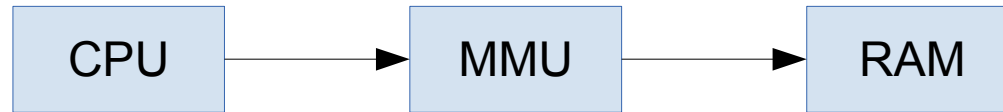


Operativni sistemi

dr.sc. Amer Hasanović

OS hardverski zahtjevi

- Za implementaciju svojih funkcija OS zahtijeva slijedeću hardversku podršku:
 1. Modovi operacija sa različitim privilegijama:
 - Procesor treba minimalno da podržava dva moda:
 - Privilegovani (**kernel**) mod u kojem su dostupne sve instrukcije date arhitekture.
 - korisnički (**user**) mod u kojem je dostupan samo podskup instrukcija.
 - CPU dok izvršava kernel kod treba da bude u privilegovanom modu, a dok izvršava kod od aplikacija treba da radi u korisničkom modu
 - Kernel → User mod tranzicija
 - Privilegovana instrukcija
 - User → kernel tranzicija
 - prekidi (interrupts), Hardverske iznimke (exceptions) i/ili sistemski pozivi (system calls)



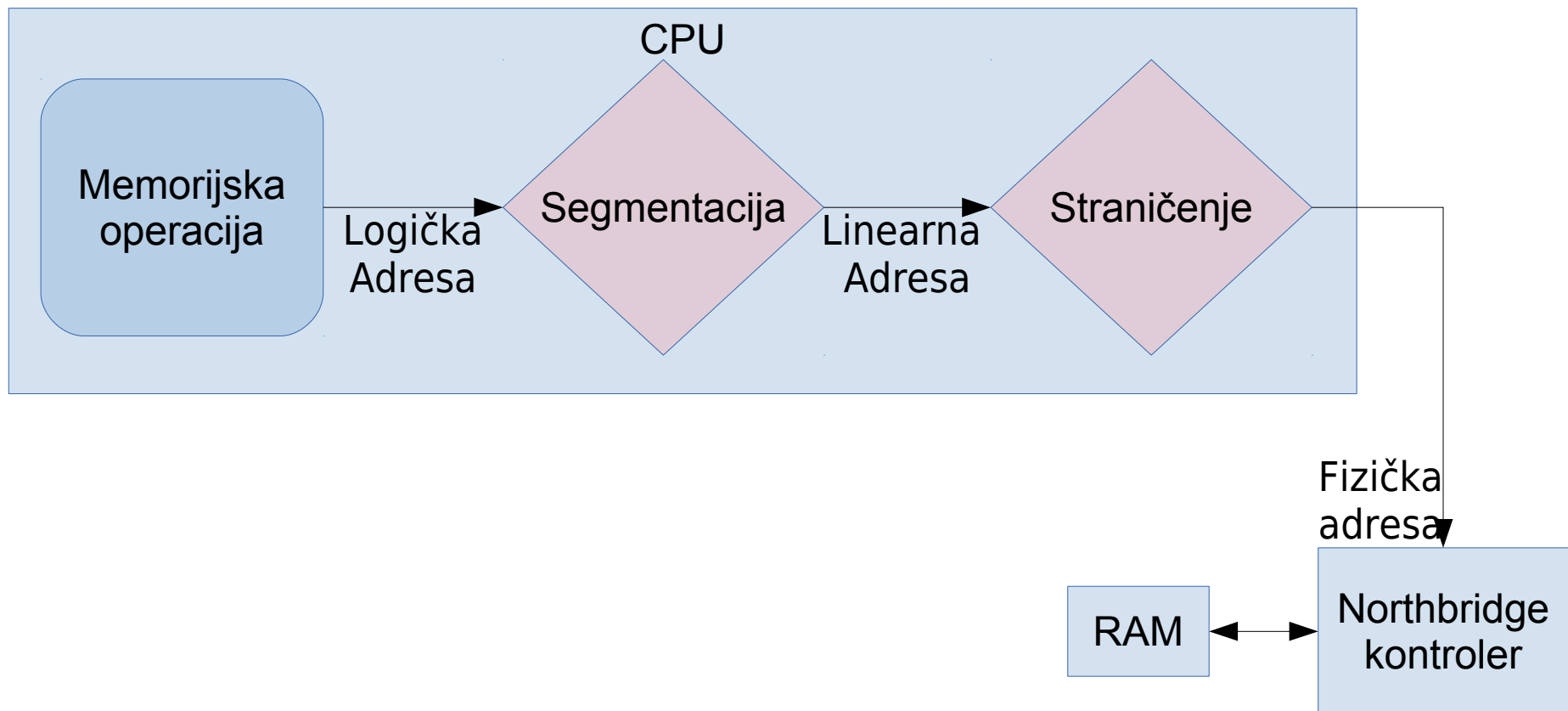
2. Memory Management Unit (MMU):

