ŽILINSKÁ UNIVERZITA V ŽILINE

Fakulta riadenia a informatiky

BAKALÁRSKA PRÁCA

Stupeň kvalifikácie: bakalár

Študijný odbor: informatika

Študijný program: informatika

Martin Kolář

Chránený režim procesorov Intel 80386 a vyšších

Vedúci práce:
doc.Ing. Ľudmila Jánošíková, PhD.

Reg.č.: 31/2008 Máj 2009

Dátum odovzdania práce: 25.5.2009

Abstrakt

Slovenský:
KOLÁŘ, MARTIN: Chránený režim procesorov Intel 80386 a vyšších [bakalárska práca] – Žilinská univerzita v Žiline. Fakulta riadenia a informatiky; Katedra dopravných sietí. – Vedúci: doc.Ing Ľudmila Jánošíková, PhD. – Stupeň odbornej kvalifikácie: Bakalár v odbore Informatika. Žilina: FRI ŽU v Žiline, 2009. – 52 strán

Cieľom tejto bakalárskej práce je zoznámiť čitateľov s princípmi a vlastnosťami chráneného režimu procesorov Intel 80386 a vyšších, vysvetliť im mechanizmus fungovania správy pamäti a spracovania prerušení v chránenom režime, princíp spolupráce reálneho a chráneného režimu a demonštrovať jeho funkčnosť na praktických ukážkových príkladoch.

Kľúčové slová: chránený režim, virtuálny režim, privilegované inštrukcie

Poďakovanie:

Rád by som predovšetkým poďakoval vedúcej mojej bakalárskej práce paní doc.Ing. Ľudmile Jánošíkovej, PhD. za poskytnutú podporu pri vypracovávaní tejto práce a tiež mojej rodine, ktorá mi poskytla potrebné zázemie.

Čestné prehlásenie:

Čestne prehlasujem, že som túto prácu vypracoval samostatne a uviedol som všetky zdroje a literatúru, z ktorých som čerpal.

V Žiline dňa 12.5.2009 _____

Martin Kolář

Obsah

1. Úvod

Informačné technológie v posledných rokoch prechádzajú veľmi búrlivým vývojom, pretože od nich ostatné odvetvia očakávajú čoraz lepšie výsledky. Vývojári navrhujú a programátori programujú stále zložitejšie a komplexnejšie aplikácie, ktoré kladú vysoké nároky na operačnú pamäť, výpočtový výkon, bezpečnosť dát, spoľahlivosť atď. Preto ruka v ruke s vývojom softvéru sa vyvíja a neustále zdokonaľuje aj hardvér, ktorý má byť schopný neustále náročnejšie aplikácie rozumne prevádzkovať.

V tejto práci som sa zameral predovšetkým na procesor, ktorý je ako zástupca hardvéru najdôležitejší, pretože tvorí najzákladnejšie jadro počítača a všetky výpočtové úkony sa vykonávajú práve na ňom.

Firma Intel patrí medzi svetovú špičku v rámci návrhu a výroby procesorov a tým pádom aj určuje smer, ktorým sa tieto zariadenia vyvíjajú. V dnešnej dobe je ešte stále tendencia pokračovať vo vývoji tzv. x86-kompatibilných procesorov (pre osobné počítače), ktoré vychádzajú z princípov starého 16-bitového procesora Intel 8086. Preto aj na najnovšom 64-bitovom procesore, ktorý vychádza z tejto rady, nie je problém natívne prevádzkovať softvér napísaný pre zmienený starý procesor.

Od vydania procesoru Intel 8086 už ale uplynuli 3 desaťročia a za tú dobu sa nároky na hardvér podstatne zmenili. Pôvodný procesor bol určený na real-time prácu jednej aplikácie, vystačil si na našu dobu s veľmi malou operačnou pamäťou a nároky na ostatný hardvér boli taktiež nízke. Procesor Intel 80386, ktorý je témou mojej bakalárskej práce, vychádza z tohto procesoru, ale podstatne rozširuje jeho možnosti.

Procesor Intel 80386 obsahuje navyše predovšetkým ***chránený režim***, ktorý otvára aplikáciám úplne nové možnosti, aké by na pôvodnom procesore neboli možné. Pridáva možnosť ochrany hardvéru, operačnej pamäti a napokon aj samotného procesora pred chceným či nechceným zneužitím zo strany aplikácií. Tieto aplikácie v prípade behu pod moderným operačným systémom, ktorý funguje v chránenom režime a využíva jeho možnosti ochrany, dostanú pridelené len určité systémové prostriedky a trebárs len na určitý čas. Týmto aplikáciám nie je umožnené zasahovať mimo vymedzené prostriedky, lebo v prípade porušenia pravidiel riadenie procesora dostane operačný systém a ten už rozhodne, čo s previnilou aplikáciou spraví. Chránený režim znamená predovšetkým nie ochranu našej aplikácie pred ostatnými, ale ochranu systému a ostatných aplikácií pred našou.

Chránený režim tiež priamo podporuje hardvérový multitasking, čiže procesor Intel 80386 obsahuje v sebe prostriedky, ktoré umožňujú pseudo-paralelný beh viacerých aplikácií v reálnom čase a o prepínanie ich kontextu sa postará procesor.

Cieľom mojej bakalárskej práce je detailne popísať problematiku chráneného režimu, teda jeho funkčnosť, použitie a využitie a vytvoriť sériu demonštračných programov, ktoré budú jeho praktickou ukážkou.

2. Výpočet adresy a správa pamäti v reálnom, chránenom a virtuálnom režime

2.1 Reálny režim

2.1.1 Procesor Intel 8086

V roku 1979 firma Intel predstavila svoj prvý plne 16-bitový procesor Intel 8086, ktorý obsahoval sadu 16-bitových univerzálnych registrov *(AX,BX,CX,DX)*, sadu 16-bitových registrov pre adresovanie *(SI,DI,BP,SP)*, sadu 16-bitových registrov pre podporu segmentácie *(CS,DS,ES,SS)*, 16-bitový čítač inštrukcií *(IP)* a príznakový register.

Tento procesor bol na svoju dobu veľmi flexibilný, jeho pracovná frekvencia bola 5 – 10 MHz a ujal sa ako štandard pre ďalší vývoj. Rada procesorov vychádzajúca z tohto procesora sa označuje pod skratkou x86.

Tento procesor je zástancom reálneho režimu, pretože on sám síce obsahuje jediný režim práce, ale na novších procesoroch sa tento režim práce označuje ako reálny, aby sa odlíšil od ostatných režimov práce.

2.1.2 Adresovanie procesoru Intel 8086

Intel 8086 obsahuje 20-bitovú adresnú zbernicu, z čoho vyplýva jeho schopnosť priamo adresovať 1 MB pamäti (2^{20} = 1 048 576 bajtov pamäti). Problém je ale v tom, že indexové registre a registre schopné adresovania sú iba 16-bitové a teda sú schopné adresovať len 64 KB pamäti (2^{16} = 65 536 bajtov pamäti).

Dizajnéri procesora Intel 8086 preto navrhli využiť pre správu pamäti segmentáciu, pričom sa využívajú 4 16-bitové segmentové registre:

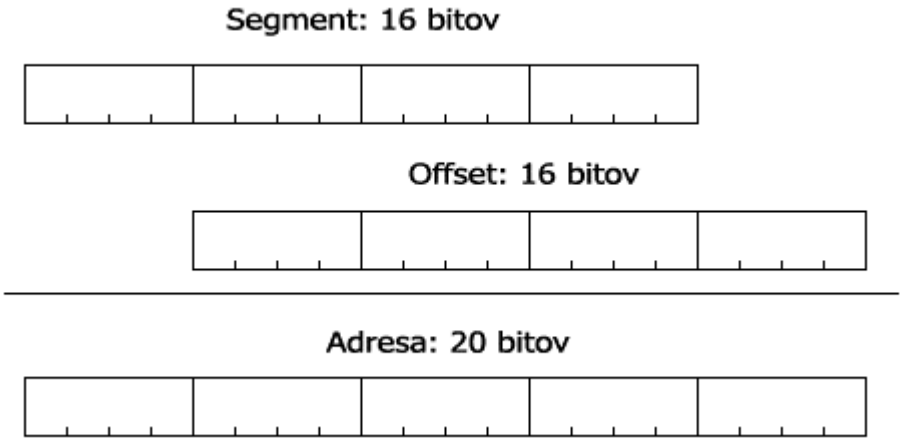
CS – *kódový segment*
DS – *dátový segment*
ES – *extra segment*
SS – *segment zásobníka*

Týmto spôsobom sa pri akejkoľvek inštrukcii procesora vždy prihliada na hodnoty uložené v týchto registroch a procesor si pomocou týchto registrov vie spočítať fyzickú adresu uložených dát v pamäti. Teraz sa jedná o spôsob, akým procesor túto fyzickú adresu vypočíta.

Uvažujme, že máme 16-bitový segmentový register (napríklad **DS**) a 16-bitový register schopný adresovania (napríklad **BX**). Vzhľadom na to, že súčet týchto bitov je 32, dostali by sme možnosť adresovať pomocou tejto dvojice registrov **DS:BX** pamäťové pole o veľkosti 4 GB (2^{32} = 4 294 967 296 bajtov pamäti). Toto by platilo, pokiaľ by offset v rámci každého segmentu (**BX**) začínal od nuly, teda každý segment by mal vyhradených celých 64 KB a každý ďalší segment by začínal za tým predošlým.

V roku 1979, keď bol tento procesor predstavený, boli 4 gigabajty pamäti príliš veľa a vytvoriť takýto segmentačný model by bolo dosť nepraktické. Pokiaľ by sme uvažovali o alokácii pamäti po jednotlivých segmentoch, tak by to viedlo k veľmi vysokému stupňu

vnútornej fragmentácie, pretože do 1 megabajtu pamäti sa vojde len 16 segmentov o veľkosti 64 KB. Preto tvorcovia procesora zvolili metódu prekrývania jednotlivých segmentov podľa nasledovnej schémy:



Obr. 1 – Výpočet fyzickej adresy procesorom 8086

Zo schémy na obr.1 vyplýva, že veľkosť jedného segmentu je stále 64 KB. V rámci 1 segmentu používame 16-bitový offset (register, pomocou ktorého adresujeme v rámci segmentu, je 16-bitový) a môžeme tým pádom alokovať pamäťové pole o veľkosti 65 536 bajtov.

Rozdiel je v tom, že jednotlivé segmenty sú od seba posunuté nie o maximálnu veľkosť 64 KB (16 bitov v adrese), ale iba o 16 bajtov, čomu zodpovedajú 4 adresové bity. Nasledujúci segment, teda taký segment, ktorý má v segmentovom registri vyššiu hodnotu o 1, má v skutočnosti fyzickú adresu väčšiu o 16 bajtov, pokiaľ uvažujeme počítanie offsetu od nuly.

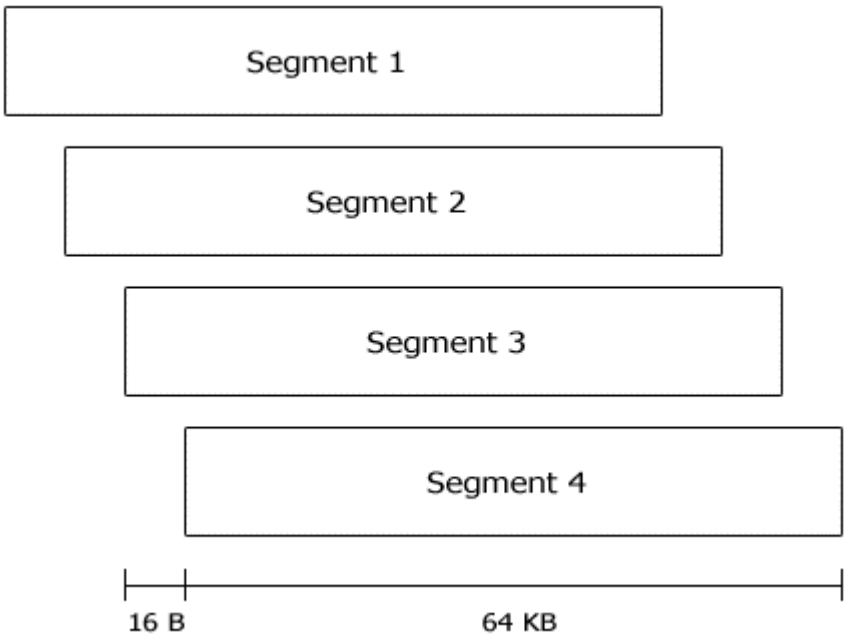
Na výpočet fyzickej adresy používa procesor 8086 nasledujúci vzorec:

fyzická adresa = (segment * 16) + offset

Násobenie mocninou 2 a delenie mocninou 2 (bezo zvyšku) je pre procesor veľmi rýchla záležitosť, pretože procesor počíta s bitmi v dvojkovej sústave a v tomto prípade sa jedná len o posun hodnoty registrov smerom doprava (pri delení) alebo doľava (pri násobení). Preto tento vzorec môžeme napísať aj nasledovným spôsobom:

fyzická adresa = (segment << 4) + offset

Na nasledujúcom obr.2 je znázornené usporiadanie jednotlivých segmentov vo fyzickej pamäti:



Obr. 2 – Fyzické usporiadanie segmentov v pamäti

Z uvedeného spôsobu adresovania pomocou páru segment:offset vyplýva ešte jeden dôležitý fakt, a to taký, že procesor môže vygenerovať aj adresy v rozsahu 0x100000 – 0x10FFEF, ktoré ležia už za hranicou 1 MB, ktorú je procesor 8086 schopný adresovať. Jedná sa presne o 65520 bajtov, ktoré ležia za hranicou 1. megabajtu a procesor týmto adresám oreže najvyšší bit, pretože 20-bitová adresná zbernica také adresy nedokáže pojať a tým pádom sa tieto adresy premietnu na začiatok adresného priestoru.

2.2 Chránený režim

2.2.1 Procesor Intel 80286

Ďalším z rady x86 procesorov Intel je procesor 80286. Bol predstavený v roku 1982 a rozširuje možnosti procesoru 8086. Takisto obsahuje sadu 16-bitových registrov rovnakých ako jeho predchodca 8086, no zároveň obsahuje niektoré veľmi výrazné zmeny, ktoré z neho z hľadiska programovania robia akoby inú architektúru.

V prvom rade procesor 80286 obsahuje ***24-bitovú adresnú zbernicu***. Z toho vyplýva, že tento procesor je schopný adresovať celých ***16 megabajtov*** operačnej pamäti (2^{24} = 16 777 216 bajtov pamäti), no spomínaný systém adresovania, ktorý obsahuje procesor 8086, dokáže priamo adresovať len 1 MB pamäti. Preto (ale nie len preto) obsahuje tento procesor špeciálny režim, ktorý sa nazýva chránený režim *(protected mode)*, ktorý prináša tieto zmeny:

- **nový spôsob adresovania pamäti**
- **nový spôsob spracovania prerušení**
- **podpora multitaskingu**
- **systém ochrany pamäti a hardvéru**

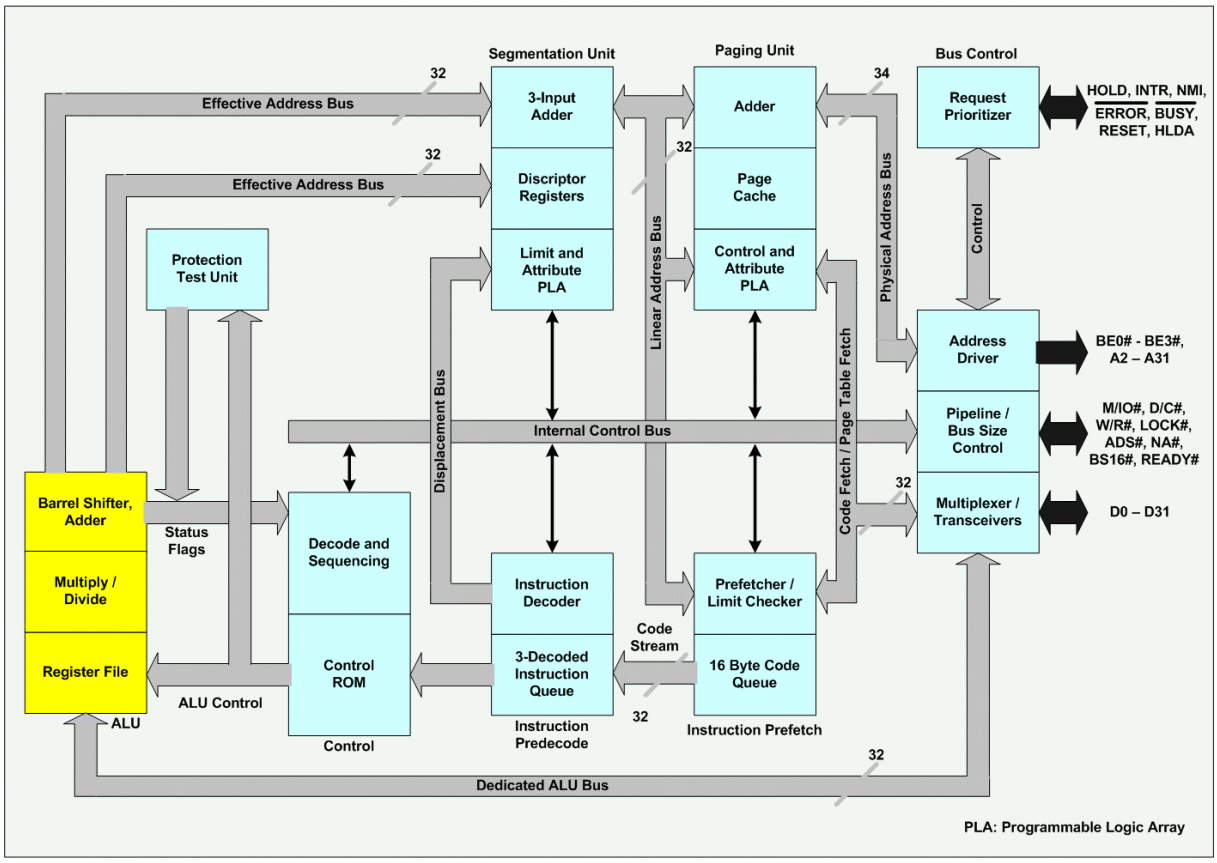
Dôležité je uvedomiť si, že tento nový režim práce procesora ***nie je*** kompatibilný so starým režimom práce procesora 8086. Preto procesor 80286 po svojom naštartovaní pracuje v pôvodnom režime práce 8086, tiež nazývanom ***reálny mód*** *(real mode)* a priamo môže pristupovať iba do 1. megabajtu pamäti. Prepnutie do nového režimu práce vykoná softvér, ktorý na to používa špeciálne, tzv. ***privilegované inštrukcie***, ktoré sú určené na správu chráneného režimu.

