Inicializace v chráněném (32bitovém) módu

29. září 2021

Minule jsme si demonstrovali inicializaci systému v dnes již silně archaickém reálném 16bitovém režimu. Tento režim má řadu omezení. Na první pohled se jedná o omezenou 16bitovou instrukční sadu s možností adresovat jen 1 MB paměti. Vedle toho v reálném režimu nejsou žádné prostředky, které by umožňovaly implementovat ochranu paměti a oddělit kód jádra operačního systému od uživatelských procesů, nemluvě o implementaci dalších funkcí operačních systémů jako je virtuální pamět.

1 Chráněný režim procesoru

Historicky si správa paměti na platformě x86 prošlo postupným vývojem. V reálném režimu (procesory 8086) byla použita segmentace jako nástroj, který umožňuje na 16bitové architektuře využívat 20bitové adresy a zpřístupnit tak 1MB operační paměti.

S příchodem procesorů 80286 je k dispozici nový režim procesoru, tzv. chráněný mód (protected mode), kdy segmentace začíná plnit další úlohu, ochranu paměti.[1]

Z pohledu instrukční sady nedochází k žádným výrazným změnám. Jednotlivé instrukce přistupují k paměti obdobným způsobem jako v reálném režimu (segment[2] + adresa[3]), avšak segmenty a práce s nimi dostala jinou formu. Zatímco v reálném režimu procesoru hodnota v segmentovém registru určuje přímo adresu daného segmentu (úseku paměti), v chráněném režimu hodnota v segmentovém registru určuje index do tabulky deskriptorů, kde jednotlivé záznamy (deskriptory) popisují vlastnosti jednotlivých segmentů. Díky tomu je možné zvětšit rozsah adresovatelné paměti[4] a ke každému segmentu přiřadit příznaky jako je úroveň oprávnění, oprávnění číst/zapisovat do segmentu, případně indikovat, zda je segment v paměti, či k němu bylo přistupováno. S tímto již lze implementovat „moderní" operační systémy jako jsou unix nebo Windows. I když takové operační systémy vznikly, jejich ohlas byl omezený, protože z povahy 16bitové instrukční sady bylo možné použít segmenty o maximální velikosti 64 kB.

Výraznou změnu proto představuje příchod procesorů 80386, které přináší zpětně kompatibilní 32bitovou instrukční sadu, systém správy paměti založený na segmentaci, který zachovává a rozšiřuje vlastnosti uvedené procesory 80286 tak, aby bylo možné adresovat celých 4 GB RAM. Dodejme, že procesory 80386 přináší i podporu stránkování, tato funkce je volitelná a lze ji vypnout, resp. nezapnout.

Pro naše potřeby budeme potřebovat přepnout procesor do chráněného 32bitového režimu.

[1] Tyto procesory mohou stále používat i původní reálný mód.
[2] určený segmentovým registrem
[3] offset v rámci segmentu
[4] v případě 80286 na 16MB

1.1 Přepnutí procesoru do 32bitového režimu

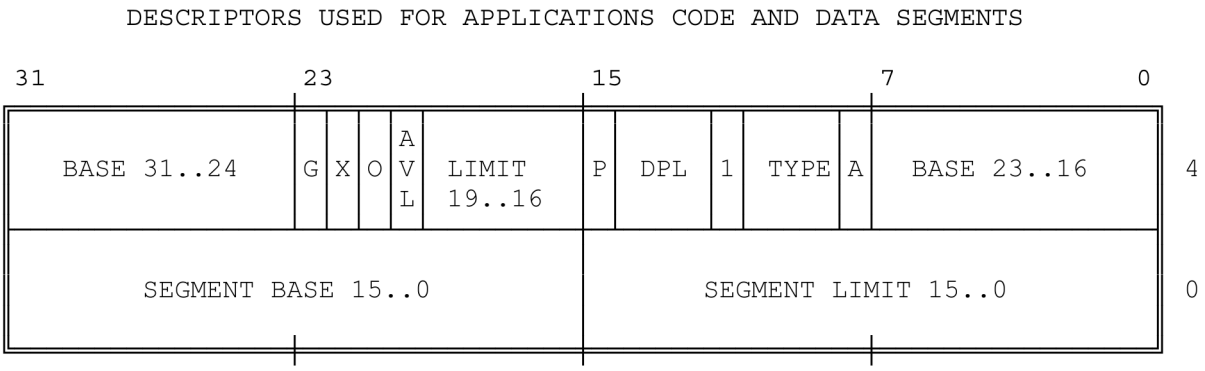
Přepnutí procesoru do 32bitového režimu je realizováno ve třech krocích.

