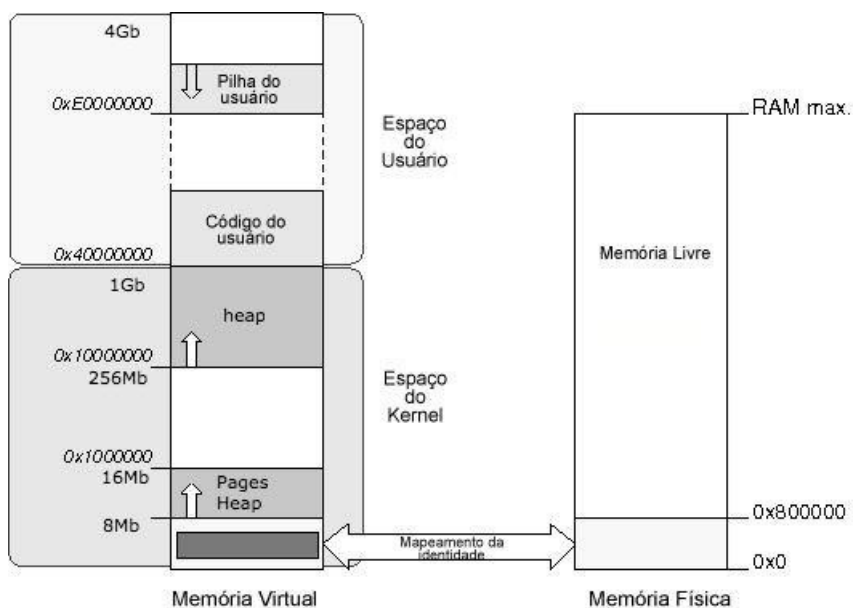
- O kernel
- GDT, IDT et TSS
- Kernel Stack (Pilha do kernel)
- Algum espaço reservado para o hardware (memória de vídeo, ...)
- Diretório e tabela de páginas para o kernel

O resto da memória física é livremente disponibilizada para o kernel e aplicações.



Mapeamento de Memória Virtual

O espaço de endereço entre o início da memória e o endereço 0x40000000 é o espaço do kernel, enquanto o espaço entre o endereço 0x40000000 e o fim da memória corresponde ao espaço do usuário:



O espaço do kernel na memória virtual no qual está usando 1 Gb da memória virtual é comum para todas as tarefas (kernel e usuário).

Isso é implementado apontando para as primeiras 256 entradas do diretório de página de tarefa para o diretório de página do kernel (Em [vmm.cc](#)):

```
/*
 * Kernel Space. v_addr &lt; USER_OFFSET are addressed by the kernel pages table
 */

1      /*
2      * Kernel Space. v_addr &lt; USER_OFFSET are addressed by the kernel pages table
3      */
4      for (i=0; i &lt; 256; i++)
5          pdir[i] = pd0[i];
```

Linux

◀ ARQUITETURA X86 ▶ ASSEMBLY ▶ BACKBONE ▶ C ▶ COMPILAR ▶ GDT ▶ GRUB ▶ IDT ▶ KERNEL

◀ LINUX ▶ PAGINAÇÃO ▶ RUNTIME ▶ SISTEMA OPERACIONAL ▶ VAGRANT