Tento softvér je väčšinou operačný systém, ktorý zastrešuje hardvér počítača pre bežné programy, ale môže sa jednať aj o tzv. ***extender***, ktorý sprostredkúva výhody chráneného režimu pre aplikácie, ktoré vyžadujú napr. dostatočné množstvo operačnej pamäti aj v operačnom systéme, ktorý chránený režim nepodporuje.

Vzhľadom na to, že chránený režim procesora 80286 nie je náplňou mojej bakalárskej práce, tak začnem popisovať až chránený režim procesora 80386. Procesor 80286 obsahuje aj pár závažných nedostatkov vyplývajúcich z nedostatočného návrhu, ktoré znižujú atraktivitu použitia tohto režimu a zvyšujú jeho ťažkopádnosť.

2.2.2 Procesor Intel 80386

Procesor 80386 bol predstavený v roku 1986 ako procesor plne 32-bitový a zároveň ako procesor plne kompatibilný so starým 16-bitovým procesorom 8086. Obsahuje mnoho vylepšení oproti svojim predchodcom, a preto poskytuje omnoho vyššiu flexibilitu spracovania inštrukcií, pokročilejšie možnosti správy operačnej pamäti a veľa ďalších vylepšení. No kvôli týmto vlastnostiam procesor 80386 má aj o dosť zložitejší dizajn, ako jeho predchodcovia. Na nasledujúcom obrázku je zobrazená základná architektúra tohto procesora.



Obr. 3 – Architektúra procesora Intel 80386
[zdroj 2]

Procesor 80386 obsahuje sadu 32-bitových registrov, a to:

Univerzálne registre:

- **EAX** (akumulátor)
- **EBX** (bázový register)
- **ECX** (čítač, používa sa napr. pri cykloch)
- **EDX** (dátový register, rozširuje register EAX)
- **ESI** (source index – index pre zdroj)
- **EDI** (destination index – index pre cieľ)
- **EBP** (base pointer – ukazovateľ na dáta v zásobníku)
- **ESP** (stack pointer – ukazovateľ na vrchol zásobníku)

Čítač inštrukcií:

- ***EIP*** (instruction pointer – ukazovateľ na nasledujúcu inštrukciu, nie je priamo prístupný)

Register príznakov:

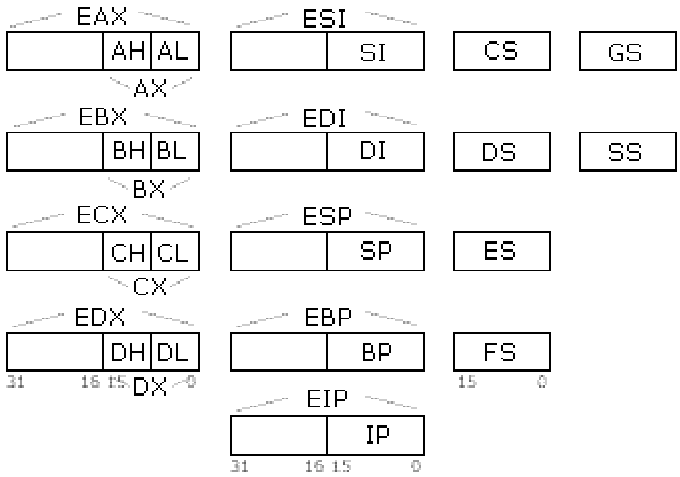
- **EFLAGS** (obsahuje jednotlivé príznakové bity, ako CF, PF, AF, ZF atď.)

Procesor 80386 obsahuje aj sadu 16-bitových registrov, a to:

Segmentové registre

- **CS** (code segment – kódový segment)
- **DS** (data segment – dátový segment)
- **SS** (stack segment – segment zásobníku)
- **ES** (extra segment – segment pre voľné využitie)
- **FS** (extra segment – segment pre voľné využitie)
- **GS** (extra segment – segment pre voľné využitie)

Oproti procesoru 80286 je najmarkantnejší rozdiel práve v rozšírení univerzálnych registrov zo **16** na **32** bitov. Predpona **E** u týchto registrov znamená anglickú skratku *extended*, tzn. rozšírenie. Tieto registre (napr. **EAX**) sa rozdeľujú na horných a dolných **16** bitov, pričom horných **16** bitov nie je priamo prístupných a dolných **16** bitov je priamo prístupných ako **16**-bitový register (napr. **AX**). Týchto dolných **16** bitov sa naďalej rozdeľuje na horných a dolných **8** bitov, pričom obe časti sú priamo prístupné ako **8**-bitové registre (napr. **AH** a **AL**). Nasledujúci obrázok ilustruje práve toto rozdelenie:



Obr. 4 – Rozdelenie registrov procesoru Intel 80386

Ako už bolo spomenuté, procesor Intel 80386 je ďalším nasledovníkom rady x86 a zachováva si plnú kompatibilitu s predošlými procesormi (8086, 80286). Oproti predošlým procesorom ale obsahuje toľko nových vylepšení a rozšírení, že je ho možné označiť aj za úplne novú architektúru. Nový softvér z obdobia, kedy sa rozšíril tento procesor, vôbec na starších procesoroch nefungoval, pretože väčšinou obsahoval 32-bitový kód, ktorý funguje až od 80386. Softvér, ktorý prevádzkuje 32-bitový kód, používa predovšetkým 32-bitové univerzálne registre, 32-bitové adresovanie (offset v rámci segmentu) a funguje v chránenom režime.

Všetok nový softvér využíva prinajmenšom inštrukčnú sadu Intel 80386, pretože je 32-bitový a 16-bitový softvér je už dávno považovaný za zastaraný. Nový 32-bitový softvér potrebuje ku svojej činnosti 32-bitové registre a 32-bitový chránený režim, ktorý mu je

schopný ponúknuť až procesor 80386. Procesor Intel 80386 obsahuje 3 rôzne režimy práce, ktoré umožňujú jednak spätnú kompatibilitu so staršími procesormi a jednak pridávajú nové funkcie procesoru, ako napr. už spomínaný 32-bitový režim.

Procesor Intel 80386 obsahuje tieto režimy práce:

- **reálny režim** (real mode)
- **chránený režim** (protected mode)
- **virtuálny režim** (virtual 8086 mode)

2.2.3 Režimy procesora Intel 80386

Reálny režim

Po spustení alebo reštarte počítača sa inicializuje okrem iného hardvéru aj procesor. Pôvodný x86 procesor 8086 obsahoval iba jediný režim práce a po reštarte procesora sa tento inicializoval na určitú počiatočnú adresu, na ktorej sa nachádzala prvá inštrukcia na vykonanie, obvykle inštrukcia systému **BIOS**.

Procesor 80386 sa chová úplne rovnako, tiež po svojom reštarte sa inicializuje a rovnakým spôsobom sa nastaví na počiatočnú adresu prvej inštrukcie. Tento procesor začína svoj chod práve v **reálnom režime**, pretože tento režim je 100% kompatibilný s prácou procesora 8086.

Reálny režim procesora 80386 sa vyznačuje nasledujúcimi vlastnosťami:

- **procesor má priamy prístup len do 1. megabajtu pamäti**
- **správa pamäti je rovnaká ako u procesoru 8086**
- **spracovanie prerušenia je rovnaké ako u procesoru 8086**
- **všetky segmenty sú 16-bitové**

Chránený režim

Hlavné využitie procesora 80386 je práve v efektívnom využívaní jeho rozšírenej inštrukčnej sady a schopnosti spracovávať 32-bitový kód. Každý moderný softvér, ktorý bol navrhnutý pre procesory 80386 a lepšie, využíva nové vlastnosti procesora 80386 a je 32-bitový.

Aby bol procesor schopný spúšťať takýto softvér, musí byť prepnutý do chráneného režimu, ktorý je ale oproti pôvodnému reálnemu režimu značne odlišný. Chránený režim funguje od základov úplne iným spôsobom, ako pôvodný režim procesoru 8086 a obsahuje množstvo nových možností oproti starému režimu.

Medzi hlavné rysy chráneného režimu patrí:

- **umožňuje priamo adresovať 4 GB operačnej pamäti**
- **obsahuje komplexný systém ochrany systémových prostriedkov**
- **obsahuje nový spôsob správy pamäti**
- **obsahuje nový spôsob spracovania prerušení**
- **hardvérovo podporuje multitasking**

- **nie je možné zapisovať do kódového segmentu**

Chránený režim *(protected mode)* dostal svoj názov právom. Hlavná filozofia tohto režimu je chrániť bežiaci softvér na počítači pred iným softvérom. Z pohľadu programátora to znamená, že naša aplikácia, ktorú vyvíjame, je chránená pred inými aplikáciami a aj ostatné aplikácie sú chránené pred tou našou.

V reálnom režime neexistovala možnosť ochrany. Nebol problém naprogramovať a spustiť program, ktorý prepísal celú operačnú pamäť napr. náhodnými číslami a samozrejme spôsobil pád celého systému. Niekedy stačí naozaj málo, napr. prepísať jediný bajt na správnom mieste, ako je prepísanie 1 bajtu v tabuľke vektorov prerušení a systém sa zrúti po prvom vyvolaní tohto prerušenia.

Toto sú dôvody, prečo užívateľské aplikácie nemôžu mať neobmedzený prístup k hardvérovým prostriedkom počítača. Samozrejme nejedná sa iba o záškodný softvér, ako sú rôzne druhy vírusov, ale aj o normálny softvér, ktorý takmer vždy obsahuje nejaké chyby, ktorým sa programátor nevyhol a tieto chyby tiež môžu narušiť integritu celého operačného systému.

Operačný systém je prvý, ktorý potrebuje byť chránený od vplyvu ostatného softvéru. Chránený mód presne toto zabezpečuje – poskytuje 4 rôzne úrovne oprávnenia, ktoré určujú, ktorá akcia je softvéru s daným oprávnením dovolená a ktorá nie je, resp. ktorá vyvolá prerušenie procesora.

Virtuálny režim

Virtuálny režim sa ako prvý objavil práve u procesoru 80386 ako dôsledok značnej neefektívnosti pri spracovávaní kódu napísaného v štýle reálneho módu a potrebe chráneného režimu. Keď spustíme program, ktorý je určený pre chránený režim, tak tento program často potrebuje komunikovať so systémom (minimálne s BIOSom) a pri tejto komunikácii sa musí prepínať neustále medzi reálnym a chráneným režimom (viď kapitola 4. - Volanie funkcií BIOSu a DOSu z chráneného režimu). Toto prepínanie stojí procesor nemalý čas a po opustení chráneného režimu hoci aj na krátku dobu je počítač v podstate bez systému ochrany, pokročilej správy pamäti atď. Preto dizajnéri procesora 80386 navrhli nový režim, ktorý je akýmsi hybridom medzi chráneným a reálnym režimom a vyznačuje sa nasledovnými vlastnosťami:

- **kód je 16-bitový**
- **všetky segmenty sú 16-bitové**
- **adresovanie je v štýle reálneho režimu**
- **je možné využiť stránkovanie**
- **je možné zapisovať do kódového segmentu**
- **systém ochrany zostáva zapnutý**
- **prerušenia sú v štýle chráneného režimu**

Moderné operačné systémy fungujú v chránenom režime a pri potrebe spustiť kód určený pre reálny režim neprepnú procesor priamo do reálneho režimu, ale do režimu virtuálneho. Týmto spôsobom si operačný systém neustále zachováva svoju prevahu nad softvérom a nehrozí, že by kód spustený vo virtuálnom režime mohol na systéme napáchať nejaké škody (napr. prepísať systémové tabuľky), ako by to mohlo hroziť v reálnom režime.

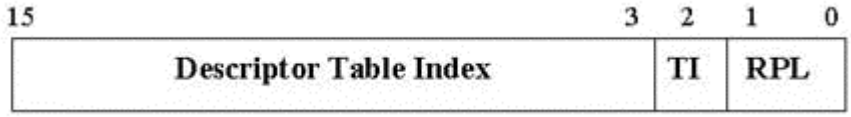
Vo virtuálnom režime procesor v podstate zostáva v režime chránenom, akurát kód, ktorý spracúva, tak o ňom vie, že je určený pre reálny režim a je 16-bitový a podľa toho ho aj procesor vykonáva. Okrem iného odpadá aj nákladná réžia procesora pri prepínaní medzi reálnym a chráneným režimom, kedy procesor v podstate musí pozmeniť dosť značnú časť systémových prostriedkov.

Virtuálny režim má ešte jednu veľkú výhodu, a to, že podporuje stránkovanie. Týmto spôsobom kód určený pre reálny režim vôbec nemusí byť umiestnený fyzicky na začiatku pamäti, ako je to v prípade reálneho režimu, ale môže byť umiestnený hocikde v pamäti a pomocou mechanizmu stránkovania sa v podstate ľubovoľná fyzická adresa preloží na zodpovedajúcu logickú adresu, ktorá sa bude tváriť, že spadá do 1. megabajtu.

2.2.4 Segmentácia

Procesor 80386, pokiaľ beží v chránenom režime, využíva ako hlavný model správy pamäti ***segmentáciu***. Pri segmentácii procesor využíva segmentové registre procesora ***CS***,***DS***,***ES***,***FS***,***GS*** a ***SS***, pričom hodnota v nich uložená sa nepoužíva na priamy výpočet lineárnej adresy, ako to bolo v prípade reálneho režimu. Hodnota v nich uložená slúži ako odkaz do tabuľky tzv. ***deskriptorov***, ktoré obsahujú informácie potrebné na výpočet adresy. 16-bitové číslo, ktorého hodnota sa priamo ukladá do segmentových registrov, sa nazýva ***selektor***.

Na nasledujúcom obrázku je zobrazená štruktúra selektora:

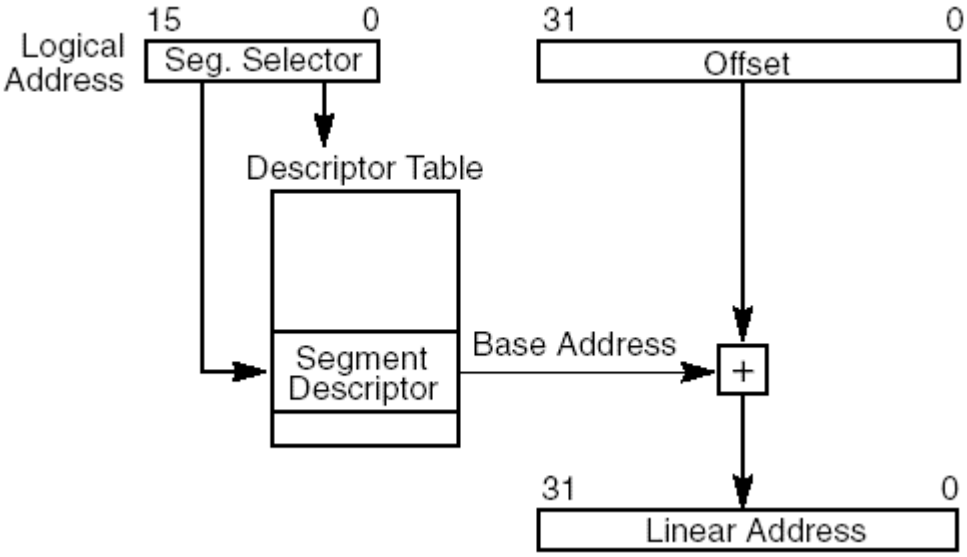


Obr. 5 – Štruktúra selektora

Popis významu jednotlivých položiek:

- **Descriptor Table Index** – toto 13-bitové číslo udáva index do tabuľky deskriptorov, celkovo je možných 8192 rôznych indexov.
- **TI** – ***Table Indicator*** - tento 1 bit určuje, či sa selektor odkazuje na ***GDT*** *(Global Descriptor Table)* – keď je 0, alebo či sa selektor odkazuje na ***LDT*** *(Local Descriptor Table).*
- **RPL** – tieto 2 bity určujú tzv. ***Requestor's Privilege Level***. Ich hodnota slúži ako ukazovateľ oprávnenia daného selektora.