1. vytvoření tabulky deskriptorů,
2. načtení tabulky deskriptorů do řídících registrů,
3. zapnutí chráněného režimu a jeho inicializace.

1.1.1 Nastavení tabulky deskriptorů

Architektura x86 významným způsobem rozlišuje mezi kódovým segmentem (segmentem obsahujícím prováděný kód) a datovými segmenty, kde jsou uložena data, zásobník atp. Proto pro inicializaci 32bitového režimu budeme muset nastavit minimálně dva segmenty, jeden kódový a jeden datový.

Tyto segmenty jsou popsány v jedné ze dvou tabulek GDT (global descriptor table), která je globální pro všechny procesy, a LDT (local descriptor table), která je lokální pro konkrétní proces. Pro naše potřeby bude stačit nastavit tabulku GDT. Záznamy v této tabulce mají 8 B a jejich strukturu ukazuje Obrázek 1.[5].



Obrázek 1: Struktura deskriptoru pro kódový nebo datový segment

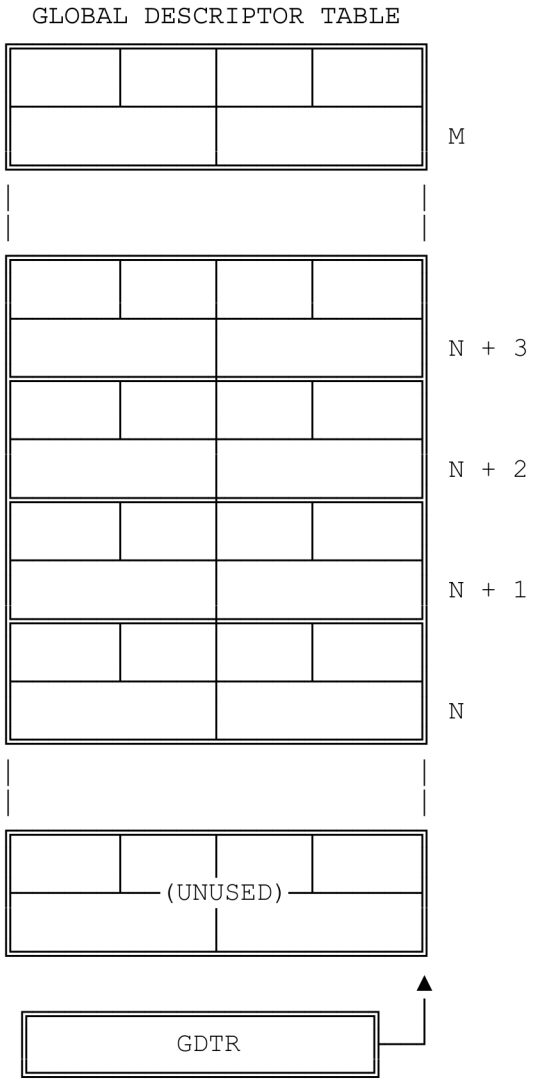
Význam jednotlivých bitů je následující:

- segment limit – velikost segmentu,
- segment base – začátek segmentu,
- A – accessed,
- TYPE
  - DC
    - u datového segmentu určuje, jestli segment roste od menších adres k vyšším (0) nebo opačně (1)
    - u kódového segmentu určuje konformitu, tj. jestli může být kód prováděn s nižším oprávněním než má segment,

[5]Obrázky datových struktur převzaty z: INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986

  - RW – read/write (povolené čtení kódového segmentu, zápis do datového)
  - EX – kódový (1), datový (0) segment
- 1 – značí běžný kódový nebo datový segment (0 pro systémové segmenty)
- DPL – descriptor privilege level (2 bity),
- P – present,
- AVL – available for system programmers,
- 0,
- X – typ descriptoru (16/32 bitů)
- G – granularity (byty/stránky)

Tyto deskriptory jsou uloženy v tabulce GDT, která může pojmout až 8192 takových záznamů, jak ilustruje Obrázek 2, přičemž první záznam v GDT musí být null-deskriptor.



Obrázek 2: Struktura tabulky GDT

Kde je uložena tabulka GDT, určuje řídící registr GDTR. Avšak pozor, protože tabulky GDT mohou mít různé velikosti, nestačí specifikovat jen, kde se tabulka GDT nachází, ale i její velikost. K tomu slouží deskriptor tabulky GDT[6], který se skládá ze dvou částí (i) 16 bitů obsahujících velikost tabulky mínus 1 a (ii) 32 bitové adresy tabulky.

