Inicializace v reálném (16bitovém) módu

ZS 2023

Dnešní procesory Intel a počítače rodiny IBM PC z historických důvodů udržují velkou míru zpětné kompatibility se staršími modely, které sahají hluboko do sedmdesátých a osmdesátých let minulého století. V dnešních počítačích jsou tyto relikty již značně upozaděny, i tak se dnes dají najít místa, kdy se dá setkat s, obrazně řečeno, živoucími fosíliemi. Jedním z těchto míst je tzv. reálný mód, který zajišťuje kompatibilitu s 16bitovými procesory Intel 8086, které stály na počátku éry dnešních osobních počítačů. V dnešních počítačích je tento režim v zásadě přeskočen zavaděčem operačního systému, my si jej však představíme proto, abychom mohli pochopit chování počítače při inicializaci a mohli také porozumět některý vlastnostem dnešní ISA procesorů x86.

1 Reálný mód

Reálný mód procesoru emuluje vlastnosti procesorů Intel 8086 a 8088[1], tyto procesory byly charakteristické tím, že měly sadu 16bitových registrů ax, bx, cx, dx, které se dělily na registry ah, al, bh, bl, ..., dále registry si a di sloužící k přesunu dat, registry sp, bp pro práci se zásobníkem a řídící registry ip a f, kde registry ip ukazuje na následující instrukci, která se má provádět a registr f obsahuje příznaky související s prováděnými instrukcemi a stavem procesoru.

Procesory typu 8086 měly možnost adresovat až 1 MB paměti.[2] Avšak pomocí 16bitových registrů lze indexovat jen 65.536 B (tj. 64 kB), aby bylo možné tento nesoulad překonat, používá ISA 8086 pro adresaci paměti *segmentaci*. To v kontextu těchto procesorů znamená, že paměť lze rozdělit na bloky o velikosti 64 kB umístěné libovolně ve fyzické paměti, kde každý segment má svůj účel, jmenovitě je to *kódový segment* (obsahující prováděný kód), *zásobníkový segment* (obsahující zásobník), *datový segment* a *extra segment* (obsahující data, se kterými se pracuje).

Kde se jednotlivé segmenty nachází, určuje čtveřice 16bitových registrů cs (code segment), ss (stack segment), ds (data segment) a es (extra segment). A to tak, že při přístupu do paměti je adresa vypočtena jako *hodnota segmentového registru* $\times$ 16 + *adresa*. Například máme-li hodnotu ds rovnu `0x1a00` a hodnotu si rovnu `0x500`, načte následující instrukce `mov al, [ds:si]`, do registru al jednobytovou hodnotu z adresy `0x1a500`. Všimněme si, že vynásobení segmentového registru šestnácti odpovídá bitovému posunu vlevo o čtyři bity.

Při přístupu do paměti je možné určit segment, se kterým se bude pracovat pomocí segmentového re-

[1] Oba procesory mají stejnou 16bitovou instrukční sadu, ale procesory 8088 měly jen osmibitovou externí sběrnici, která umožňovala levnější výrobu počítačů, proto se s nimi dalo setkat častěji.

[2] U jednotek souvisejících s kapacitou paměti budeme používat jednotky vycházející za základu 2, tj. 1 kB = 2^{10} B, 1 MB = 2^{20} B atd.

gistru, jak jsme viděli v předchozím odstavci. Avšak mnohem častěji se s jednotlivými segmenty pracuje implicitně, tj. pokud se jedná o instrukci pracující s kódem (např. instrukce skoku), je použit implicitně kódový segment, pracuje-li se zasobníkem (tj. je použit registr sp nebo bp) je použit zásobníkový segment, v ostatních případech datový segment. Předchozí příklad by šel zjednoduši na `mov al, [si]`.

V reálném módu procesoru je možné téměř cokoliv.[3] Z praktického pohledu to znamená, že můžeme využívat bez omezení všech instrukcí, které tento mód nabízí, včetně těch, které přímo pracují s hardwarem. Je možné nastavovat segmentové registry, případně přistupovat do všech částí paměti.