Momentálne sme si zaviedli veľa nových pojmov, ktoré nezasvätený čitateľ nepozná. Preto sa ich pokúsim postupne všetky vysvetliť. Na nasledujúcom obrázku je znázornené, ako prebieha samotný mechanizmus segmentácie:



Obr. 6 – Mechanizmus segmentácie v chránenom režime

Ako vidíme, tak procesor má v segmentovom registri uloženú hodnotu ***selektora***. Pokiaľ chceme do daného segmentu pristúpiť, teda z neho čítať alebo do neho zapisovať, procesor vykoná radu úkonov a buď nám danú akciu povolí (ak nasledovný test nevyjde), alebo ju zakáže (ak nasledovný test vyjde), lepšie povedané nevykoná ju a namiesto nej vyvolá výnimku procesora. Hodnotu ***selektora*** procesor najprv rozloží po bitoch a skontroluje, či môžeme k danému segmentu pristupovať. Použije na to nasledovný vzorec, pričom jednotlivé hodnoty **DPL**, **CPL** a **RPL** sú vyjadrené numericky a platí, že najvyššie oprávnenie má hodnotu **0** a najnižšie oprávnenie má hodnotu **3**:

DPL < max (CPL , RPL)

RPL *(Requestor's Privilege Level)* označuje hodnotu uloženú v selektore, ktorý odkazuje na segment, ku ktorému chceme pristúpiť, ***CPL*** *(Current Privilege Level)* označuje hodnotu oprávnenia, v ktorej beží náš kód (teda dolné 2 bity selektora **CS**) a ***DPL*** *(Descriptor Privilege Level)* označuje hodnotu, ktorá je uložená v deskriptore segmentu, ku ktorému chceme pristúpiť.

Ak **DPL < max (CPL , RPL)**, procesor vyvolá výnimku, tzv. ***GPF*** *(General Protection Fault)*. Pokiaľ vyjde, procesor pokračuje ďalej. Prečíta príslušný deskriptor zo odpovedajúcej tabuľky deskriptorov, ktorý získa z tabuľkového indexu a tabuľkového indikátora. Z tohto deskriptora procesor prečíta bázovú adresu, ktorá určuje lineárnu adresu začiatku príslušného segmentu. Potom túto adresu pripočíta k offsetu, ktorý sme zadali ako súčasť adresy a tým získa lineárnu adresu, na ktorej sa nachádza nami požadovaná pamäťová bunka.

Deskriptor navyše obsahuje aj limit segmentu, buď v bajtoch alebo v 4-kilobajtových stránkach a procesor testuje, či nami daný ofset, ktorým pristupujeme do segmentu, neprekračuje tento limit:

offset <= limit

Ak táto podmienka nie je splnená, tak procesor opäť vyvolá ***GPF*** *(General Protection Fault)*.

V systéme sa môžu nachádzať 2 rôzne tabuľky deskriptorov, a to:

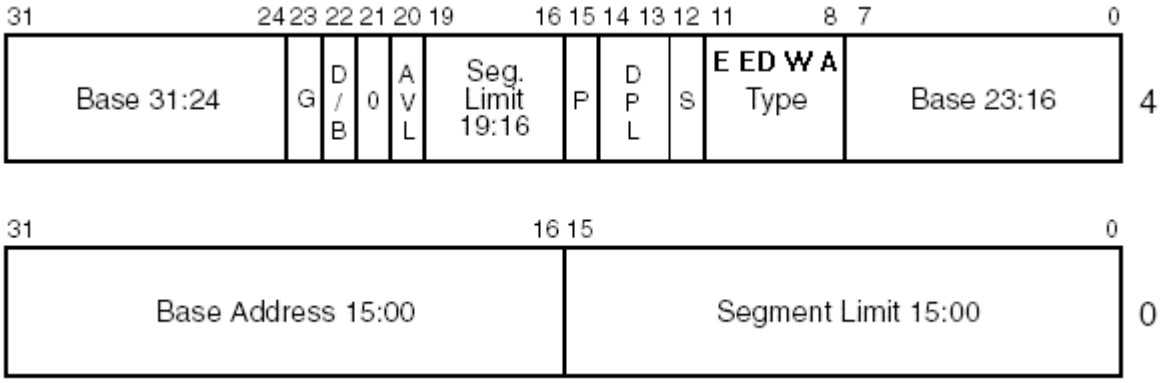
- **GDT** – Global Descriptor Table (Globálna tabuľka deskriptorov)
- **LDT** – Local Descriptor Table (Lokálna tabuľka deskriptorov)

Každá z týchto tabuliek môže mať maximálne 8192 položiek *(deskriptorov)*, pričom 1 položka má veľkosť 8 bajtov. ***GDT*** je určená na to, aby sa nemenila s prepínaním úloh a obsahovala prevažne systémové deskriptory, ktoré potrebuje ku svojej činnosti napr. jadro operačného systému. ***LDT*** je určená na to, aby v prípade multitaskového operačného systému mala každá úloha svoju vlastnú tabuľku deskriptorov, ktorá sa zmení s prepnutím úlohy.

Pred prepnutím procesora do chráneného režimu je mu potrebné najskôr dať vedieť, kde sa tieto tabuľky v pamäti nachádzajú. Slúžia na to inštrukcie ***LGDT*** *(Load Global Descriptor Table)* a ***LLDT*** *(Load Local Descriptor Table)*, ktoré budú bližšie popísané v kapitole o privilegovaných inštrukciách. Prepínanie tabuliek ***LDT*** je pomerne rýchla záležitosť, aby mohol multitasking efektívne fungovať.

Pozorný čitateľ si isto všimol, že neustále spomínam pojem deskriptor a v podstate nič bližšie som k nemu nepovedal. ***Deskriptor*** je 8-bajtová dátová štruktúra, ktorej rozumie procesor a má presne popísaný význam jednotlivých bitov. Deskriptor obsahuje množstvo informácií, ktoré sú potrebné pre správu systému v chránenom režime, ako je napr. v našom prípade segmentácia. Deskriptorov v tabuľke môže byť až 8192, to znamená, že celá tabuľka je maximálne 64 kilobajtov dlhá.

Na nasledujúcom obrázku je zobrazená štruktúra deskriptora pre správu segmentov:



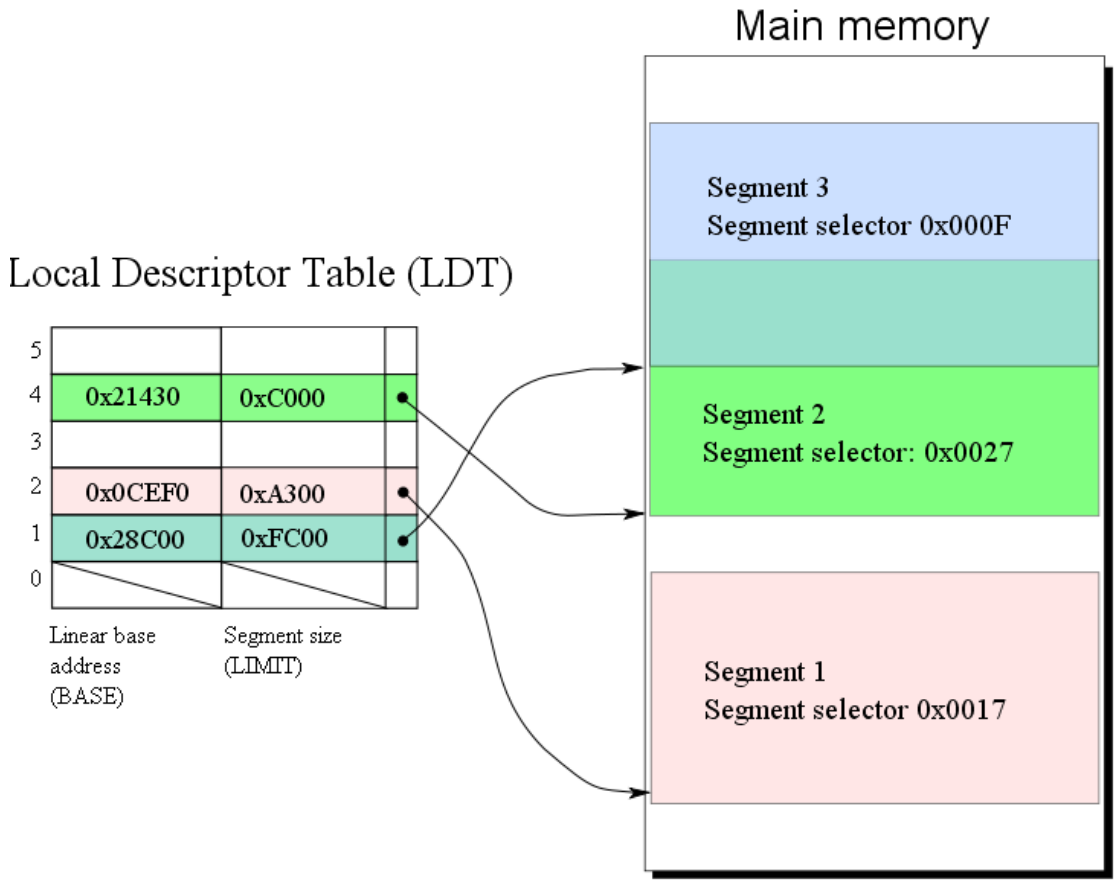
Obr. 7 – Štruktúra deskriptora pre správu segmentov

Popis štruktúry deskriptora pre správu segmentov:

- **Segment Limit** (20 bitov) – určuje dĺžku segmentu, buď v bajtoch alebo v 4-kilobajtových stránkach
- **Base Address** (32 bitov) – určuje lineárnu adresu, na ktorej sa začína daný segment
- **A** (1 bit) - ***Accessed***, bit je nastavený na 1 po každej read/write operácii v segmente
- **W** (1 bit) – ***Writeable***, ak je 1, potom je možné do segmentu zapisovať (dátový segment)

- **ED** (1 bit) – ***Expand Down***, ak je 1, potom má segment fixnú hornú hranicu, používa sa hlavne pre zásobník (dátový segment)
- **E** (1 bit) – ***Executable***, ak je 1, potom sa jedná o kódový segment, ak je 0, potom sa jedná o dátový segment
- **S** (1 bit) – ***System***, ak je 0, jedná sa o systémový deskriptor, inak o kódový alebo dátový deskriptor
- **DPL** (2 bity) – ***Descriptor Privilege Level***, určuje hodnotu oprávnenia daného segmentu
- **P** (1 bit) – ***Segment Present***, tento bit v tomto prípade znamená, či sa daný segment nachádza v operačnej pamäti
- **AVL** (1 bit) – voľný bit pre voľné použitie
- **D/B** (1 bit) - implicitný spôsob adresovania v segmente, ak je 0, segment je 16-bitový, inak je 32-bitový (veľmi dôležité u kódových segmentov)
- **G** (1 bit) – ***Granularity***, určuje, či je veľkosť segmentu udaná v bajtoch (pri 0), alebo v stránkach (pri 1)

Na nasledujúcom obrázku je pre názornosť zobrazený príklad troch segmentov umiestnených v pamäti, pričom dva z nich sa čiastočne prekrývajú:



Obr. 8 – Príklad umiestnenia segmentov v pamäti
[zdroj 3]

2.3 Virtuálny režim

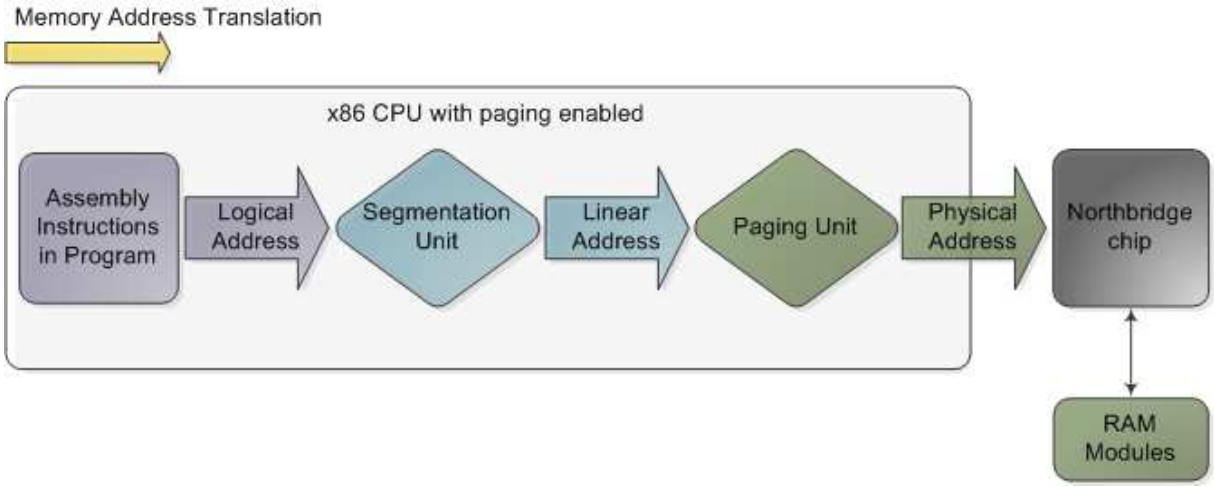
2.3.1 Stránkovanie

Virtuálny režim, ako už bolo spomenuté, je akýsi mix medzi reálnym a chráneným režimom. Procesor v skutočnosti neopúšťa chránený režim, avšak adresovanie pomocou segmentových registrov vykonáva v štýle reálneho módu. Programy napísané pre reálny režim očakávajú, že segmentové registre môžu nadobudnúť akúkoľvek hodnotu, že môžu zapisovať do kódového segmentu atď.

Toto všetko im virtuálny režim umožňuje. Avšak v prípade segmentácie v chránenom režime, táto pracuje iba s ***lineárnymi adresami***. Lineárna adresa je kombináciou bázovej adresy segmentu a offsetu a je reprezentovaná 32-bitovou hodnotou.

Na druhej strane existuje ešte tzv. ***fyzická adresa***, ktorá je tiež udaná 32-bitovou hodnotou, pričom táto hodnota udáva priamo fyzické uloženie dát v pamäti. V reálnom režime vždy platí, že ***fyzická adresa*** a ***lineárna adresa*** sa rovnajú. Preto sa medzi nimi ani nerobí žiaden rozdiel. No v chránenom režime, a teda aj v režime virtuálnom, toto platiť nemusí. Pokiaľ je zapnuté stránkovanie, tak procesor navyše prekladá lineárne adresy na adresy fyzické a tento mechanizmus prepočtu sa nazýva ***stránkovanie***. V prípade zapnutého stránkovania sa vo virtuálnom režime lineárna adresa vypočíta ako v reálnom režime, no ďalej sa prepočíta na fyzickú adresu pomocou mechanizmu uvedeného ďalej.

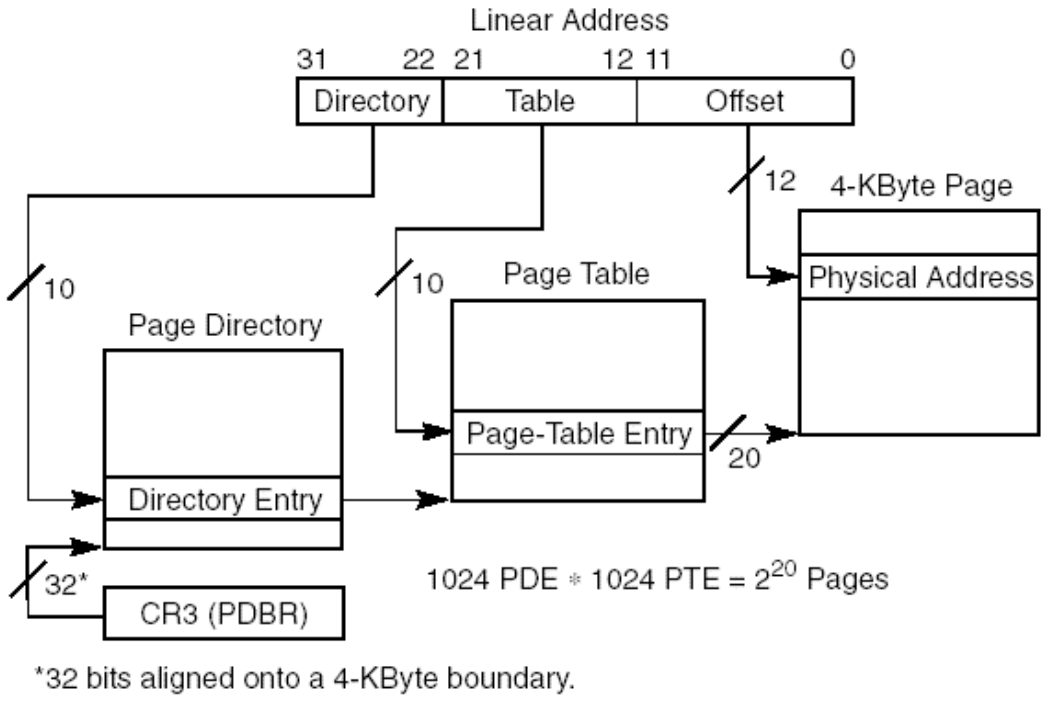
Na nasledujúcom obrázku vidíme princíp prekladu adries pri zapnutom stránkovaní:



Obr. 9 – Preklad adries na procesore 80386 a novších

Z obrázku je vidieť, že získanú lineárnu adresu mechanizmom segmentácie procesor znovu preloží na fyzickú adresu pomocou mechanizmu stránkovania. Toto už je konečná adresa, ktorá fyzicky určuje pamäťovú bunku v počítači. Stránkovanie je implicitne vypnuté, takže v takom prípade sa posledný preklad už neuskutoční a potom naozaj platí, že lineárna adresa priamo udáva pamäťovú bunku v počítači.