Na úrovni assembleru můžeme tabulku GDT i její deskriptor pohodlně zapsat formou pseudoinstrukcí db, dw, dd, jak ukazuje následující kód.

```
;
; GDT
;
gdt:
gdt_null:             ; povinny null-zaznam v GDT
    dd 0x0
    dd 0x0

gdt_cs:               ; descriptor kodoveho segmentu
    dw 0xffff         ; limit (bity 0-15)
    dw 0x0000         ; base (bity 0-15)
    db 0x00           ; base (bity 16-23)
    db 10011010b      ; 1: present; 00: ring0; 1: not-system-seg.; 1: code-seg.;
                      ; 0: not-conforming; 1: read/write; 0: accessed
    db 11001111b      ; 1: granularity = pages; 1: 32bit mode; 00: reserved;
                      ; 1111: limit (bity 16-19)
    db 0x00           ; base (bity 24-31)

gdt_ds:               ; descriptor datoveho segmentu
    dw 0xffff         ; limit (bity 0-15)
    dw 0x0000         ; base (bity 0-15)
    db 0x00           ; base (bity 16-23)
    db 10010010b      ; 1: present; 00: ring0; 1: not-system-seg.; 0: data-seg.;
                      ; 0: grows up; 1: read/write; 0: accessed
    db 11001111b      ; 1: granularity = pages; 1: 32bit mode; 00: reserved;
                      ; 1111: limit (bity 16-19)
    db 0x00           ; base (bity 24-31)
gdt_end:

;
; descriptor GDT
;
gdt_desc:             ; descriptor tabulky GDT
    dw gdt_end - gdt - 1      ; velikost tabulky - 1
    dd gdt            ; adresa GDT
```

[6] nezaměňovat s deskriptorem segmentu!

1.1.2 Nastavení řídících registrů

Pokud máme k dispozici platnou tabulku GDT, je potřeba provést nastavení tak, aby se procesor daty z této tabulky začal řídit. První krok, který bychom měli podle dokumentace procesoru provést, je zablokování přerušení pomocí instrukce `cli`, tím se předejde nekonzistentní stavům, protože 32bitový režim pracuje s přerušeními jinak než reálný mód.

V dalším kroce pomocí instrukce `lgdt` uložíme do GDTR adresu tabulky GDT, která je dána jejím deskriptorem.

Samotný chráněný mód procesoru se zapne nastavením nejnižšího bitu v řídícím registru CR0.

Tento postup představuje následující kód.

```
cli                     ; zablokujeme preruseni
lgdt [gdt_desc]         ; nacteme do GDTR deskriptor GDT

mov eax, cr0            ; v ridicim registru CR0 nastavime
or eax, 1               ; bit indikujici chraneny mod
mov cr0, eax
```

1.1.3 Inicializace 32bitového režimu

Pokud vše proběhlo v pořádku, v tento okamžik bychom měli mít procesor ve 32bitovém chráněném režimu. Avšak došlo nám k významné změně práce se segmentovými registry, a je tedy nutné je přenastavit. První, co musíme udělat, je nastavit kódový segment a to tak, že provedeme skok na adresu, která je dána selektorem kódového segmentu a vstupním bodem 32bitového kódu.

```
jmp 0x08:main32         ; skok do 32bitoveho segmentu (selektor 0x08)
```

Následně musíme nastavit hodnoty ostatních segmentových registrů, tj. ds, ss, es, fs, gs. Jelikož máme jeden datový segment, nastavení je přímočaré.

```
mov ax, 0x10            ; selektor dataveho segmentu
mov ds, ax
mov ss, ax
; etc.
```

Všimněme si, že v přiloženém zdrojovém kódu kombinujeme 16bitový i 32bitový kód. Jednotlivé části je potřeba oddělit direktivami `[BITS16]` a `[BITS32]` a důsledně si hlídat, abychom nechtěně nepřecházeli mezi jednotlivými módy, např. při volání podprogramů.

2 Práce v 32bitovém režimu

Tím, že jsme se přepnuli z 16bitového režimu do režimu 32bitového, jsme sice získali lepší instrukční sadu a více funkcí, které procesor nabízí. To je však vykoupeno tím, že jsme přišli o funkce, které poskytuje BIOS, jako jsou přístup k terminálu nebo disku. Přístup k periferiím si tedy musíme řešit ve vlastní režii.

Pokud chceme ověřit, že jsme opravdu ve 32bitovém režimu a něco vypsat na terminál, musíme to řešit přímým přístupem do paměti, která odpovídá vypisovanému obsahu. Tato paměť se nachází na adrese `0x000b:8000` a jednotlivé znaky jsou tam uloženy ve formě 8 bitů znak + 4 bity barva pozadí + 4 bity barva popředí.

Podobně, pokud chceme načíst kód, který přesahuje MBR, musíme to udělat ve vlastní režii (ukážeme si později) nebo musíme data načíst ještě v 16bitovém režimu pomocí služeb BIOSu, což jsme si ukázali minule.