2 Rozdělení paměti při iniciliazaci a inicializace

Je-li k inicializaci počítače použit BIOS (Basic Input Output System) je využit pouze 1 MB paměti, což je dáno omezením reálného módu procesoru a jednotlivé části této paměti mají svůj definovaný význam, např. na adresách `0x0000:0000` až `0x0000:03fff` jsou umístěny adresy obslužných rutin pro jednotlivá přerušení, na adrese `0x000a:0000` nalezneme grafickou paměť, atd. Viz Tabulka 1. Z rozdělení paměti plyne, že z 1 MB je pro operační systém a jednotlivé uživatelské programy a jejich data k dispozici cca 480 kB.[4]

začátek	**konec**	**velikost**	**obsah**
`0x0000:0000`	`0x0000:03FF`	1 kB	Real Mode (Interrupt Vector Table)
`0x0000:0400`	`0x0000:04FF`	256 B	BIOS data area
`0x0000:0500`	`0x0000:7BFF`	cca 30 kB	Conventional memory
`0x0000:7C00`	`0x0000:7DFF`	512 B	Boot Sector
`0x0000:7E00`	`0x0007:FFFF`	cca 480 kB	Conventional memory
`0x0008:0000`	`0x0009:FFFF`	128 kB	Extended BIOS Data Area
`0x000A:0000`	`0x000B:FFFF`	128 kB	Video display memory
`0x000C:0000`	`0x000C:7FFF`	32 kB	Video BIOS
`0x000C:8000`	`0x000E:FFFF`	160 kB	BIOS Expansions
`0x000F:0000`	`0x000F:FFFF`	64 kB	Motherboard BIOS

Tabulka 1: Přibližné rozložení paměti v reálném módu procesoru

Pro nás je zajímavá a klíčová adresa `0x0000:7c00`, kam je při inicializaci načten Master Boot Record (MBR) první sektor disku, jehož obsah je po načtení vykonán. MBR má velikost 512 B, přičemž prvních 440 až 446 B lze použít pro zavaděč operačního systému, dále následuje tabulka dělící disk na jednotlivé oddíly (partitions, tu nebudeme používat) a sektor je zakončen dvoubytovou signaturou `0xaa55`.

Ukažme si nejjednodušší možný příklad, jak by takový MBR mohl vypadat, pokud by byl psán v externím assembleru NASM.

```
1 [ORG 0x7c00]
2 [BITS 16]
3 stop:
```

[3] Samozřejmě se všemi důsledky.

[4] Což by mělo stačit každému. Bill Gates nejspíše není autorem tohoto citátu, ale ve své době 640 kB představovalo opravdu velké množství paměti.

```
4       jmp stop                ; vstoupime do nekonecne smycky
5
6       times 510 -($-$$) db 0 ; vyplnime zbyvajici prostor 0
7       dw 0xaa55               ; signatura MBR
```

Kód v tomto příkladu obsahuje pouze jednu nekonečnou smyčku (řádky 3 a 4), která je spuštěna po načtení MBR do paměti. Počítač po zavedení našeho kódu nedělá nic jiného, než cyklí. Využijeme tento příklad k tomu, abychom si představili některé direktivy, které se v externím assembleru používají. Na řádku 1, je to direktiva ORG, která assembleru udává, že kód, který bude následovat bude v paměti uložen od adresy `0x0000:7c00`. Direktiva BITS 16 udává, že chceme generovat 16bitový kód.

Dále se zde můžeme setkat s s pseudoinstrukcemi `db`, `dw` a `dd`, které slouží k tomu, aby se do generovaného kódu vložily konkrétní hodnoty, tj. `db` vloží jednobytovou, `dw` dvoubytovou a `dd` čtyřbytovou hodnotu. Na řádku 6 najdeme pseudoinstrukci `db` ve spojení s prefixem `times`, který způsobí to, že se následující instrukce bude opakovat tolikrát, kolik je zadaný počet opakování.

V našem případě, potřebujeme zajistit, že vygenerovaný kód bude mít velikost přesně 512 B, přičemž poslední dva byty ponesou hodnoty `0xaa55`. Proto počet opakování dáme jako 510 a snížíme hodnotu o velikost vygenerovaného kódu. Hodnota `$` odpovídá aktuální adrese v programu, hodnota `$$`, odpovídá adrese začátku generovaného kódu.

Tento kód přeložíme pomocí příkazu:

```
nasm boot-v1.asm -f bin -o boot.bin
```

Přepínač `-f bin` zajistí, že se vygeneruje strojový kód v podobě, kterou lze nakopírovat přímo do paměti a spustit, přepínač `-o boot.bin` udává, že se výstup má uložit do souboru `boot.bin`.

Nyní můžeme zkusit zavést náš kód do emulátoru počítače.

```
qemu-system-i386 -drive format=raw,file=boot.bin
```

Spustíme-li emulátor, nebude se dít nic příliš zajímavého. V emulátoru zůstane vypsaná hláška, že systém bootuje, a emulátor bude spotřebovávat procesorový čas. Což můžeme chápat, jako částečně dobrou zprávu, protože systém nejspíš skončil v nekonečné smyčce, kterou jsme vytvořili, ale nemůžeme si tím být jisti.