Princíp ***stránkovania*** na procesoroch 80386 a novších je znázornený na nasledujúcom obrázku:



Obr. 10 – Princíp stránkovania procesora 80386

Pri zapnutom stránkovaní je lineárna adresa rozdelená na 3 časti, pričom každá časť má svoju pevnú šírku:

- **Adresár** (10 bitov) – udáva index položky v adresári stránok, pričom adresár stránok má 1024 položiek ($2^{10} = 1024$)
- **Podadresár** (10 bitov) – udáva index položky v podadresári stránok, pričom podadresár stránok má tiež 1024 položiek ($2^{10} = 1024$)
- **Offset** (12 bitov) – udáva offset do danej stránky, pričom je v rozsahu 0 – 4095 ($2^{12} = 4096$)

Z uvedenej schémy vyplýva, že procesor jednoduchým rozdelením lineárnej adresy vie rýchlo získať 2 indexy a offset do stránky. Stránka v podaní procesora 80386 vždy musí mať 4 kilobajty, no od Pentia je možná aj schéma bez podadresára stránok, pričom jedna stránka potom musí mať 4 megabajty.

Procesor pri preklade ***lineárnej adresy*** na ***fyzickú adresu*** najprv potrebuje vedieť, kde sa nachádza adresár stránok. Ten je iba jeden a určuje ho riadiaci register procesora ***CR3***. Tento register obsahuje ***fyzickú adresu*** adresára stránok. Tento adresár je iba jeden a môže obsahovať maximálne 1024 položiek. Jeho jednotlivé položky sú 32-bitové hodnoty uložené za sebou v poli, ktoré udávajú okrem iného ***fyzickú adresu*** podadresárov stránok (zarovnanú na 4 kilobajty).

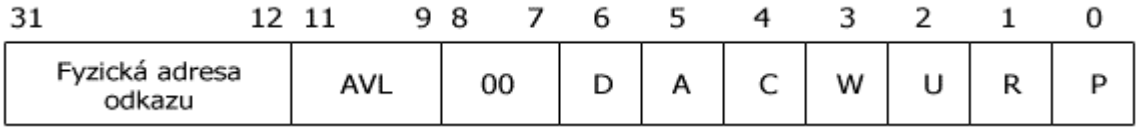
Potom, čo procesor získal pomocou prvého indexu ***fyzickú adresu*** podadresára stránok, získa z tohto podadresára pomocou druhého indexu ďalšiu položku. Položka v podadresári stránok je takisto 32-bitová hodnota, v tomto prípade udávajúca ***fyzickú adresu***

konkrétnej stránky. Podadresárov stránok môže byť maximálne 1024 a každý z nich môže mať maximálne 1024 položiek (stránok). Z toho vyplýva, že maximálny počet 4-kilobajtových stránok môže byť 1.048.576 (= 2^{20}) a týmto spôsobom dokáže stránkovanie pokryť celý adresný priestor procesora 80386, čo sú 4 gigabajty.

V prípade stránkovania Pentia môžeme mať maximálne 1024 stránok o veľkosti 4 megabajty, čo nám tiež pokryje celý adresný priestor. Rozdiel je iba v tom, že neexistuje v tomto prípade podadresár stránok a adresár stránok sa odkazuje rovno na stránku. Offset je samozrejme väčší, 22-bitový.

Poslednú hodnotu ***lineárnej adresy*** tvorí v prípade normálneho stránkovania procesora 80386 12-bitový offset, ktorý vystačí na adresovanie 4 kilobajtov vo vnútri stránky.

Na nasledujúcom obrázku je zobrazenie štruktúry 1 položky adresára a podadresára:



Obr. 11 – Položka adresára a podadresára stránok

Význam jednotlivých bitov je nasledovný:

- **Fyzická adresa odkazu** (20 bitov) – jedná sa o fyzickú adresu buď podadresára alebo stránky, pričom táto adresa je zarovnaná na 4 kilobajty.
- **AVL** (3 bity) – voľné bity pre voľné použitie
- **D** (1 bit) – ***Dirty*** **-** tento bit je nastavený na 1 po každom zápise do stránky
- **A** (1 bit) – ***Accessed*** **-** tento bit je nastavený na 1 po každej read/write operácii v stránke
- **C** (1 bit) – ***Cache*** – určuje, či používať alebo nepoužívať cache pre túto stránku (dostupné od procesoru 80486)
- **W** (1 bit) – ***Write*** – určuje, či používať zápis *write-through (1)*, alebo *write-back (0)* (dostupné od procesoru 80486)
- **U** (1 bit) – ***User/Supervisor*** – určuje, či program s CPL=3 smie použiť stránku *(smie pri 1)*
- **R** (1 bit) – ***Read/Write*** – určuje, či program s CPL=3 smie zapisovať do stránky *(smie pri 1)*
- **P** (1 bit) – ***Present*** – určuje, či je stránka prístupná v pamäti, alebo nie je. Používa sa pri virtualizácii pamäti

Štruktúra položky adresára a podadresára obsahuje mnoho užitočných aj menej užitočných bitov, ktoré sa môžu pri stránkovaní využiť. Za všetky by som menoval predovšetkým bit ***present***, ktorý pokiaľ nie je nastavený na 1 a procesor sa pokúsi s touto položkou pracovať, lebo ju získal z ***lineárnej adresy***, tak sa vyvolá výnimka procesora ***0x0E***, ktorá znamená chybu stránkovania. Procesor zároveň do riadiaceho registra ***CR2*** uloží ***lineárnu adresu***, ktorá túto chybu spôsobila. Na tomto prerušení sa môže nachádzať obslužná rutina stránkovania, ktorá chýbajúcu stránku napr. načíta z disku a vráti riadenie programu, ktorý chybu vyvolal. Týmto spôsobom je možné realizovať virtuálnu pamäť.

3. Spracovanie prerušení v chránenom režime.

3.1 Prerušenia obecne

Veľmi dôležitým prostriedkom procesora je schopnosť spracovávať prerušenia. Prerušenia sa delia podľa druhu na nasledujúce skupiny:

Rozdelenie podľa druhu externého signálu na vývode procesora:

NMI *(Non Mascable Interrupt)* – jedná sa o nemaskovateľné prerušenie, ktoré sa používa v prípade kritických udalostí, ako je napr. výpadok napájania alebo chyba parity operačnej pamäti apod. Tento typ prerušenia nie je možné zakázať softvérovo, teda pomocou vynulovania bitu ***IF*** v príznakovom registri procesora.

INTR *(Mascable Interrupt)* – maskovateľné prerušenie, ktoré používa bežný hardvér počítača, ako napr. diskový radič, zvuková karta, klávesnica atd. Tento typ prerušenia je možné zakázať vynulovaním bitu ***IF***. Zároveň si ale radič prerušenia pamätá svoj stav a prerušenie sa vykoná hneď, ako to bude možné (procesor vyšle signál **INTA** *(Interrupt Acknowledge)*).

Rozdelenie podľa pôvodu signálu prerušenia:

SW_INT (Softvérové prerušenie) – toto prerušenie je synchrónne a vo svojej podstate to ani nie je prerušenie, lebo je vyvolané pomocou inštrukcie procesora. Procesor pri natrafení na inštrukciu prerušenia postupuje rovnako, ako pri spracovaní akéhokoľvek iného prerušenia. Postup bude popísaný ďalej.

HW_INT (Hardvérové prerušenie) – toto prerušenie je asynchrónne, je generované vonkajším hardvérom na základe žiadosti o prerušenie pomocou liniek **IRQ** (Interrupt Request) a môže byť ako maskovateľné, tak aj nemaskovateľné.

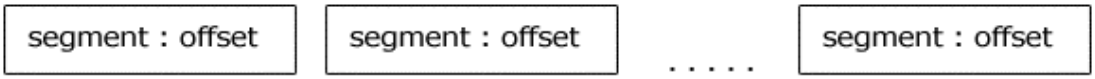
CPU Exception (Výnimka) – toto prerušenie je generované priamo procesorom ako dôsledok vykonania nedovolenej operácie, napr. delenie nulou, neznáma inštrukcia alebo prístup do pamäti mimo dovolený rozsah (v chránenom režime) atď.

Na počítačoch typu **AT** existuje 16 rôznych **IRQ** liniek. Na jednotlivé linky je pridelený jednotlivý hardvér počítača a tieto sú štandardne obsluhované obslužným softvérom týchto zariadení na konkrétnych prerušeniach.

Procesor dokáže spracovať 256 rôznych prerušení, a preto voľné prerušenia obsadzuje rôzny softvér, ako napr. operačný systém, ktorý si na tieto prerušenia zavádza rozhranie pre svoje **API** funkcie, rôzne ovládače apod. Užívateľský program potom zavolá konkrétne prerušenie a tým pádom aj určitú službu operačného systému.

3.2 Spracovanie prerušenia 8086

Teraz popíšem mechanizmus, akým procesor 8086 spracúva prerušenie. Na začiatku pamäti, teda na fyzickej adrese 0, sa nachádza tabuľka vektorov prerušení **IVT** *(Interrupt Vector Table)*. Táto tabuľka obsahuje 256 položiek (pre každé prerušenie jednu) a každá položka má veľkosť 4 bajty. Celá tabuľka je dlhá 1024 bajtov a jednotlivé položky slúžia ako vzdialené ukazovatele do pamäti, teda každá položka tvorí pár **segment:offset**. Tento ukazovateľ je adresou obslužného programu pre konkrétne prerušenie.



Obr. 12 – Schéma tabuľky vektorov prerušení

Po vyvolaní prerušenia akýmkoľvek spôsobom procesor najprv vynásobí číslo tohto prerušenia číslom 4 a získa fyzickú adresu, na ktorej sa nachádza vektor s adresou pre toto konkrétne prerušenie. Následne procesor uloží na zásobník hodnoty registrov **Flags**, **CS** a **IP** (v tomto poradí) a odskočí na adresu obsluhy prerušenia, ktorú si predtým prečítal. Procesor ešte vynuluje príznakové bity **IF** *(Interrupt Flag)* a **TF** *(Trace Flag)*. Od tohto okamihu procesor spracováva inštrukcie obsluhy prerušenia.

Pre ukončenie obsluhy prerušenia procesor obsahuje inštrukciu **IRET**. Keď procesor na túto inštrukciu v kódovom segmente narazí, tak obnoví pôvodné hodnoty **Flags**, **CS** a **IP** zo zásobníka (samozrejme v opačnom poradí) a program potom pokračuje v pôvodnom mieste, kde nastalo prerušenie. Pokiaľ sa jedná o asynchrónne prerušenie, tak obsluha prerušenia musí dať do pôvodného stavu všetky registre procesora, inak by mohlo dôjsť k nepredvídateľným udalostiam alebo aj pádu celého počítača. Výnimku tvorí iba synchrónne prerušenie, pri ktorom sa napr. volajú služby operačného systému.

3.3 Chránený režim

V chránenom režime je situácia o poznanie zložitejšia. Keď dôjde k prerušeniu od externého zdroja, tak procesor sa môže nachádzať v hocijakom stave a teda aj režime. Môžu nastať tieto situácie:

- **obsluha prerušenia je v štýle reálneho režimu a procesor je v reálnom režime**
- **obsluha prerušenia je v štýle chráneného režimu a procesor je v reálnom režime**
- **obsluha prerušenia je v štýle reálneho režimu a procesor je v chránenom režime**
- **obsluha prerušenia je v štýle chráneného režimu a procesor je v chránenom režime**

Pri prerušeniach vyvolaných softvérovo takéto možnosti nenastanú, pretože u nich sa jedná o synchrónne prerušenie a vždy je jasné, v akom režime procesora sa vyskytnú, keďže sa nachádzajú v kóde, ktorý je určený buď pre reálny režim, alebo pre režim chránený. U asynchrónnych prerušení treba zaistiť, aby sa zavolala správna obsluha prerušenia, nech už sa prerušenie vyskytne počas reálneho či chráneného režimu.

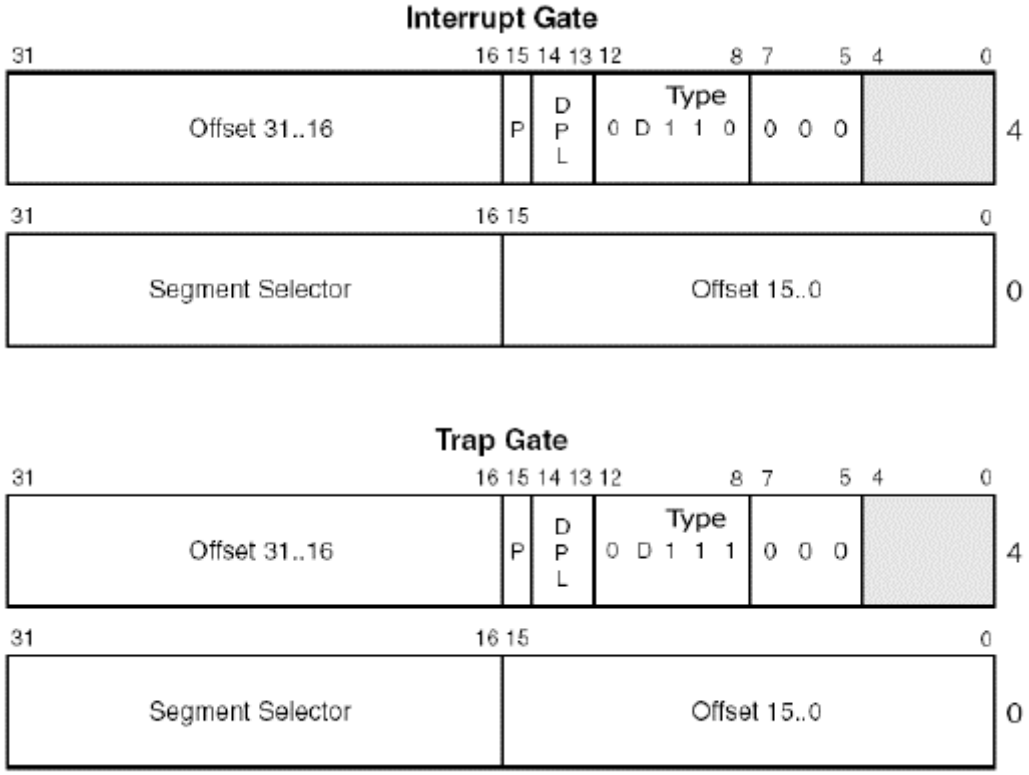
3.4 Brány

U reálneho režimu sa tabuľka vektorov vždy začína na tej istej fyzickej adrese, a to na počiatku pamäti a jej veľkosť je tiež vždy rovnaká. Jedná sa síce o veľmi jednoduchú implementáciu, no z hľadiska flexibility a systému ochrany chráneného režimu je neprijateľná. Tabuľka prerušení obsahuje len adresy obslužných rutín prerušení a nie je v nej miesto na nejaké dodatočné informácie.

Chránený režim nepoužíva tabuľku vektorov. Pre spracovanie prerušenia používa špeciálny typ deskriptora, ktorý sa nazýva ***brána*** *(gate)*. Tieto jednotlivé brány sú tiež zoradené v skupine, tzv. ***IDT*** *(Interrupt Descriptor Table)* – tabuľka deskriptorov prerušení.

Ako už vieme, tak deskriptor je 8-bajtová dátová štruktúra, ktorej priamo rozumie procesor a táto štruktúra je presne špecifikovaná. Podľa typu deskriptora sa jeho štruktúra značne odlišuje.

Na nasledujúcom obrázku je znázornená štruktúra brány:



Obr. 13 – Štruktúra deskriptora brány

Ako je z obrázku vidieť, tak brány obsahujú omnoho viac informácií, ako len adresu rutiny obsluhy prerušenia. Vďaka týmto dodatočným informáciám je spracovanie prerušení v chránenom režime o mnoho komplexnejšie a flexibilnejšie a je možné ho využiť k viacerým účelom, napr. na prepínanie úloh.