- Uređaj koji služi za translaciju (mapiranje) adresa iz virtuelnih memorijskih prostora različitih procesa u jedan fizički adresni prostor.
 - Kernel kontroliše:
 - Konfiguraciju virtuelnih adresnih prostora;
 - Konfiguraciju MMU radi odabira aktuelnog virtuelnog adresnog prostora u kojem procesor trenutno radi.
- Određene adrese virtuelnih prostora moguće je označiti nedostupnim, i/ili dostupnim samo dok je procesor u određenom privilegovanom modu operacija (protekcija)

X86 i memorija

- i386 podžava dva potpuno različita moda komunikacije sa memorijom:
 - 20 – bitni, **real** mode
 - koristi segmentiranje;
 - ne pruža nikakvu protekciju ili izolaciju;
 - zadržan radi kompatibilnosti sa 8086 procesorom;
 - po pokretanju svaki x86 procesor je u ovom modu.
 - 32 – bitni, **protected** mode
 - Koristi segmentiranje i straničenje (paging);
 - Straničenje može biti isključeno;
- Moguće je izvršiti promjenu moda: real → protected bez resetovanja CPU-a.

X86 i memorija



X86 segmentacija

- Obavlja se putem segmentnih registara:
 - šest 16 – bitnih registara slijedećih oznaka:
 - %cs → code segment
 - %ss → stack segment
 - %ds → data segment
 - %es, %fs, %gs → dodatni data segmenti

Odabir segmenta

- **Svaka** referenca memorije uključuje segmentni registar putem operatora : , npr:
 - `movw %ax, %ss:12(%esp)`
 - `mov %ds:0xa0, %ebh`
- Ukoliko se pri memorijskoj referenci izostavi segment registar, assembler implicitno odabira isti na osnovu pravila:
 - instrukcije za kontrolu toka koriste %cs (`jmp`, `call` itd)
 - stack instrukcije (`push`, `pop` i `mov` sa baznim registrom %esp i/ili %ebp) koriste %ss
 - Većina load/store instrukcija koriste %ds
- Slijedeća instrukcija koja se izvršava preuzima se sa memorijske lokacije određene %cs registrom tj:
 - `%cs:%eip`

Real mode segmentiranje

- U ovom modu logička adresa iz instrukcije konvertuje se u fizičku na slijedeći način:
 - $\text{seg}:\text{offset} \rightarrow (\text{seg} \ll 4) + \text{offset}$
 - offset je 16b vrijednost
- primjer:

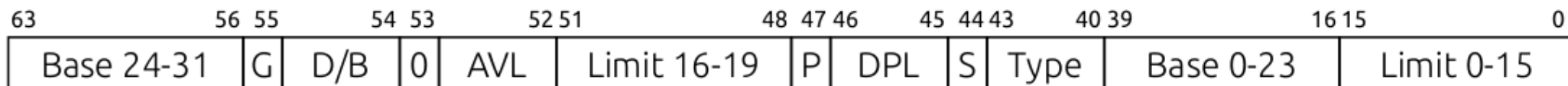
```
movw 0xaf2c, %ax  
movw %ax, %ds  
movb %bh, 0xb
```



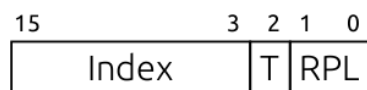
$(0xaf2c \ll 4) + 0xb \rightarrow 0xaf2cb$

Protected mode adresiranje

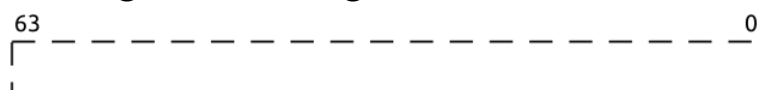
Deskriptor



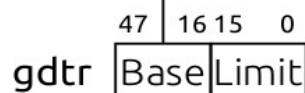
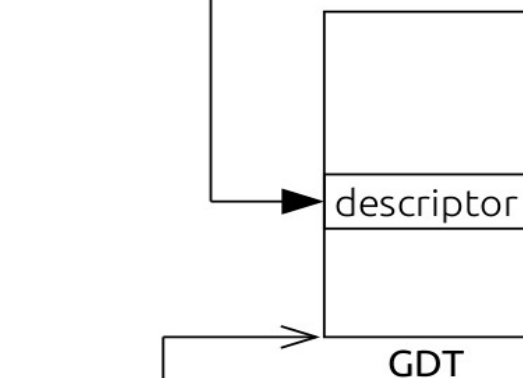
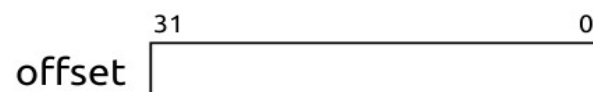
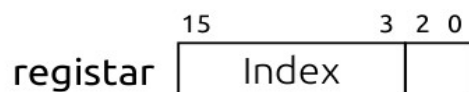
Selektor – Segmentni registri