3 Basic Input Output System

3.1 Výstup na terminál

Při inicializaci systému toho nemáme mnoho k dispozici, avšak pro komunikaci se základními periferiemi nám BIOS poskytuje to nejnutnější, co je potřeba. Pro záčátek použijeme výstup na terminál k tomu, abychom si ověřili, že náš systém opravdu nabootoval.

Jednotlivé služby BIOSu jsou k dispozici ve formě softwarových přerušení. To znamená, že podle typu úlohy zvolíme přerušení a do jednotlivých registrů uložíme parametry, se kterými se má úloha provést.

Pro práci s terminálem je vyhrazeno přerušení `0x10` a pomocí parametrů specifikujeme jednotlivé operace, viz `http://www.ctyme.com/intr/int-10.htm`. Pro nás bude zajímavá operace představující textový výstup na terminál, kterou získáme tak, že do registru ah vložíme `0x0e` (číslo operace), do registru `al` znak, který se má vypsat. Následně pomocí instrukce `int 0x10` vyvoláme přerušení. V rámci obsluhy tohoto přerušení dojde k vypsání znaku na terminál.

Spojíme-li si kód z předchozí kapitoly s touto službou BIOSu, měly bychom dostat kód, který po spuštění vypíše jeden znak a začne cyklit.

```
[ORG 0x7c00]
[BITS 16]
    mov ah, 0x0e
    mov al, 'H'
    int 0x10

stop:
    jmp stop                ; vstoupime do nekonecne smycky

    times 510 -($-$$) db 0 ; vyplnime zbyvajici prostor 0
    dw 0xaa55               ; signatura MBR
```

Tento kód není opět nic světoborného z očekávatelného textu `Hello world!` vypíše jen první písmeno, ale již nám umožňuje ověřit, že náš kód byl skutečně správně zaveden do paměti a spuštěn.

Chceme-li vypsat celý řetězec, musíme se o to postarat ve vlastní režii. Z pohledu programu to není nijak náročné, čteme postupně jednotlivé znaky řetězce a pomocí služby BIOSu je vypisujeme.

```
1  [ORG 0x7c00]
2  [BITS 16]
3      mov si, hello_txt       ; vlozime adresu retezce
4      mov ah, 0x0e            ; identifikator vystupni operace
5
6  print:
7      mov al, [si]            ; precteme jeden znak
8      cmp al, 0               ; pokud je \0,
9      jz stop                 ; skoncime
10
11     int 0x10                ; vypiseme znak
12     add si, 1               ; a pokracujeme dalsim znakem
13     jmp print
14
15 stop:
16     jmp stop                ; vstoupime do nekonecne smycky
17
18 hello_txt:
```

```
19      db 'Hello world!', 0    ; retezec urceny pro vystup
20
21      times 510 -($-$$) db 0 ; vyplnime zbyvajici prostor 0
22      dw 0xaa55              ; signatura MBR
```

Za povšimnutí v tomto kódu stojí práce s řetezcem, kdy je řetezec vytvořen pomocí pseudooperace `db` na řádku 19 a adresa tohoto řetezce je určena bežným návěštím, viz `hello_txt` na řádcích 3 a 18. Dále si povšimněme, že tento řetezec je umístěn až za nekončenou smyčku. Pokud bychom řetezec umístili před ni, byl by interpretován jako instrukce procesoru, což není žádoucí, protože by to vedlo k provedení nezamýšlených instrukcí.

Úkol: Implementujte v assembleru funkci `print_hex`, která vypíše obsah paměti na dané adrese v hexadecimální soustavě, tj. jednotlivé byty jsou vypsány jako dvojice cifer šestnáctkové soustavy.

3.2 Práce s diskem

Vstup a výstup na terminál patří k tomu nejjednoduššímu, co bychom od BIOSu mohli očekávat. Jeho možnosti jsou však mnohem širší. Můžeme například přistupovat k disku, tj. číst z něj jednotlivé bloky a případně je zapisovat. Pokud můžeme přečíst jednotlivé bloky z disku, můžeme načíst do paměti další kód, který se má provádět.

Z historických důvodů obsahuje BIOS dvě odlišná rozhraní pro práci s diskem. Jedno je postaveno na adresaci jednotlivých bloků pomocí souřadnic, které jsou daný jako cylindr, hlava, sektor (CHS), které počítájí s rotačními disky a vedle toho existuje modernější[5] rozhraní logical block addressing (LBA), která pracuje s logickým uspořádáním bloků, kdy je disk chápán jako souvislý prostor bloků.