Popis štruktúry deskriptora brány:

- **Offset** (32 bitov) – určuje ofsetovú časť adresy rutiny spracovania prerušenia. Offset je rozdelený na 2 časti, horné slovo a dolné slovo
- **Segment Selector** (16 bitov) – určuje segmentovú časť adresy rutiny spracovania prerušenia, konkrétne jeho selektor
- **Type** (4 bity) – typ brány, význam jednotlivých bitov bude popísaný v tabuľke
- **DPL** (2 bity) – Descriptor Privilege Level
- **P** (1 bit) – Segment Present Flag, označuje, či je deskriptor prítomný *(1)*
- **D** (1 bit) – tu je súčasťou typu, označuje, či je brána 16-bitová *(0)* alebo 32-bitová *(1)*

V nasledujúcej tabuľke si uvedieme význam jednotlivých bitov u typu brány:

Typ		Význam
0	0 0 0 0	Zakázaný typ - použitie vyvolá výnimku procesora 0x0D
1	0 0 0 1	TSS procesora 286
2	0 0 1 0	Deskriptor LDT
3	0 0 1 1	TSS procesora 286 (aktívny)
4	0 1 0 0	Brána volania podprogramu 286
5	0 1 0 1	Brána volania úlohy TSS
6	0 1 1 0	Brána volania obslužnej rutiny prerušenia 286, IF=0
7	0 1 1 1	Brána volania obslužnej rutiny prerušenia 286, IF=1
8	1 0 0 0	Zakázaný typ - použitie vyvolá výnimku procesora 0x0D
9	1 0 0 1	TSS procesora 386
10	1 0 1 0	Rezervované
11	1 0 1 1	TSS procesora 386 (aktívny)
12	1 1 0 0	Brána volania podprogramu 386
13	1 1 0 1	Rezervované
14	1 1 1 0	Brána volania obslužnej rutiny prerušenia 386, IF=0
15	1 1 1 1	Brána volania obslužnej rutiny prerušenia 386, IF=1

V našom prípade som uviedol na obrázku 2 prípady brán, a to tzv. ***Interrupt Gate*** a ***Trap Gate***. Interrupt Gate je podľa tabuľky brána volania obslužnej rutiny prerušenia s ***IF=0*** *(Interrupt Flag)* a Trap Gate je podľa tejto tabuľky brána volania obslužnej rutiny prerušenia s ***IF=1***. Pri spracovaní prerušenia na procesore 8086 (alebo v reálnom režime) sa vynuluje Interrupt Flag, aby pri obsluhe prerušenia nedochádzalo k ďalším asynchrónnym prerušeniam od hardvéru. Takisto aj Interrupt Gate vynuluje príznak ***IF***. No niekedy nechceme, aby sa Interrupt Flag vynuloval, a to napríklad v prípade, že nespracovávame asynchrónne prerušenie, ale softvérové prerušenie alebo výnimku procesora. V takomto prípade potom použijeme Trap Gate, ktorá nám príznak ***IF*** nechá zapnutý a tým pádom umožní aj v rámci obsluhy prerušenia vyvolať nové asynchrónne prerušenie.

3.5 Register IDTR

V chránenom režime sa tabuľka vektorov, lepšie povedané tabuľka deskriptorov (brán) nenachádza na dopredu pevne určenom mieste. Môže byť totiž umiestnená hocikde

v operačnej pamäti a tak ju môžeme chrániť napr. pred prepísaním softvérom, ktorý beží v reálnom režime. Procesor ale musí vedieť, kde konkrétne sa táto tabuľka nachádza, a preto obsahuje špeciálny register ***IDTR*** *(Interrupt Descriptor Table Register)*.

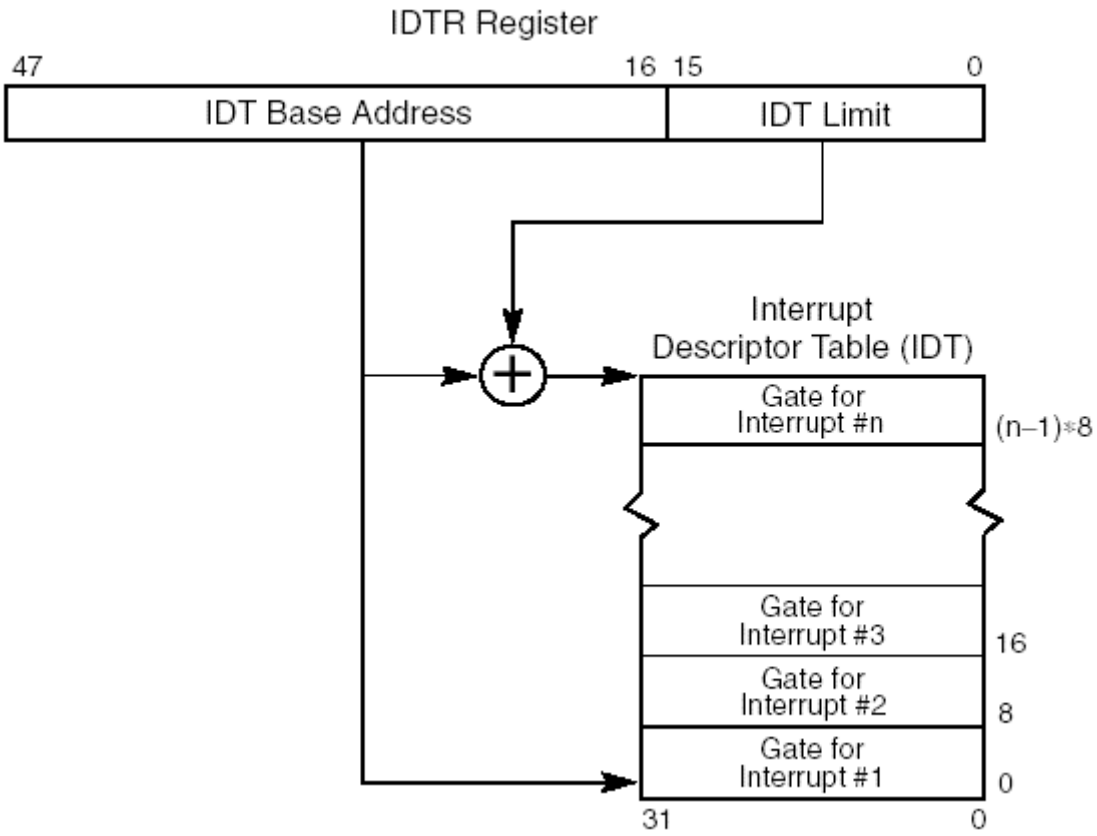
Register ***IDTR*** obsahuje 32-bitovú bázu a 16-bitový limit a na prácu s ním slúžia privilegované inštrukcie ***LIDT*** *(Load Interrupt Descriptor Table)* a ***SIDT*** *(Store Interrupt Descriptor Table)*. O týchto inštrukciách bude viac popísané v kapitole o privilegovaných inštrukciách.

3.6 Spracovanie prerušenia 80386

Pri spracovaní akéhokoľvek prerušenia v chránenom režime procesor postupuje vždy podľa rovnakého algoritmu. Aká akcia nastane, závisí práve od informácií uložených v určitom deskriptore pre konkrétne prerušenie.

Pri výskyte prerušenia procesor vykoná nasledovný algoritmus:

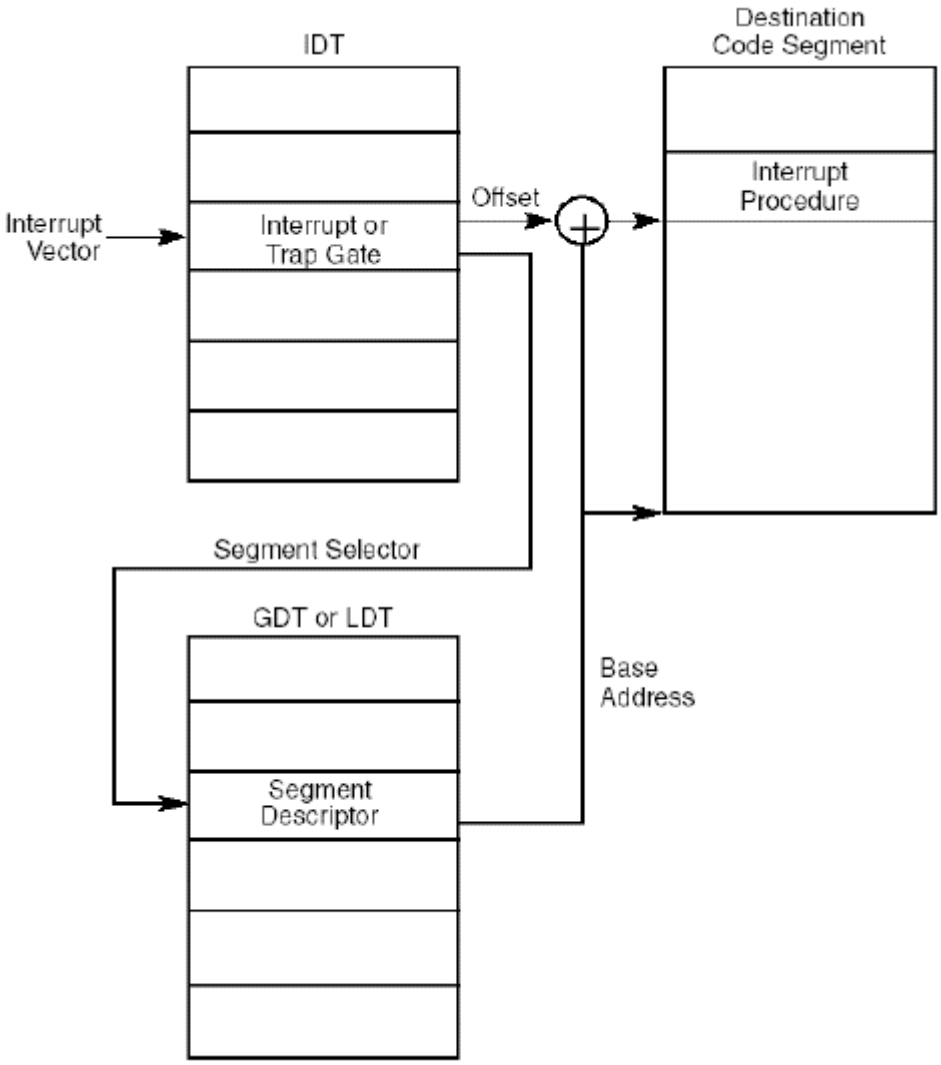
1. Zistí polohu ***IDT***
2. Vypočíta offset pre bránu, ako číslo prerušenia * veľkosť položky (8 bajtov)
3. Zistí, či sme neprekročili limit ***IDT*** uložený v registri ***IDTR***
4. Teraz nasleduje test ochrany práv. Porovná sa ***CPL*** *(Current Privilege Level)*, ktorý sa nachádza v najnižších dvoch bitoch selektora ***CS*** s ***DPL*** *(Descriptor Privilege Level)*, ktorý sa nachádza v deskriptore príslušnej položky v ***IDT***. Pokiaľ ***CPL***>***DPL*** *(numericky)*, tak to znamená, že náš kód nemá dostatočné oprávnenie vyvolať dané prerušenie a to vyvolá výnimku procesora 0x0D. Tento test sa vynecháva u asynchrónne volaných prerušení a výnimiek procesora.
5. Teraz sa porovná ***CPL*** nášho volajúceho kódu a ***CPL*** obslužnej rutiny prerušenia. Pokiaľ sa nerovnajú, tak to znamená, že nebežia na rovnakej úrovni oprávnenia a preto procesor prepne zásobník pomocou ***TSS*** *(Task State Segment)*, aby nový zásobník spĺňal prioritu obsluhy prerušenia. Zároveň sa na nový zásobník uloží ***(E)SP*** a ***SS*** volajúceho.
6. Na zásobník sa teraz uloží ***(E)FLAGS***, ***(E)IP*** a ***CS*** volajúceho.
7. ***TF*** *(Trace Flag)* sa vynuluje. To spôsobí, že pokiaľ krokujeme náš kód v debuggeri, tak softvérové prerušenia v našom kóde, ktoré väčšinou volajú API funkcie, budú vykonané celé odrazu. Podľa typu brány sa prípadne ešte vynuluje aj ***IF*** *(Interrupt Flag)*, čo spôsobí odstavenie asynchrónnych prerušení.
8. V tomto momente procesor odovzdá riadenie na adresu ***selektor:offset*** udanú v bráne, kde začína kód obsluhy spracovania prerušenia.



Obr. 14 – Grafické znázornenie výpočtu polohy a platnosti položky IDT

Algoritmus tu popísaný je plne vyčerpávajúci a popisuje každý krok procesora, ktorý musí vykonať, aby mohlo byť spracované prerušenie. Toto platí, pokiaľ pre príslušné prerušenie je v tabuľke deskriptor typu brány, ktorého úlohou je zavolať obsluhu pre toto prerušenie. Namiesto brány ale môže tiež obsahovať selektor ***TSS*** *(Task State Segment),* čo je špeciálna dátová štruktúra slúžiaca k hardvérovej podpore multitaskingu. Potom možno pomocou prerušenia hardvérovo prepnúť úlohy a procesor sa o uloženie a načítanie svojho kontextu pre tieto jednotlivé úlohy postará sám.

Na nasledujúcom obrázku je graficky znázornené, ako prebieha spracovanie prerušenia, pokiaľ je deskriptor v IDT typu brány:



Obr. 15 – Grafické znázornenie spracovania prerušenia

Po vykonaní obsluhy prerušenia kód obsahuje inštrukciu ***IRET*** *(Interrupt return)*, ktorá zabezpečuje návrat do pôvodného kódu programu, ktorý buď prerušenie vyvolal, alebo bol sám prerušený asynchrónnym prerušením. Po návrate z obsluhy prerušenia pôvodný program pokračuje ďalej vo svojej činnosti. Preto inštrukcia ***IRET*** musí zabezpečiť obnovenie pôvodných hodnôt ***(E)FLAGS***, ***(E)IP*** a ***CS*** zo zásobníka a pokiaľ procesor zistí zmenu úrovne oprávnenia medzi volajúcim kódom a obsluhou prerušenia, tak musí obnoviť ešte aj starý zásobník. To znamená, že procesor obnoví z aktuálneho zásobníka hodnoty registrov ***(E)SP*** a ***SS*** a tým pádom sa obnoví starý zásobník volajúceho kódu, ktorý spĺňa jeho požiadavky na oprávnenie.

4. Volanie funkcií BIOSu a DOSu z chráneného režimu.

4.1 Funkcie a procedúry

Pod pojmom funkcia alebo procedúra rozumieme časť kódu vo vnútri väčšieho programu, ktorá vykonáva určitú špecifickú úlohu a môže byť relatívne nezávislá od ostatného kódu. V praxi je funkcia alebo procedúra najprv zavolaná niektorou z týchto inštrukcií:

- **call**
- **call far**
- **int**

V strojovom kóde takáto funkcia existuje ako skupina inštrukcií, ktoré sú zakončené niektorou z týchto inštrukcií špeciálneho významu:

- **ret**
- **retf**
- **iret**

Procesor spracúva kód programu a keď natrafí na niektorú z inštrukcií ***call***, ***call far*** alebo ***int***, tak vyvolá podprogram, ktorý sa nachádza na určitej adrese. Táto adresa sa vyskytuje buď priamo ako operand danej inštrukcie, alebo sa nachádza v pamäti, na ktorú opäť ukazuje operand inštrukcie, alebo sa nachádza v pamäti, napr. ako súčasť tabuľky vektorov prerušenia.

Po vykonaní volajúcej inštrukcie procesor na zásobník uloží prinajmenšom návratovú adresu, na ktorú sa procesor vracia po ukončení funkcie, teda po vykonaní zodpovedajúcej inštrukcie ***ret***, ***retf*** alebo ***iret***.

Za konzistenciu inštrukcií je zodpovedný tvorca programu, teda buď priamo programátor, alebo prekladač, ktorý kód vygeneroval. Ak nie je dodržaná párová konzistencia inštrukcií – napr. ***call – ret***, ***int – iret*** apod., tak s najväčšou pravdepodobnosťou dôjde k pádu celého programu, v reálnom režime aj k pádu celého systému.

4.2 Inštrukcie *call - ret*

Inštrukčný pár *call – ret* sa používa na obsluhu funkcií v rámci jedného kódového segmentu. Operand inštrukcie *call* je 16-bitové alebo 32-bitové číslo, ktoré udáva relatívnu adresu voči aktuálnej hodnote čítača inštrukcií IP; offset procedúry získa procesor tak, že operand inštrukcie call pripočíta k IP. Inštrukcia *call* ukladá na zásobník offset nasledujúcej inštrukcie po *call*, tzv. blízky ukazovateľ *(near pointer)*.

Inštrukcia *ret* odoberie zo zásobníka tento blízky ukazovateľ, ktorý môže byť 16-bitový alebo 32-bitový v závislosti od zvolenej dĺžky offsetu v kódovom segmente a vykoná návrat na adresu danú týmto ukazovateľom. Inštrukcia *ret* môže obsahovať tiež parameter, ktorý udáva, koľko bajtov má procesor odobrať zo zásobníka po opustení funkcie.

Sú rôzne variácie inštrukcie *call*, ako je priame a nepriame adresovanie, to je dané vysokou flexibilitou architektúry x86.

4.3 Inštrukcie *call far – retf*

Inštrukčný pár *call far – retf* má podobné použitie ako pár *call – ret*, s jediným rozdielom, a to, že tieto inštrukcie pracujú s úplnou adresou, nie relatívnou voči čítaču inštrukcií.

Inštrukcia *call far* obsahuje ako argument adresu zloženú z páru ***segment:offset***, pričom segment je vždy 16-bitový a offset môže byť ako 16-bitový, tak aj 32-bitový, opäť v závislosti od aktuálneho prostredia.

Inštrukcia *retf* sa taktiež líši od inštrukcie *ret* iba v tom, že navracia riadenie procesora na kód pomocou úplnej adresy, tzv. ďaleký ukazovateľ *(far pointer)*.

Obe tieto inštrukcie sa používajú pri volaní vzdialených funkcií, teda funkcií, ktoré sa nachádzajú v inom kódovom segmente. Samozrejme možno pomocou nich volať funkcie aj v rovnakom kódovom segmente, ale nedáva to veľký zmysel.