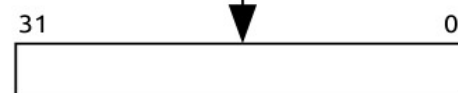
Registar



Keširani deskriptor

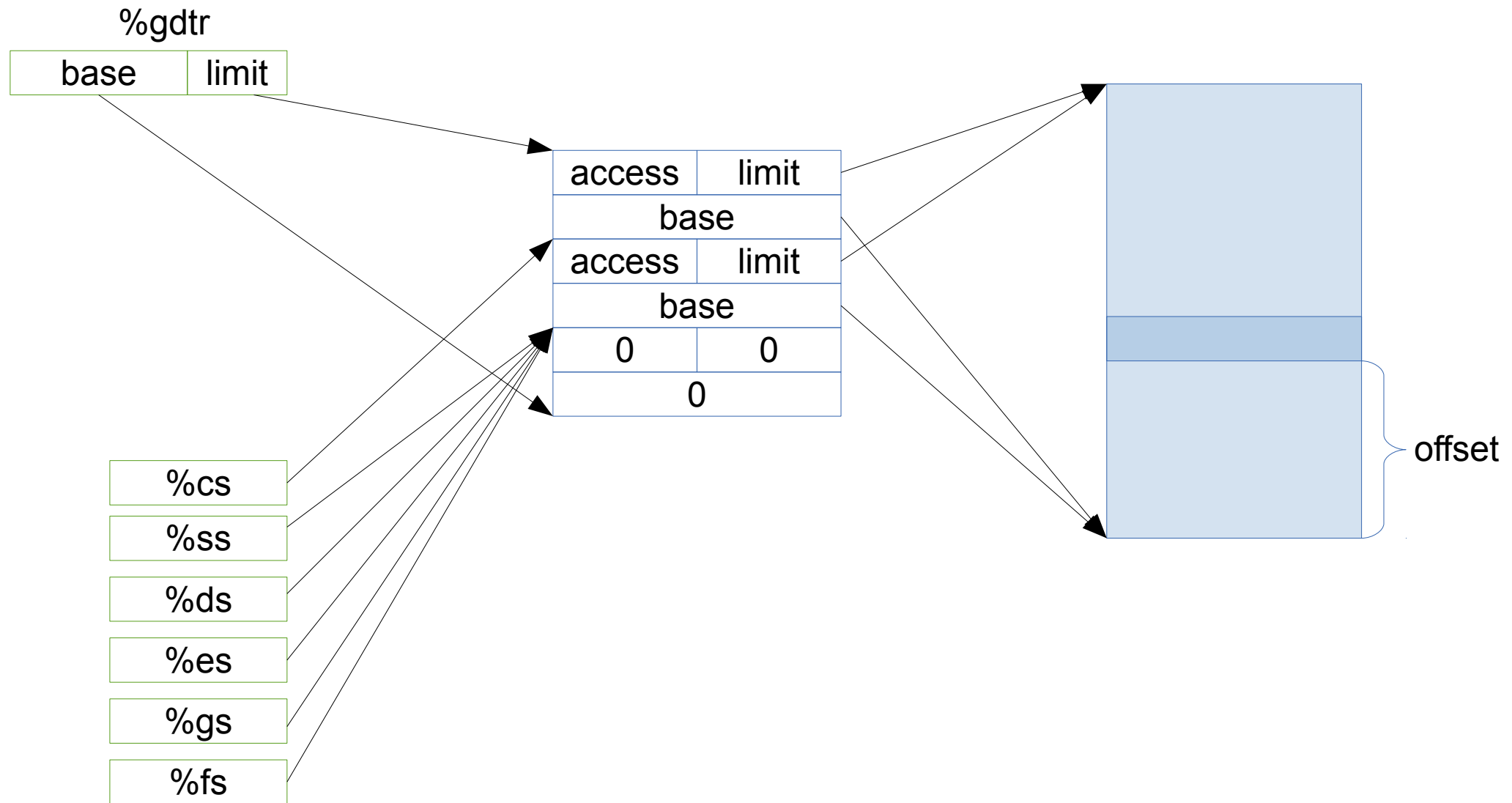


base



Linearna adresa

Flat model segmentacije



Memorija pri PC pokretanju

- Prilikom pokretanja procesor starta u *real* modu sa slijedećim izgledom memorije:

+-----+ BIOS ROM +-----+	<- 0x00100000 (1MB)
+-----+ 16-bit devices, expansion ROMs +-----+	<- 0x000F0000 (960KB)
+-----+ VGA Display +-----+	<- 0x000C0000 (768KB)
+-----+ Low Memory +-----+	<- 0x000A0000 (640KB)
+-----+	<- 0x00000000

- Registri cs i eip su inicijalizirani sa vrijednostima:
 - %cs → 0xf000 i %eip → 0xffff0
 - što prouzrokuje izvršavanje koda iz BIOS segmenta.
- BIOS je pohranjen u flash memoriji matične ploče

Bootloader

- BIOS učitava prvi sektor (512B) sa diska, odabranog u BIOS konfiguraciji (tzv boot disk), na lokaciju 0x7C00 i izvršava:
 - jmp 0x7c00
- Prvi sektor diska treba da sadrži program (bootloader) koji ima zadatak da:
 - izvrši transfer u protected mod;
 - učitá kernel (code, data) sa diska, i to na odgovarajuću adresu;
 - formira kernel stack;
 - preda kontrolu kernelu.
- Kernel je na disku u nekom objektnom formatu (npr ELF)