Další problém, kterému musíme čelit, je skutečnost, že veškerý kód, který jsme dosud vytvořili, je napsaný v assembleru, což není úplně komfortní. Proto se běžně v assembleru píší jen kritické části kódu, které nemohou být psány ve vyšším programovacím jazyce, a zbytek se píše v nějakém vysokoúrovňovém jazyce typu C, C++, případně dnes Rust.[7]

S těmito problémy se vypořádáme následovně.

1. Vytvoříme kód v jazyce C, který bude načten do paměti a spuštěn (může sloužit jako základ jádra OS).
2. Načteme kód s pomocí BIOSu.
3. Přepneme procesor do 32bitové chráněného režimu a necháme jej provádět.

2.1 Kód v jazyce C

Potřebujeme vytvořit kód, který bude možné zavolat po přepnutní do 32bitového režimu. Vytvoříme si proto zdrojový kód, který bude obsahovat právě jednu funkci bez jakýchkoliv parametrů.

```
void kernel_main()
{
  unsigned char *vram = (unsigned char *) 0xb8000;
  for (int i = 0; i < 5; i++) {
    vram[i * 2] = hello[i];
   }
}
```

Kompilace tohoto kódu pro naše potřeby je složitější, než jsme zvyklí, protože:

(i) potřebujeme vygenerovat čistý strojový kód, nikoliv soubor ve formátu ELF,

(ii) potřebujeme specifikovat, jak má být kód v paměti rozložen,

(iii) nemůžeme využít standardní knihovny.

S problémem (iii) je možné se vypořádat s pomocí přepínačů `-ffreestanding` a `-nostdlib`. Problémy (i) a (ii) řeší podrobnější konfigurace linkovací fáze pomocí linker skriptu, viz přiložený soubor `kernel.ld` a `Makefile`.

[7] I když existují i pokročilé operační systémy kompletně napsané v assembleru, např. MenuetOS.

2.2 Načtení kódu do paměti

Minule jsem si ukázali, jak za MBR připojit další sektor a ten načíst do paměti. Ve své podstatě použijeme to samé jen s drobnými rozdíly. Musíme načítat více sektorů, protože náš kód již bude mít víc než 512 B. Toho lze dosáhnout drobnou úpravou minulého kódu, je však nutné počítat s tím, že služba BIOSu pro čtení dat z disku umí načíst maximálně 127 bloků.

Situaci dále komplikuje, že kód zavádějící budoucí jádro operačního systému musí vědět, jak velký kód se bude načítat. Tady si vypomůžeme drobným hackem. Protože de facto sestavujeme bootovací disk, včetně MBR, můžeme velikost jádra předat jako konstantu v průběhu překladu MBR.

Viz přiložené kódy `pm-boot.asm` a `Makefile`.

3 Poznámky závěrem

1. Že je jádro uloženo na specifikovaném místě disku, je dnes již archaické řešení, které se např. používalo u prvních verzí Linuxu. Od tohoto řešení se upustilo, protože je značně nepružné.

2. Dnešní zavaděče typu GRUB jsou schopny přepnout procesor do 32bitového režimu a načíst jádro z regulérního souborového systému. My jsme tuto variantu nepoužili, protože jsme chtěli ukázat nastavení GDT a zatím nemáme kód, který by byl schopen pracovat se souborovým systémem.[8]

3. Značně jsme si ulehčili práci tím, že jsme operační systém nahráli do nižší paměti (do spodních 1 MB). Protože budeme mít minimalistické jádro nemělo by to vadit. Pokud bychom chtěli jádro zavést do vyšší paměti (nad 1 MB), museli bychom zapojit ještě jedno volání BIOSu, které umožňuje kopírovat data do paměti nad nad 1 MB.[9]

4 Úkoly

1. Implementujte funkci `void kprint(char *s)`, která vypíše zadaný řetězec na terminál. Pokud nebude stačit místo, obrazovka se „odscrolluje", aby se uvolnilo místo. Funkce by měla korektně pracovat s řídícími znaky `\r`, `\n`.

2. Implementujte funkci `void kprint_i(unsigned int i)`, která vypíše zadané číslo `i`, podobně jako `kprint`.

3. (Doplňkový úkol) Implementujte funkci `void kprint_color(int fg, int bg)`, která nastaví, jakou barvou se text bude vypisovat.

[8] Sestavení souborového systému navíc vyžaduje rootovská oprávnění, což komplikuje vývoj.
[9] Hezky to ukazuje Minimal Linux Bootloader, viz `https://github.com/wikkyk/mlb`.