4.4 Inštrukcie *int – iret*

Inštrukčný pár *int – iret* sme si už predstavili v kapitole o spracovaní prerušení. Netreba ho preto bližšie popisovať, no pre úplnosť som ho sem zaradil. Aj spracovanie prerušenia je istým spôsobom volanie podprogramu, teda nejakej funkcie alebo procedúry, ktorá môže vracať návratovú hodnotu.

4.5 Volania podprogramov medzi režimami

Predstavme si, že sme 16-bitová alebo 32-bitová aplikácia bežiaca v chránenom režime a potrebujeme zavolať 16-bitovú funkciu, ktorá je ale napísaná pre reálny režim a teda používa 16-bitové adresovanie pamäti, zapisuje do kódového segmentu atď.

Alebo ešte horšie, predstavme si, že sme 16-bitový program bežiaci pod DOSom v reálnom režime a potrebujeme zavolať 16-bitový alebo 32-bitový kód, ktorý je napísaný pre chránený režim a dokonca sa nachádza niekde v 30-tom megabajte operačnej pamäti a náš spôsob adresovania stačí pokryť akurát tak prvý megabajt.

Z toho všetkého vyplýva, že nie je možné z chráneného režimu zavolať podprogram, ktorý je napísaný v štýle reálneho režimu a použiť na to jednoducho inštrukcie ***call – ret*** alebo ***call far – retf***. Inštrukcie ***int – iret*** je možné použiť, ale vyžaduje si to istú réžiu v pozadí týchto inštrukcií.

Aby sme sa mohli pohybovať medzi jednotlivými režimami, tak prinajmenšom musíme procesor prepnúť z jedného do druhého. Nie je možné, aby procesor bol napr. v chránenom režime a začal spracovávať inštrukcie z reálneho režimu, to by celkom isto viedlo po chvíli k pádu programu alebo aj systému. Toto prepnutie procesora musí byť naprogramované v assembleri, pretože vyžaduje použitie špeciálnych (privilegovaných) inštrukcií a presný postup. Tieto časti kódu, ktoré sú zodpovedné za prepínanie režimov procesora sú veľmi náchylné na chyby a tieto chyby sa ťažko hľadajú.

Samozrejme, prepínanie režimov procesora a všetku réžiu, ktorá je na to potrebná, môžeme vykonávať my sami, alebo to necháme na inom softvéri, ako je už spomínaný ***extender***, ktorý nám poskytuje mnoho ***API*** funkcií na to potrebných.

4.6 Prepnutie režimu procesora

Ako príklad uvediem jeden podprogram (funkciu), ktorý je napísaný v štýle reálneho módu a ktorý chceme zavolať z kódu, ktorý je napísaný v štýle chráneného režimu.

Funkcia:

```
mov bx, 0x100
mov cs:[bx], ax
add ax, 2
ret
```

Náš kód:

```
mov ecx, 4
xor eax, eax
call Funkcia
add ecx, eax
```

Náš kód je časť programu, ktorý je 32-bitový, beží v chránenom režime a pomocou inštrukcie *call* chce zavolať funkciu, ktorá je 16-bitová, napísaná v štýle reálneho módu a aj keby bolo zavolanie tejto funkcie úspešné, tak by prinajmenšom vyvolala výnimku procesora kvôli inštrukcii, ktorá zapisuje do kódového segmentu, čo je v chránenom režime neprípustné.

Ďalším problémom je spôsob fungovania inštrukcií ***call – ret***. Pretože sa nachádzame v 32-bitovom chránenom režime, inštrukcia *call* uloží na zásobník relatívnu adresu, ktorá je tiež 32-bitová a inštrukcia *ret* obnovuje a odstraňuje túto 32-bitovú hodnotu zo zásobníku preč. No v 16-bitovom reálnom móde inštrukcie *call* a *ret* pracujú iba s 16-bitovou relatívnou adresou, a preto sú tieto 2 adresy navzájom nekompatibilné.

Riešenie tejto situácie môže byť nasledovné:

Náš kód:

```
mov ecx, 4
xor eax, eax
call prepni_RM
call Funkcia
call prepni_PM
add ecx, eax
```

V tomto prípade sme pred volanie našej funkcie umiestnili volanie podprogramu ***prepni_RM***, ktorý prepne procesor do reálneho režimu a za volanie našej funkcie sme umiestnili volanie podprogramu ***prepni_PM***, ktorý prepne procesor zasa späť do chráneného režimu. Toto je najjednoduchší spôsob, akým zavolať z chráneného režimu funkciu, ktorá je písaná pre reálny režim.

Aby to celé fungovalo, tak náš 32-bitový kód sa musí nachádzať v dolnej pamäti, teda v prvom megabajte, aby bol prístupný aj z reálneho režimu. Teraz si vysvetlíme, čo urobia jednotlivé procedúry:

prepni_RM:

- výpočet adresy pre kódový segment v štýle reálneho módu
- prepnutie režimu procesoru nastavením bitu ***PE*** = 0
- inicializácia segmentových registrov (limit = 64 KB)
- inicializácia kódového segmentu, ktorý je „aliasom“ ku starému ***CS***
- vzdialený skok do nového kódového segmentu
- (úprava návratovej adresy z 32-bitovej na 16-bitovú)

prepni_PM:

- výpočet adresy pre kódový segment v štýle chráneného módu
- prepnutie režimu procesoru nastavením bitu ***PE*** = 1
- inicializácia segmentových registrov (napr. na neplatný selektor)
- inicializácia kódového segmentu, ktorý je „aliasom“ ku starému ***CS***
- vzdialený skok do nového kódového segmentu
- (úprava návratovej adresy z 16-bitovej na 32-bitovú)

Vyzerá to trochu chaoticky, no snažím sa priblížiť, prečo to všetko treba urobiť. Náš 32-bitový kód je spustený pod chráneným režimom a beží. Procesor natrafí na volanie podprogramu ***prepni_RM***, ktorý sa zavolá a jeho kód je spočiatku tiež 32-bitový. Návratová adresa je 32-bitová. Najprv si vypočítame adresu kódového segmentu pre reálny režim podľa jednoduchého vzorca:

segmentová časť adresy = báza segmentu / 16

Dôležité je upozorniť, že báza segmentu pripadá do prvého megabajtu pamäti a tiež, že náš 32-bitový kód sa nachádza v prvých 64 kilobajtoch tohto segmentu. Keby tomu tak nebolo, nemáme možnosť adresovať tento úsek kódu z reálneho režimu.

Ďalej *báza segmentu*, ktorá určuje, na ktorej lineárnej adrese sa nachádza začiatok 32-bitového kódového segmentu, musí byť násobok ***16***. Keby táto báza nebola násobkom 16, došlo by k posuvu relatívnej (ofsetovej) časti adresy v rámci segmentu, a to je v našom prípade neprípustné.

Analogicky v opačnom prípade pri prechode z reálneho režimu do režimu chráneného v podprograme ***Prepni_PM*** si vypočítame bázu segmentu podľa vzorca:

báza segmentu = segmentová časť adresy * 16

Touto bázou segmentu potom vyplníme príslušnú položku v deskriptore zodpovedajúcom novému kódovému segmentu a jeho limit nastavíme na 64 kilobajtov, čím vlastne vytvoríme *alias* segment starého kódového segmentu.

V ďalšom kroku procedúry ***prepni_RM*** a ***prepni_PM*** nastavujú ***PE*** *(Protection Enable)* bit procesora. Tento bit procesoru hovorí, či procesor prevádzkuje chránený, resp. virtuálny

režim, alebo či beží pod reálnym režimom. Hneď po jeho nastavení na 1 alebo na 0 sa procesor začne chovať podľa nových pravidiel, teda podľa pravidiel jedného z týchto režimov. Preto sa musí pred prepnutím tohto bitu všetko potrebné pripraviť a po jeho zmene sa kód musí chovať korektne v súlade s novým režimom.

Bit ***PE*** sa nachádza v 32-bitovom riadiacom registri ***CR0*** procesora 80386 a novších, ktorý je priamo prístupný pomocou inštrukcie *mov*, ktorá umožňuje presúvať jeho obsah do ostatných 32-bitových registrov (napr. univerzálne registre). V týchto registroch už nie je problém s jednotlivými bitmi manipulovať.

Register ***CR0*** je síce novinkou procesora 80386, ale jeho spodných 16 bitov v rovnakej štruktúre obsahoval aj procesor 80286, kde sa tento register nazýval ***MSW*** *(Machine Status Word)* a pristupuje sa k nemu pomocou inštrukcií ***lmsw*** a ***smsw***, o ktorých je viac popísané v kapitole *Privilegované inštrukcie*.

V nasledujúcej tabuľke je zobrazená štruktúra registra ***CR0***:

Bit	Označenie	Význam	Popis
31	PG	Paging	Tento bit, ak je nastavený na 1, zapína stránkovanie
30	CD	Cache disable	Pri 1 zakazuje použitie vnútornej cache pamäti 486
29	NW	Non-write through	
18	AM	Aligment mask	Kontrola zarovnávania programu pri CPL=3, ak AM=1 a AC=1 (EFLAGS)
16	WP	Write protect	
5	NE	Numeric error	Chyby koprocesoru sa spracúvajú na INT 0x10 (1) alebo na INT 0x0D (0)
4	ET	Extension type	
3	TS	Task Switched	Procesor nastavuje tento bit na 1 po každom prepnutí úloh (zmena TR)
2	EM	Emulation	Ak je tento bit nastavený na 1, potom je prítomný emulátor koprocesoru
1	MP	Math present	Ak je tento bit nastavený na 1, potom je prítomný koprocesor
0	PE	Protection Enable	Pri 1 procesor prevádzkuje chránený alebo virtuálny režim, inak reálny

Štruktúra riadiaceho registra CR0
[zdroj 4]

Prepnutie bitu ***PE*** je možné vykonať nasledovne:

Nastavenie bitu *PE*:

```
mov eax,cr0
or al,1
mov cr0,eax
```

Zrušenie bitu *PE*:

```
mov eax,cr0
and al,0xFE
mov cr0,eax
```

Bit ***PE*** je najnižší bit v riadiacom registri. Musíme si uvedomiť, že keď pracujeme s jednotlivými bitmi, tak nesmieme ovplyvniť bity ostatné. Preto pri jeho zmene používame

inštrukcie ***or*** a ***and***, ktoré pri správnom použití nám zabezpečia, že nastavíme na 1 (pri logickom súčte) alebo na 0 (pri logickom súčine) iba tie bity, ktoré naozaj chceme.

Obsah riadiaceho registra ***CR0*** najprv presunieme do registra ***eax***, na ktorom nastavíme bit ***PE*** a opäť presunieme do registra ***CR0***. Hneď po zmene bitu v riadiacom registri sa procesor začne správať podľa pravidiel nového režimu:

Bit *PE*: **0** – procesor prevádzkuje ***reálny režim***
1 – procesor prevádzkuje ***chránený (resp. virtuálny) režim***

Po nastavení bitu ***PE*** prichádza na rad inicializácia segmentových registrov. Je to dôležité, lebo momentálne sú nastavené na starý režim a nový režim k nim pristupuje podľa úplne iných pravidiel. Segmentové registre sa aj po zmene režimu chovajú „akoby“ ešte patrili do starého režimu, pokiaľ sa nepokúsime o ich zmenu. Tým mám na mysli možnosť s nimi pracovať rovnako, ako pred prepnutím bitu ***PE***. Lenže vo chvíli, keď sa pokúsime o ich zmenu, tak tá je už vykonaná v zmysle nového režimu.

Preto je dobrým zvykom inicializovať všetky segmentové registre hneď po prepnutí režimu a žiaden nenechať neinicializovaný, aj keď ho momentálne nepotrebujeme. V prípade podprogramu ***prepni_RM*** si vypočítame nové hodnoty pre segmentové registre pre reálny mód a v prípade ***prepni_PM*** si vypočítame nové hodnoty segmentových registrov pre chránený mód podľa rovnakého vzorca, ktorý je použitý pre kódový segment.

Limit pre segmenty reálneho režimu nastavíme na 64 kilobajtov, aby bol tento režim kompatibilný s režimom práce procesora 8086. Je ale pravdou, že to nie je podmienka a je možnosť ponechať limit hoci aj 4 gigabajty. Niektorý softvér to využíva a potom môže adresovať aj z reálneho režimu celú operačnú pamäť procesora 80386 pomocou 32-bitových registrov. Takýto režim sa zvykne nazývať ***Unreal mode*** alebo ***Flat real mode***, ale nejedná sa o programátorsky čistú techniku.

Limit pre segmenty chráneného režimu tiež môžeme nastaviť na 64 kilobajtov, keďže vytvárame *alias* segmenty k segmentom reálneho režimu.

Segmentové registre, ktoré momentálne nepotrebujeme, môžeme v chránenom režime nastaviť na hodnotu *neplatný selektor*, čo v praxi znamená hodnotu 0. V tomto prípade každý odkaz na tento segment vyvolá výnimku procesora a nás nemusí trápiť, aké hodnoty týmto registrom priradiť.

V nasledujúcom prípade hodnoty _DSeg, _ESeg, _SSeg, _FSeg a _GSeg obsahujú platné zmysluplné hodnoty segmentových registrov pre kód bežiaci v reálnom režime a hodnoty _DSel, _ESel a _SSel obsahujú platné zmysluplné hodnoty segmentových registrov pre kód bežiaci v chránenom režime.

Príklad nastavenia segmentových registrov pre reálny režim:

```
mov ds,cs:[_DSeg]
mov es,cs:[_ESeg]
mov ss,cs:[_SSeg]
mov fs,cs:[_FSeg]
mov gs,cs:[_GSeg]
```

Príklad nastavenia segmentových registrov pre chránený režim:

```
        mov ds,cs:[_DSel]
        mov es,cs:[_ESel]
        mov ss,cs:[_SSel]
        xor ax,ax                   ;AX = 0
        mov fs,ax                   ;neplatný selektor
        mov gs,ax                   ;neplatný selektor
```

Už máme inicializované všetky segmentové registre, ale kódovému segmentu nemôžeme priradiť nejakú hodnotu rovnakým spôsobom, ako ostatným registrom. Inicializujeme ho iným spôsobom, a to ***vzdialeným skokom***. Vzdialený skok je skok do iného segmentu, pričom pre jeho vykonanie musíme dať procesoru k dispozícii úplnú adresu, teda jeho segmentovú aj ofsetovú časť.

Vykonáme ho takto:

Vzdialený skok:

```
        ...
        mov ax,1
        jmp far _CSeg:navestie
  navestie:
        or bx,2
        ...
```

Vzdialený skok pozostáva z inštrukcie ***jmp far***, ktorá obsahuje operand ***_CSeg : navestie***. Tento operand udáva úplnú adresu pre skok. ***_CSeg*** obsahuje buď hodnotu nového kódového segmentu pre reálny režim (v prípade procedúry **prepni_RM**), alebo obsahuje hodnotu selektora, ktorý odkazuje na deskriptor nového kódového *alias* segmentu (v prípade procedúry **prepni_PM**).

Navestie je ofsetová časť adresy, ktorá obsahuje hodnotu pre čítač inštrukcií po vykonaní skoku a odkazuje sa na ďalšiu inštrukciu v kóde. V našom prípade sa teda vykoná inštrukcia ***mov ax,1*** v pôvodnom kódovom segmente starého režimu a hneď za ňou sa vykoná vzdialený skok do nového kódového segmentu nového režimu, ktorý je zároveň *alias* segmentom ku starému kódovému segmentu. Ďalšia inštrukcia ***or bx,2*** sa teda vykoná už v novom kódovom segmente patriacom novému pracovnému režimu procesora.

Chránený režim procesora 80386 a vyšších obsahuje podporu ako pre 16-bitové kódové segmenty, tak aj pre 32-bitové kódové segmenty. Preto pri vzdialených skokoch pri zmene režimu je potrebné si dávať pozor, v koľko bitovom kódovom segmente náš kód beží (buď je to dané prekladačom, alebo si to programátor sám explicitne určí napr. v assembleri).

Dôležité je to z hľadiska veľkosti offsetovej časti adresy, ktorá je podľa toho tiež buď 16-bitová, alebo 32-bitová. Pokiaľ vykonávame vzdialený skok do 16-bitového kódového segmentu, tak musíme použiť 16-bitový offset a naopak. Je zrejmé, že 16-bitový kódový segment môže mať maximálne dĺžku 64 kilobajtov.

4.7 DOS a BIOS

DOS *(Disk Operating System)* je dnes už zastaraný 16-bitový operačný systém, ktorý sa využíval na osobných počítačoch prevažne v 90-tych rokoch minulého storočia. Je schopný práce už od procesora 8086 a pre svoju činnosť využíva 1 megabajt pamäti, aj keď pomocou špeciálnych ovládačov dokáže nepriamo využiť celú operačnú pamäť – na novších procesoroch. Tieto ovládače ku svojej činnosti niekedy využívajú tiež chránený režim (napr. EMM386.EXE).

Jadro DOSu je čisto 16-bitové, je realizované ako monolitický kernel, ktorý sa pri načítaní systému rezidentne ukladá do dolnej časti pamäti (bez dodatočných ovládačov). Komunikácia s užívateľom prebieha na úrovni príkazového riadku, ktorý má pomerne jednoduchú syntax a po nainštalovaní dodatočného softvéru je možné túto komunikáciu ešte značne uľahčiť.