Makefile

```
bootblock: bootasm.S bootmain.c
    $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
    $(OBJDUMP) -S bootblock.o > bootblock.asm
    $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
```

```
xv6.img: bootblock kernel fs.img
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc
```

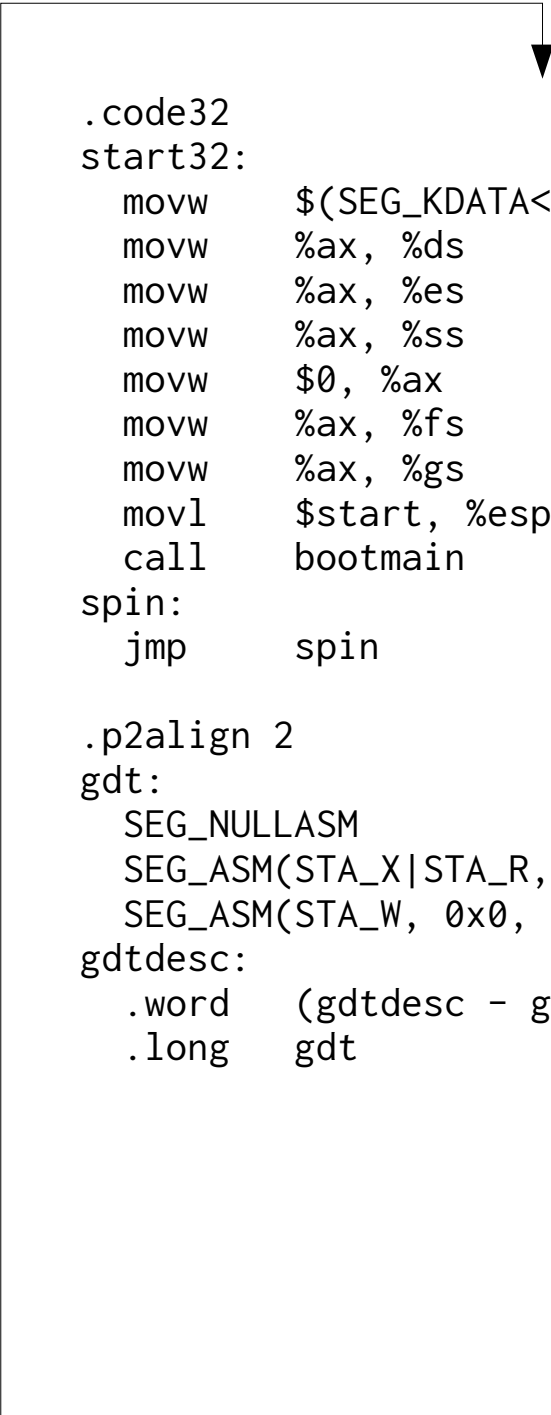
Transfer u protected mode

- Aktivirati A20 adresnu liniju, komuniciranjem sa kontrolerom tastature (sekvenca in i out instrukcija)
- gdt r registar (48b), pomoću instrukcije lgdt, treba inicijalizirati tako da pokazuje na globalnu tabelu sa deskriptorima segmenata
- izvršiti real → protected mode transfer
 - promjenom nultog bita kontrolnog registra %cr0 (32b) iz 0 u 1.
- Pomoću ljmp instrukcije izvršiti skok na prvu instrukciju u 32b modu
 - Ovim je učitana i novi sadržaj %cs registra
- Adekvatno podesiti i ostale segmentne registre