3.2.1 Čtení dat z disku

Pro větší pohodlí budeme používat adresaci LBA. Abychom přečetli konkrétní bloky dat z disku, potřebujeme definovat adresy, odkud se budou data číst a kam se budou ukládat. K tomu slouží data address packet, což je datová struktura umístěná v paměti. My ji můžeme inicializovat pomocí pseudooperací `db`, `dw` apod.

```
;
; struktura obsahujici paket s adresami (disk address packet)
; urcena pro cteni dat z disku
;
dapack:
    db 0x10      ; velikost prikazu
    db 0         ; vzdycky 0
blkcnt:
    dw 1         ; pocet bloku k precteni/nactenych (hodnota zmenena po provedeni preruseni)
dest:            ; adresa ciloveho bufferu (0x0000:0x9000)
    dw 0x9000    ; offset v ramci segmentu (0x9000)
```

[5] Přibližně od poloviny devadesátých let minulého století.

```
    dw 0        ; segment (0x0000)
src:
    dd 1        ; adresa bloku na disku
    dd 0        ;
```

Tato struktura nám slouží jako příkaz pro práci s diskem, který můžeme vyvolat pomocí přerušení 0x13 a to následovně. Podrobnosti viz `http://www.ctyme.com/intr/rb-0708.htm`.

```
mov si, dapack          ; adresa paketu s adresami pro prenos dat
mov ah, 0x42            ; identifikator operace
mov dl, 0x80            ; cislo disku 0 (plus 0x80)
int 0x13
```

Protože tato operace může selhat, je žádoucí s tím počítat a nějakým způsobem o tom informovat. To, že operace selhala, je indikováno příznakem CF. Tedy po provedení diskové operace by měla následovat kontrola:

```
jc disk_err
```

Zbývá dořešit, jaká data budeme načítat. Můžeme využít toho, že pomocí NASM de facto generujeme obraz disku, se kterým se pracuje. Doplníme si do něj proto druhý sektor.

```
second_blk:
    db 'Hello from the other sector', 0
    times 512 - ($-second_blk) db 0
```

Že se nám data úspěšně načetla, můžeme ověřit tím, že si necháme vypsat data z adresy `0x0000:9000`, kam jsme data načetli.

Kompletní příklad demonstrující tuto práci s diskem najdete v souboru `boot-v4.asm`.

3.2.2 Data jsou kód, kód jsou data

V předchozí podkapitole jsme do paměti načetli textový řetezec a ten vypsali. Můžeme však využít toho, že na von Neumannově architektuře kód = data, a načíst kód, který se začne provádět, jinými slovy provedeme skok na adresu načtených dat.

```
jmp 0x9000
```

To vyžaduje úpravu druhého sektoru tak, aby obsahoval proveditelný kód. Převedeme tedy do tohoto bloku jak funkci pro výpis dat, tak data samotná a nesmíme zapomenout ani na nekonečnou smyčku.

```
second_blk:
    ; vlozime adresu retezce
    mov si, (0x9000 + hello_txt - second_blk)
```

```
    mov ah, 0x0e  ; identifikator vystupni operace
print:
    mov al, [si]  ; precteme jeden znak
    cmp al, 0     ; pokud je \0,
    jz stop       ; skoncime

    int 0x10      ; vypiseme znak
    add si, 1     ; a pokracujeme dalsim znakem
    jmp print
stop:
    jmp stop      ; vstoupime do nekonecne smycky
hello_txt:
    db 'Hello from the other sector', 0
    times 512 - ($-second_blk) db 0
```

Kompletní příklad demonstrující tuto práci s diskem najdete v souboru `boot-v5.asm`.

Všimněme si, že tímto způsobem již můžeme do paměti načíst libovolně dlouhý kód a nejsme omezeni místem v MBR. Máme tak prostor pro načtení operačního systému, který je schopen vykonávat komplexnější akce. Nutno dodat, že se stále pohybujeme v 16bitovém režimu procesoru.

4 Poznámky závěrem

- V tento okamžik máme již docela široké znalosti, se kterými bychom mohli v krátké době vytvořit jednoduchý operační systém typu CP/M nebo MS-DOS. Chybí nám jen implementace souborového systému.
- Nemuseli bychom se ale omezovat jen na operační systém. Z disku bychom mohli místo tradičního operačního systému zavést třeba interpreter nějakého programovacího jazyka, který bude sloužit podobně jako operační systém, z těch jednodušších na implementaci se nabízí BASIC, Forth, fajnšmekři mohou sáhnout po Lisp/Schemu.
- V dnešních systémech se dá spíše setkat se systémem UEFI, který se stará o mnohem pokročilejší inicializaci systému a načtení operačního systému.