BIOS *(Basic Input / Output System)* je základný vstupno-výstupný systém, ktorý je tvorený programom umiestneným v pevnej pamäti *(napr. ROM)* a nachádza sa v každom počítači. BIOS sprostredkúva základnú komunikáciu medzi hardvérom a softvérom počítača a bez jeho pomoci by nebolo možné hardvér na počítači vôbec riadiť, tzn. nebolo by možné zobrazovať znaky na monitore, čítať znaky z klávesnice, zapisovať údaje na pevný disk atď.

BIOS je zároveň prvým programom, ktorý vykoná procesor po spustení počítača. Vykoná tzv. ***POST*** *(Power-On Self Test)*, čo znamená základný test hardvéru počítača a pripojených komponentov, ako je napr. integrita pamäti alebo funkčnosť klávesnice. Potom BIOS načíta do operačnej pamäti na konkrétnu adresu *(0000:7C00)* z určitého (nastaveného) média bootovací sektor, ktorý je hneď prvým sektorom na médiu a predá tomuto úseku kódu v pamäti riadenie procesora. Kód v bootovacom sektore sa postará o načítanie operačného systému.

4.8 Funkcie API

Pri programovaní pod nejakým operačným systémom vždy potrebujeme mať prístup k systémovým prostriedkom daného systému, aby mohol náš program s ním komunikovať, aby sme mohli zadávať nášmu programu vstupy, aby sme mohli z neho čítať výstupy atď. Na toto všetko obsahuje operačný systém ***API funkcie*** *(Application programming interface)*.

Tieto API funkcie buď môžeme volať priamo, alebo pokiaľ programujeme vo vyššom programovacom jazyku, tak väčšinou API funkcie priamo nevoláme, ale využijeme jeho knižničné funkcie k podobným účelom. Po preložení zdrojového kódu prekladačom a prilinkovaní potrebných knižníc tieto knižnice v konečnom dôsledku tiež volajú funkcie API (nie vždy, ale v mnohých prípadoch).

Funkcie API môžu byť implementované rôznym spôsobom, napr. v operačnom systéme Microsoft Windows sa nachádzajú v dynamických systémových knižniciach ***DLL*** *(Dynamic-Link Library)*, ktoré sa pripájajú k programu až pri jeho spustení. O tomto spôsobe sa ale baviť nebudeme.

DOS a ***BIOS*** má prístup k API funkciám riešený iným spôsobom. DOS po svojom naštartovaní obsadí niektoré prerušenia, na ktoré umiestni svoje obslužné programy. Keď niektorá aplikácia vyvolá softvérovo toto prerušenie, tak obslužný program operačného systému dostane riadenie a podľa dohodnutých hodnôt registrov procesora pozná, čo má vykonať. To isté platí aj pre BIOS, akurát s tým rozdielom, že prerušenia, ktoré obsluhuje

BIOS, sú správne nastavené skôr, ako riadenie dostane kód, ktorý sa načíta z bootovacieho sektoru. API funkcie BIOSu sú prístupné v podstate od spustenia počítača.

Ako príklad uvediem v nasledujúcej tabuľke zopár API funkcií DOSu:

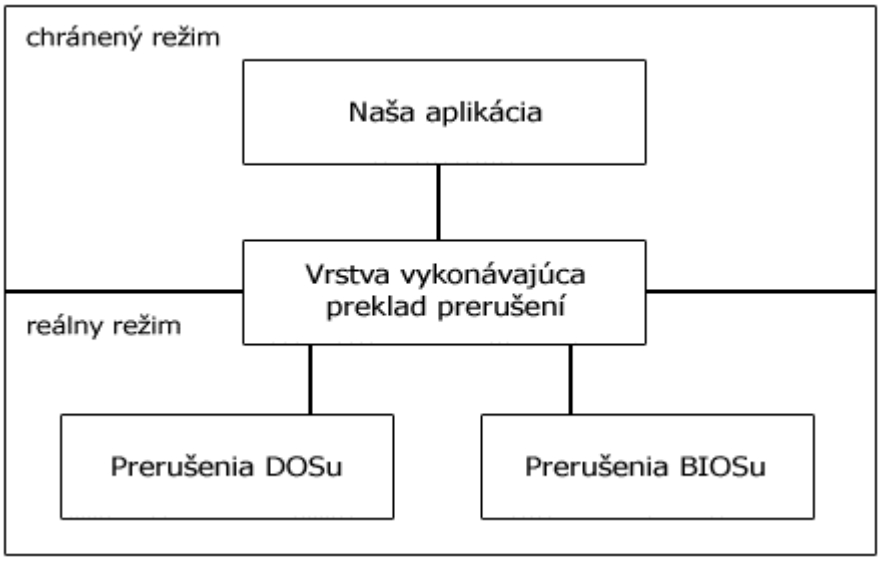
Funkcia	Parametre	Návratová hodnota	Služba
AH=0x0		neexistuje	Ukonči program
AH=0x1		AL=načítaný znak	Čítaj znak zo štandardného vstupu
AH=0x2	DL=znak		Zapíš znak na štandardný výstup
AH=0x9	DS:DX=ukazovateľ na reťazec ukončený '$'		Vypíš reťazec

Ukážka zopár API funkcií DOSu

Tieto API funkcie sú prístupné na prerušení ***0x21***.

Pri programovaní aplikácie určenej pre chránený režim potrebujeme využívať práve aj tieto API funkcie. Vzhľadom na to, že ***DOS*** aj ***BIOS*** obsahujú 16-bitový kód, tak musíme vytvoriť určitú vrstvu, akýsi preklad volaní prerušení medzi chráneným a reálnym režimom a naopak.

Na nasledujúcom obrázku je zobrazená schéma prekladu prerušení:



Obr. 16 – Schéma prekladu prerušení

Pri prechode z reálneho režimu do režimu chráneného nie je problém s hodnotami uloženými v registroch. Tie si procesor pamätá, dokiaľ mu ich sami nezmeníme. Problém nastáva hlavne pri odovzdávaní parametrov v pamäti a o to hlavne sa musíme pri preklade postarať. Program napísaný pre chránený režim sa môže nachádzať (a aj sa väčšinou nachádza) v pamäti nad 1. megabajtom, a preto nie je adresovateľný z reálneho režimu. Takisto tento program môže mať

omnoho väčšiu veľkosť ako len 1 megabajt, a preto by sa do pamäti adresovateľnej reálnym režimom ani nevošiel.

Pri preklade prerušenia našťastie vieme, ktoré hodnoty registrov čo znamenajú. Pokiaľ si zoberieme API funkcie DOSu a BIOSu, tak tieto sú dobre známe a popísané v rôznych manuáloch. Majú svoju pevnú štruktúru a tú aj dodržujú. Niektoré hodnoty registrov obsahujú len čísla, ako napr. rôzne počty, znaky, číslo API funkcie atď. Niektoré iné hodnoty registrov ale môžu obsahovať aj adresy, väčšinou úplné, teda obsahujú dvojicu ***segment:offset***. A tu už nastáva problém, že adresy, ktoré napr. odkazujú na nejaké údajové štruktúry, nie sú v rámci rôznych režimov kompatibilné.

Pokiaľ je to možné, stačí pri volaní prerušenia, ktoré patrí reálnemu režimu (teda DOSu alebo BIOSu), vypočítať len hodnotu novej adresy tak, aby zodpovedala adresovaniu nového režimu. Toto je sa dá iba v prípade, že adresa, ktorá zodpovedá chránenému režimu, teda páru ***selektor:offset***, sa nachádza v 1. megabajte fyzickej pamäti a to sme si povedali, že väčšinou neplatí. Ďalšou podmienkou je, že dátová štruktúra, na ktorú adresa ukazuje, sa musí vojsť do 64 kilobajtov. Pokiaľ by sme mali pole dlhé napr. 100 000 bajtov, tak potom nemáme možnosť toto pole súvislo adresovať v reálnom režime, pretože offset adresy je iba 16-bitový.

V ostatných prípadoch, keď nestačí len upraviť adresu pri prechode medzi režimami, musíme fyzicky prekopírovať celú údajovú štruktúru, na ktorú nám adresa ukazuje. Nezaujíma nás obsah tejto štruktúry, ale iba jej veľkosť. Pri prechode z chráneného režimu do reálneho si musíme pripraviť alebo mať celý čas k dispozícii buffer o určitej rozumnej veľkosti, ktorý sa nachádza v konvenčnej pamäti, teda v pamäti pod 1. megabajtom. Tento buffer je adresovateľný ako z reálneho režimu, tak aj z chráneného režimu.

Potom naša rutina, ktorá vykonáva prekladanie prerušenia, pokiaľ je zavolaná z chráneného režimu (pomocou prerušenia), tak v prípade potreby prečíta adresu, ktorá ukazuje na údajovú štruktúru. Táto adresa je v tvare ***selektor:offset*** a odkazuje niekam do pamäti v rámci 4 gigabajtov, ktoré je schopný chránený režim adresovať. Ďalej musíme poznať veľkosť tejto údajovej štruktúry, ale tú poznáme buď podľa špecifikácie konkrétnej API funkcie, alebo podľa hodnoty niektorého z registrov (často ***CX***). Následne nato prekopírujeme obsah pamäti z chráneného režimu do nášho bufferu, ktorý máme pripravený a ku ktorému momentálne pristupujeme z chráneného režimu. Potom prepneme režim na reálny a možnosť adresovania našej aplikácie sa stratila. No máme všetko potrebné pripravené a môžeme zavolať skutočnú API funkciu, s rovnakými parametrami, akurát namiesto pôvodnej adresy ukazujúcej na údajovú štruktúru v chránenom režime jej odovzdáme ako parameter adresu buffera, ktorý sa nachádza v dolnej pamäti, takže s tým nie je žiaden problém. Ešte by som poznamenal, že samozrejme API funkcii odovzdáme adresu buffera v štýle reálneho režimu, keďže k tomuto bufferu pristupujeme pomocou dvoch (navzájom nekompatibilných) adries.

Rovnakým spôsobom postupujeme, keď potrebujeme preniesť dáta v pamäti z reálneho do chráneného režimu. Je pravda, že celá oblasť pamäti reálneho režimu je priamo adresovateľná aj z chráneného režimu, no v chránenom režime túto pamäť obvykle aplikácii nesprístupníme, pretože adresný priestor aplikácie je uzavretý, aby nemohla chtiac či nechtiac poškodiť dôležité systémové štruktúry apod.

Po vykonaní API funkcie riadenie dostane opäť naša vrstva starajúca sa o preklad. Tá už len prepne režim späť na chránený a riadenie odovzdá pôvodnému programu, ktorý túto API funkciu zavolal. Hodnoty registrov táto vrstva meniť nesmie, pretože tvorí iba akýsi člen navyše, ktorý je nutný z hľadiska réžie okolo chráneného režimu a pokiaľ niektoré registre zmení, tak ich musí aj obnoviť do pôvodného stavu.

5. Privilegované a neprivilegované inštrukcie

5.1 Inštrukcie obecne

Jedna inštrukcia je základná a z hľadiska programátora ďalej nedeliteľná operácia, ktorú procesor vykonáva. Môže sa jednať o zmenu hodnoty registra, presun hodnoty registra do pamäti, sčítanie dvoch registrov atď. Samotná inštrukcia toho sama veľa nedokáže, ale kód programu obsahuje týchto inštrukcií tisíce až milióny a tieto inštrukcie dohromady tvoria programový kód.

V našom pohľade na vec sa nebudeme zaoberať inštrukciami ako takými, nás predovšetkým zaujímajú inštrukcie potrebné pre správu chráneného a virtuálneho režimu.

Chránený a virtuálny režim vyžaduje silnú hardvérovú podporu. Obsahuje novú správu pamäti, systém ochrany, nový systém spracovania prerušení, podporuje hardvérový multitasking a mnoho ďalšieho, čo sme sa mohli dočítať v predchádzajúcich kapitolách. Aby mohol správca chráneného režimu (napr. operačný systém) riadiť celý režim a tým pádom chovanie celého systému, musí pre to mať nejaké softvérové prostriedky. Tieto softvérové prostriedky sa nazývajú ***privilegované inštrukcie***.

Privilegované inštrukcie sú špeciálne inštrukcie, ktoré slúžia na správu chráneného a virtuálneho režimu, pomocou ktorých správca tohto režimu, resp. programový kód, ktorý má dostatočné oprávnenia, môže modifikovať rôzne systémové tabuľky, nastavovať špeciálne registre procesora atď.

V drvivej väčšine prípadov musí mať program, ktorý chce tieto inštrukcie využívať, nastavenú najvyššiu prioritu, aká v chránenom režime existuje ***(CPL=0)***. Pokiaľ túto prioritu program nemá a vyvolá nejakú z týchto inštrukcií, tak procesor vyvolá výnimku, inštrukciu samozrejme nevykoná a riadenie dostane obsluha tejto výnimky. Obsluha už sama rozhodne, čo spraví s aplikáciou, ktorá sa nechová korektne. Procesor buď vykoná výnimku ***GPF*** *(General Protection Fault)*, alebo ***Opcode Trap***, čo znamená neznáma inštrukcia.

Väčšinu privilegovaných inštrukcií nie je vôbec možné použiť ani vo virtuálnom režime, pretože kód virtuálneho režimu vždy má nastavenú najnižšiu prioritu ***(CPL=3)*** a prípadné zavolanie inštrukcie opäť vyvolá výnimku procesoru.

Existujú aj obyčajné, tzv. ***neprivilegované inštrukcie***, ktoré tiež môžu pomôcť pri správe chráneného režimu. Tieto inštrukcie síce nevyvolávajú pri použití v hocijakom kóde výnimky procesora, ale spravidla majú len informatívny charakter. Môžu získať informácie napr. zo špeciálneho registra procesora, aby aplikácia mohla zistiť nejaké systémové informácie *(napr. v akom režime sa vykonáva)*, ale nemôže tento register nijako modifikovať a tým pádom systému uškodiť.

5.2 Neprivilegované inštrukcie

V tejto kapitole popíšem inštrukcie, ktoré sú neprivilegované, ale súvisia s chráneným režimom.

smsw *reg/mem16 (store machine status word)*:

- táto inštrukcia uloží do svojho operandu tzv. ***MSW*** *(Machine Status Word)*, čo je v podstate dolných 16 bitov riadiaceho registra ***CR0***. Štruktúra registra CR0 je popísaná v kapitole 4 a obsahuje mnoho užitočných informácií o procesore. Často sa

používa na jednoduché zistenie, či bol program spustený v reálnom, alebo virtuálnom režime.

***cli** (clear interrupt flag), **sti** (set interrupt flag)*:

- tieto 2 inštrukcie zakazujú a povoľujú prerušenia a to v chránenom režime môže znamenať odstavenie systému. Preto, pokiaľ ***IOPL** (I/O Privilege Level)* < ***CPL** (Current Privilege Level)*, procesor generuje výnimku a tieto inštrukcie nevykoná. Samozrejme, väčšinou správca chráneného režimu emuluje tieto 2 inštrukcie, fyzicky prerušenia nezakáže (vynulovaním ***IF*** príznaku), ale nebude asynchrónne prerušenia premietať do programu, ktorý si to vyžiadal.

5.3 Privilegované inštrukcie

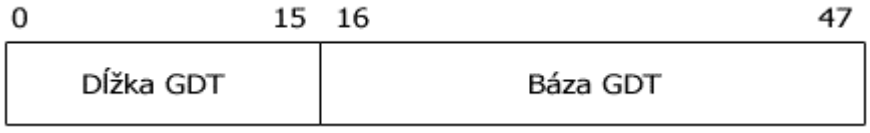
V tejto kapitole popíšem privilegované inštrukcie, ktoré sa používajú pre správu chráneného a virtuálneho režimu.

***lmsw** reg/mem16 (load machine status word)*:

- táto inštrukcia funguje opačne ako inštrukcia *smsw*, tzn. prečíta hodnotu zo svojho operandu a uloží ju do ***MSW** (Machine Status Word)*. Inštrukcia neumožňuje vynulovať bit ***PE** (Protection Enable)*, ktorý je nastavený, pokiaľ sa procesor nachádza v chránenom režime. Vo virtuálnom režime táto inštrukcia generuje výnimku (ako aj všetky ďalšie popísané).

***lgdt** mem48 (load global descriptor table)*:

- veľmi dôležitá inštrukcia, ktorá je nevyhnutná pre vstup do chráneného režimu. Táto inštrukcia prečíta 6 bajtov z operačnej pamäti a uloží ich do registra ***GDTR** (Global Descriptor Table Register)*. Tento register potom slúži procesoru ako ukazovateľ na globálnu tabuľku deskriptorov.



Obr. 17 – Štruktúra operandu pre lgdt inštrukciu

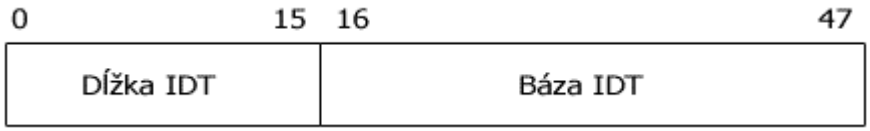
Operand pre inštrukciu ***lgdt*** obsahuje bázu ***GDT***, čo je lineárna adresa začiatku globálnej tabuľky deskriptorov a dĺžku ***GDT***, čo je veľkosť tejto tabuľky v bajtoch.