bootasm.S

```
#include "asm.h"
#include "memlayout.h"
#include "mmu.h"
.code16
.globl start
start:
    cli
    xorw    %ax,%ax
    movw    %ax,%ds
    movw    %ax,%es
    movw    %ax,%ss
seta20.1:
    inb     $0x64,%al
    testb   $0x2,%al
    jnz     seta20.1
    movb    $0xd1,%al
    outb    %al,$0x64
seta20.2:
    inb     $0x64,%al
    testb   $0x2,%al
    jnz     seta20.2
    movb    $0xdf,%al
    outb    %al,$0x60

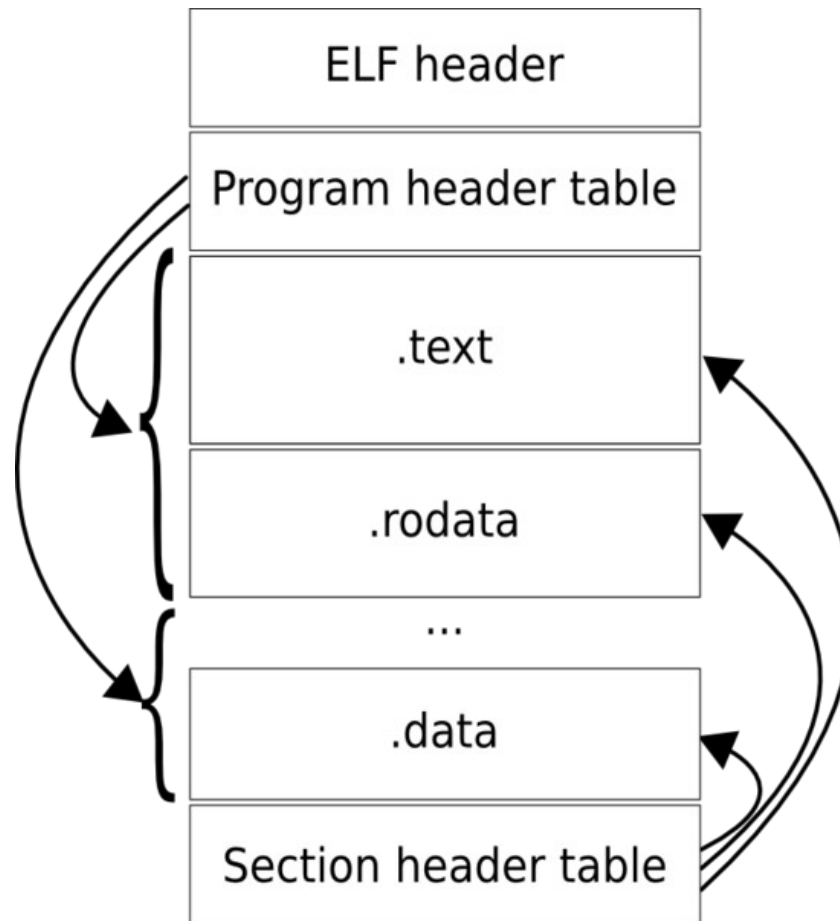
    lgdt    gdtdesc
    movl    %cr0, %eax
    orl     $CR0_PE, %eax
    movl    %eax, %cr0
    ljmp    $(SEG_KCODE<<3), $start32
```



```
.code32
start32:
    movw    $(SEG_KDATA<<3), %ax
    movw    %ax, %ds
    movw    %ax, %es
    movw    %ax, %ss
    movw    $0, %ax
    movw    %ax, %fs
    movw    %ax, %gs
    movl    $start, %esp
    call    bootmain
spin:
    jmp     spin

.p2align 2
gdt:
    SEG_NULLASM
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)
    SEG_ASM(STA_W, 0x0, 0xffffffff)
gdtdesc:
    .word   (gdtdesc - gdt - 1)
    .long   gdt
```

ELF objektni format



bootmain.c

```
#include "types.h"
#include "elf.h"
#include "x86.h"
#include "memlayout.h"
#define SECTSIZE 512

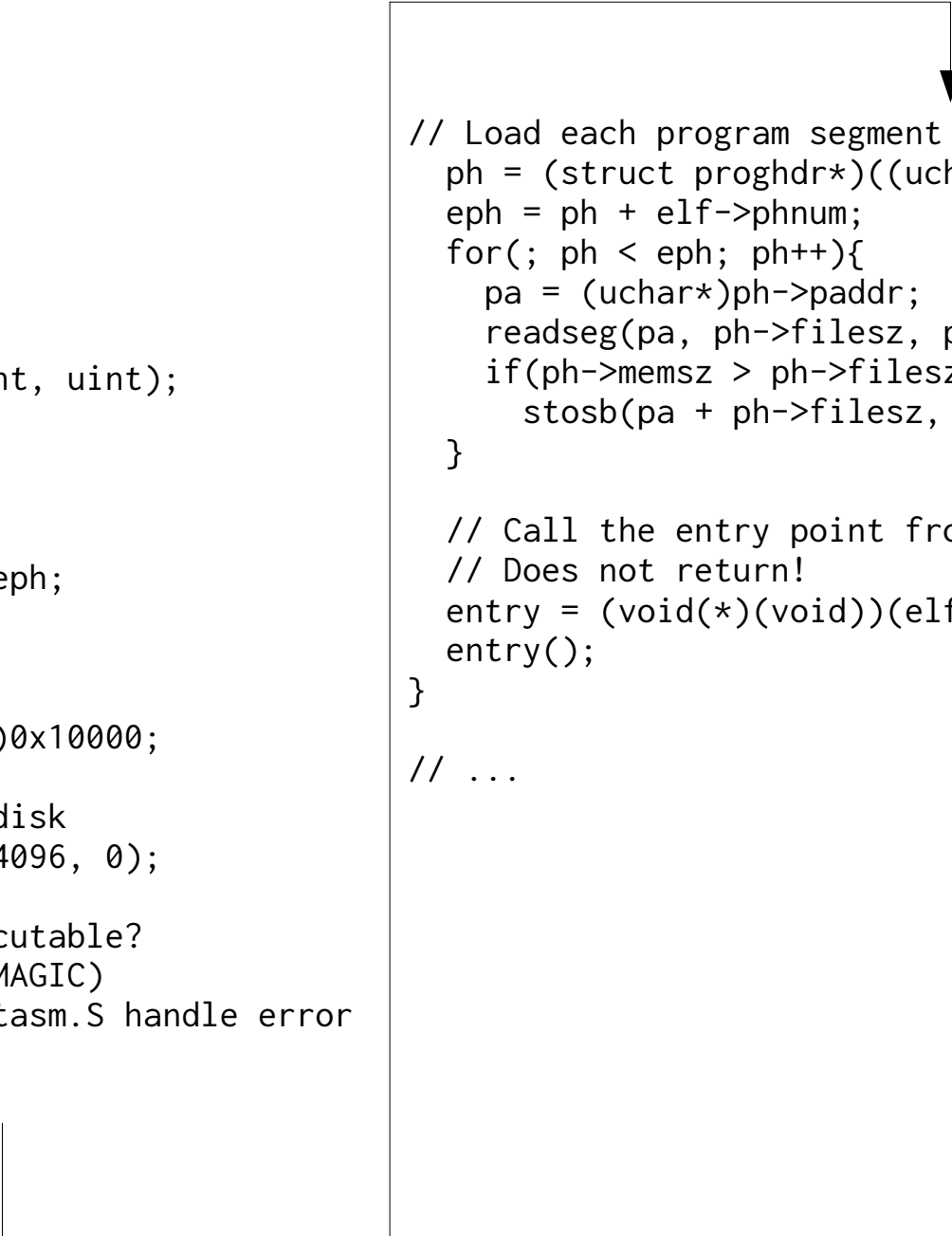
void readseg(uchar*, uint, uint);

void bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000;

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);

    // Is this an ELF executable?
    if(elf->magic != ELF_MAGIC)
        return; // let bootasm.S handle error
```



```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
        stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
}

// Call the entry point from the ELF header.
// Does not return!
entry = (void(*)(void))(elf->entry);
entry();
}

// ...
```