***sgdt** mem48 (store global descriptor table)*:

- táto inštrukcia je opäť opačná k inštrukcii ***lgdt***, do svojho operandu (6 bajtov v pamäti) uloží obsah registra ***GDTR** (Global Descriptor Table Register).*

lidt *mem48 (load interrupt descriptor table)*:

- táto inštrukcia funguje podobným spôsobom ako ***lgdt***, avšak namiesto registra ***GDTR*** naplní svojím operandom register ***IDTR*** *(Interrupt Descriptor Table Register)*. Tento register slúži procesoru ako ukazovateľ na tabuľku deskriptorov brán, používaných pri prerušeniach.



Obr. 18 – Štruktúra operandu pre lidt inštrukciu

Operand pre inštrukciu ***lidt*** obsahuje bázu ***IDT***, čo je lineárna adresa začiatku tabuľky deskriptorov brán a dĺžku ***IDT***, čo je veľkosť tejto tabuľky v bajtoch.

sidt *mem48 (store interrupt descriptor table)*:

- táto inštrukcia opäť funguje naopak, ako inštrukcia ***lidt***, do svojho operandu (6 bajtov v pamäti) uloží obsah registra ***IDTR*** *(Interrupt Descriptor Table Register)*.

lldt *reg/mem16 (load local descriptor table)*:

- táto inštrukcia tiež slúži na naplnenie registra ***LDTR*** *(Local Descriptor Table Register)*, no funguje trochu inak, ako inštrukcie ***lgdt*** a ***lidt***. Pretože sa jedná o lokálnu tabuľku deskriptorov, tak tá je pre každú úlohu iná (mala by byť) a vyžaduje rýchle prepínanie. Preto operandom tejto inštrukcie je ***selektor***, ktorý ukazuje do globálnej tabuľky deskriptorov na deskriptor typu ***LDT*** (viď kapitola 3).

sldt *reg/mem16 (store local descriptor table)*:

- táto inštrukcia uloží do svojho operandu ***selektor*** aktívnej ***LDT***, ktorý sa nachádza v ***GDT***. Opäť funguje naopak, ako inštrukcia ***lldt***.

ltr *reg/mem16 (load task register)*:

- táto inštrukcia sa používa pri použití hardvérového multitaskingu, pričom naplní obsah registra ***TR*** *(Task Register)* podľa svojho operandu, ktorý obsahuje ***selektor*** aktívneho ***TSS*** *(Task Segment State)* v ***GDT*** *(*jedná sa prirodzene o deskriptor).

str *reg/mem16 (store task register)*:

- opäť opačná inštrukcia k inštrukcii ***ltr***, do svojho operandu uloží ***selektor*** aktívneho ***TSS*** v globálnej tabuľke deskriptorov.

hlt *(halt processor)*:

- táto inštrukcia je podobný prípad ako ***cli*** a ***sti***. Jej činnosť pozostáva v pozastavení procesora, dokiaľ sa neobjaví asynchrónne prerušenie. Používa sa v prípade, že procesor nemá čo počítať a tým sa šetrí spotreba elektrickej energie. No v chránenom režime ju môže použiť iba program s ***CPL=0*** a vo virtuálnom režime vždy generuje výnimku procesoru (správca virtuálneho režimu ju môže emulovať).

6. Spolupráca chráneného a reálneho režimu

6.1 Štandard DPMI

V kapitole 4 sme si vysvetlili technické riešenie spolupráce chráneného a reálneho režimu, ktoré v podstate pozostávalo z volaní funkcií a procedúr reálneho režimu z chráneného režimu a naopak. Museli sme si dať pozor, aby sa procesor korektne prepol do nového režimu a aby bol obsah pamäti prístupný z reálneho režimu, aby veľkosť poľa v reálnom režime nepresiahla 64 KB apod.

Aby programátor nemusel strácať svoj drahocenný čas vytváraním kódu, ktorý sa bude starať o tieto nízko úrovňové záležitosti, existuje štandard ***DPMI*** *(DOS Protected Mode Interface)*. Tento štandard poskytuje aplikáciám softvérové prostriedky, aby mohli komfortne fungovať v chránenom režime, pričom prostredie, z ktorého sa spúšťajú, môže byť kľudne 16-bitový operačný systém bežiaci v reálnom režime (typicky ***DOS***).

Štandard ***DPMI*** definuje veľa užitočných funkcií, ktoré aplikácia nachádzajúca sa v chránenom režime využíva na komunikáciu medzi sebou a operačným systémom a nemusí sa toľko starať o technickú stránku veci, ako je prepínanie režimov procesora (napr. pri volaní funkcií ***BIOSu***), alebo správne umiestnenie údajových štruktúr v pamäti apod. Programovanie aplikácií určených pre chránený režim je v prípade použitia štandardu ***DPMI*** oveľa jednoduchšie.

6.2 Funkcie DPMI

Štandard ***DPMI*** definuje veľké množstvo funkcií, ktoré sú prístupné pomocou dvoch prerušení, a to ***0x2F*** a ***0x31***. Prerušenie ***0x2F*** je tzv. ***multiplexné prerušenie***, ktoré slúži mnohým procesom pod operačným systémom ***DOS***, pričom každý ďalší proces, ktorý chce obsluhovať toto prerušenie, sa musí nainštalovať do reťazca tohto prerušenia. ***DPMI*** pomocou tohto prerušenia obsluhuje iba málo funkcií, ktoré majú predovšetkým zaistiť vstup aplikácie využívajúcej ***DPMI*** do chráneného režimu. Prerušenie ***0x31*** obsahuje všetky ostatné funkcie, ktoré slúžia na správu chráneného režimu a tým pádom tieto funkcie sú dostupné iba z chráneného režimu.

Jednotlivé funkcie ***DPMI*** sa podľa štandardu odlišujú hodnotou registra ***AX***, do ktorého musíme uložiť správne číslo funkcie ešte pred zavolaním prerušenia ***0x31***. Samozrejme musíme naplniť aj ostatné hodnoty registrov podľa špecifikácie ***DPMI***.

DPMI sa ďalej vyskytuje v dvoch verziách, staršej verzii ***0.9*** a novšej verzii ***1.0***. Novšia verzia obsahuje niekoľko vylepšení navyše, ako je napr. správa viacerých deskriptorov naraz, stránkovanie a zdieľaná pamäť. Avšak pre vývoj plnohodnotnej aplikácie využívajúcej chránený režim a jeho výhody bohato postačí verzia ***0.9***.

V nasledujúcej tabuľke je uvedený vyčerpávajúci výpis jednotlivých funkcií ***DPMI***, pričom je uvedené číslo funkcie, jej význam, prerušenie, pomocou ktorého je dostupná a verzia ***DPMI***:

Initialization Services			
Function number	**Meaning**	**Interrupt**	**DPMI version**
0x1680	Release Current Virtual Machine's Time Slice	2F	1.0
0x1686	Get CPU Mode	2F	0.9

0x1687	Obtain Real-to-Protected Mode Switch Entry Point	2F	0.9
0x168A	Get Vendor-Specific API Entry Point	2F	1.0

LDT Management Services

Function number	Meaning	Interrupt	DPMI version
0x0000	Allocate LDT Descriptors	31	0.9
0x0001	Free LDT Descriptor	31	0.9
0x0002	Segment to Descriptor	31	0.9
0x0003	Get Selector Increment Value	31	0.9
0x0006	Get Segment Base Address	31	0.9
0x0007	Set Segment Base Address	31	0.9
0x0008	Set Segment Limit	31	0.9
0x0009	Set Descriptor Access Rights	31	0.9
0x000A	Create Alias Descriptor	31	0.9
0x000B	Get Descriptor	31	0.9
0x000C	Set Descriptor	31	0.9
0x000D	Allocate Specific LDT Descriptor	31	0.9
0x000E	Get Multiple Descriptors	31	1.0
0x000F	Set Multiple Descriptors	31	1.0

Extended Memory Management Services

Function number	Meaning	Interrupt	DPMI version
0x0500	Get Free Memory Information	31	0.9
0x0501	Allocate Memory Block	31	0.9
0x0502	Free Memory Block	31	0.9
0x0503	Resize Memory Block	31	0.9
0x0504	Allocate Linear Memory Block	31	1.0
0x0505	Resize Linear Memory Block	31	1.0
0x0506	Get Page Attributes	31	1.0
0x0507	Set Page Attributes	31	1.0
0x0508	Map Device in Memory Block	31	1.0
0x0509	Map Conventional Memory in Memory Block	31	1.0
0x050A	Get Memory Block Size and Base	31	1.0
0x050B	Get Memory Information	31	1.0
0x0800	Physical Address Mapping	31	0.9
0x0801	Free Physical Address Mapping	31	1.0
0x0D00	Allocate Shared Memory	31	1.0
0x0D01	Free Shared Memory	31	1.0
0x0D02	Serialize on Shared Memory	31	1.0
0x0D03	Free Serialization on Shared Memory	31	1.0

DOS Memory Management Services

Function number	Meaning	Interrupt	DPMI version
0x0100	Allocate DOS Memory Block	31	0.9
0x0101	Free DOS Memory Block	31	0.9
0x0102	Resize DOS Memory Block	31	0.9

Interrupt Management Services

Function number	Meaning	Interrupt	DPMI version
0x0200	Get Real Mode Interrupt Vector	31	0.9
0x0201	Set Real Mode Interrupt Vector	31	0.9
0x0202	Get Processor Exception Handler Vector	31	0.9
0x0203	Set Processor Exception Handler Vector	31	0.9

0x0204	Get Protected Mode Interrupt Vector	31	0.9
0x0205	Set Protected Mode Interrupt Vector	31	0.9
0x0210	Get Extended Exception Handler Vector (PM)	31	1.0
0x0211	Get Extended Exception Handler Vector (RM)	31	1.0
0x0212	Set Extended Exception Handler Vector (PM)	31	1.0
0x0213	Set Extended Exception Handler Vector (RM)	31	1.0
0x0900	Get and Disable Virtual Interrupt State	31	0.9
0x0901	Get and Enable Virtual Interrupt State	31	0.9
0x0902	Get Virtual Interrupt State	31	0.9
Translation Services			
Function number	**Meaning**	**Interrupt**	**DPMI version**
0x0300	Simulate Real Mode Interrupt	31	0.9
0x0301	Call Real Mode Procedure With Far Return Frame	31	0.9
0x0302	Call Real Mode Procedure With IRET Frame	31	0.9
0x0303	Allocate Real Mode Callback Address	31	0.9
0x0304	Free Real Mode Callback Address	31	0.9
0x0305	Get State Save/Restore Addresses	31	0.9
0x0306	Get Raw Mode Switch Addresses	31	0.9
Page Management Services			
Function number	**Meaning**	**Interrupt**	**DPMI version**
0x0600	Lock Linear Region	31	0.9
0x0601	Unlock Linear Region	31	0.9
0x0602	Mark Real Mode Region as Pageable	31	0.9
0x0603	Relock Real Mode Region	31	0.9
0x0604	Get Page Size	31	0.9
0x0702	Mark Page as Demand Paging Candidate	31	0.9
0x0703	Discard Page Contents	31	0.9
Debug Support Services			
Function number	**Meaning**	**Interrupt**	**DPMI version**
0x0B00	Set Debug Watchpoint	31	0.9
0x0B01	Clear Debug Watchpoint	31	0.9
0x0B02	Get State of Debug Watchpoint	31	0.9
0x0B03	Reset Debug Watchpoint	31	0.9
Miscellaneous Services			
Function number	**Meaning**	**Interrupt**	**DPMI version**
0x0400	Get Version	31	0.9
0x0401	Get DPMI Capabilities	31	1.0
0x0A00	Get Vendor-Specific API Entry Point	31	0.9
0x0C00	Install Resident Service Provider Callback	31	1.0
0x0C01	Terminate and Stay Resident	31	1.0
0x0E00	Get Coprocessor Status	31	1.0
0x0E01	Set Coprocessor Emulation	31	1.0

[zdroj 5]

6.3 Extendery

Predstavili sme si štandard ***DPMI*** a jeho účel. No na to, aby sme ho mohli použiť, potrebujeme softvér, ktorý nám funkčnosť ***DPMI*** sprístupní. Pod operačným systémom ***DOS*** sa na to využívajú tzv. ***Extendery***.

Extender je špeciálny program, ktorý je spustiteľný z ***DOSu*** a po svojom spustení zostane rezidentný v pamäti a zavedie do systému rozhranie ***DPMI***. Nainštaluje sa na multiplexné prerušenie (v reálnom režime) a aj na prerušenie ***0x31*** (v chránenom režime). ***Extender*** potom slúži ako hostiteľ chráneného režimu používaný aplikáciami, ktoré vyžadujú pre svoj beh chránený režim (tzv. klienti).

Extendery existujú ako 16-bitové, tak aj 32-bitové (aj kombinované) a podporujú ***DPMI*** štandard ***0.9*** alebo ***1.0***. Hojne ich využívajú aj komerčné aplikácie, ako napr. *Borland Pascal*, *Borland C++*, *Free Pascal*, *MASM*, *NASM*, *TASM* a mnoho ďalších.

Niektoré ***extendery*** idú ešte ďalej a ponúkajú na prvý pohľad neuveriteľné možnosti, ako je spúšťanie natívnych ***Win32*** aplikácií pod 16-bitovým operačným systémom MS-DOS. Rozširujú totiž pôvodné ***API*** funkcie štandardu ***DPMI*** a pridávajú nové ***API*** funkcie jadra (kernelu) 32-bitového Windows a aj ***API*** funkcie grafického subsystému ***GDI*** alebo ***OpenGL***.

Potom je možné úspešne spustiť pod MS-DOSom buď konzolové aplikácie systému Windows, alebo aj jednoduché aplikácie využívajúce grafické prostredie Windows alebo ***OpenGL***.

Zoznam niektorých známych a úspešných ***extenderov***:

- **DOS/4GW** (Tenberry Software)
- **PMODE/W** (Thomas Pytel & Charles Sheffold's)
- **HX DOS Extender** (Japheth)
- **CWSDPMI** (Charles W. Sandmann)
- **GO32** (DJ Delorie)

7. Demonštračné programy

7.1 Popis programov

V predchádzajúcich kapitolách sme si vysvetlili teoretickú časť fungovania a správy chráneného režimu, no súčasťou tejto bakalárskej práce je aj praktická časť, ktorá pozostáva zo série demonštračných programov.

Tieto programy prezentujú funkčnosť a použitie chráneného režimu primárne pod operačným systémom ***DOS***. Každý z programov slúži na demonštráciu nejakej oblasti použitia chráneného režimu, resp. každý program názorne prezentuje konkrétne vlastnosti alebo schopnosti chráneného režimu. Jednotlivé programy sú očíslované indexom a obecne platí, že čím vyšší index, tým zložitejší program, pretože programy s vyšším indexom vychádzajú z programov s nižším indexom a rozširujú jeho schopnosti.

1-test.cpp:

Tento program je veľmi jednoduchý, jeho celá funkčnosť je vypísať správu, či bol spustený v reálnom režime, alebo v režime virtuálnom. Tento test je veľmi dôležitý, lebo pre inicializáciu chráneného režimu u ďalších programov je nutné, aby boli vždy spustené iba z reálneho režimu.

2-gdt.cpp:

Program 2-gdt.cpp už fyzicky prepne procesor do chráneného režimu a z neho vypíše správu na obrazovku. Najprv vykoná test z predošlého programu, či je možné vôbec chránený režim inicializovať. Ak áno, tak si program alokuje 256 bajtov na systémovú tabuľku ***GDT***, do ktorej sa vmestí 32 deskriptorov. Ďalej tento program vyplní ***GDTR*** register, nastaví jeho limit a bázovú adresu. V ďalšom kroku sa vytvorí 5 deskriptorov, 1 alias deskriptor ku kódovému segmentu programu *(CS)*, 3 alias deskriptory k dátovým segmentom programu *(DS,ES,SS)* a 1 dátový deskriptor s bázovou adresou videopamäti textového režimu. V tomto momente sú už minimálne požiadavky pre chod chráneného režimu zaistené a program sa doňho prepne. Z chráneného režimu sa zobrazí na obrazovke správa a program sa prepne späť do reálneho režimu a ukončí sa. Prerušenia boli celý čas behu chráneného režimu zakázané.

3-idt.cpp:

Tento program funguje rovnakým spôsobom, ako predchádzajúci, avšak rozširuje chránený režim o spracovanie prerušení. Preto pri inicializácii tento program navyše alokuje pamäť pre 256 deskriptorov ako tabuľku ***IDT*** a vyplní register ***IDTR*** – nastaví jeho limit a bázovú adresu. Po vyplnení dátových a kódových deskriptorov rovnakým spôsobom, ako v predošlom prípade musí ešte vyplniť brány, teda deskriptory určené na spracovanie prerušenia. Všetky brány okrem jednej program nastaví na prázdnu obsluhu prerušenia, ktorá obsahuje akurát kód pre reset radiča prerušení. Brána, ktorá je na prerušení č.8 je asynchrónne volaná hodinami každých približne 55 ms a tá je nastavená na obsluhu prerušenia časovača. Pri každom tiku hodín táto obsluha vykreslí na obrazovke bodku. Preto po vstupe tohto programu do chráneného režimu je zobrazená správa, ako aj v predošlom prípade a navyše je horný riadok obrazovky zapĺňaný bodkami. Potom sa program prepne späť do reálneho režimu a ukončí sa.

4-vga.cpp:

Tento program demonštruje použitie grafického režimu v chránenom režime. Vychádza z programu *2-gdt.cpp* a rozširuje ho o grafický režim. Inicializácia chráneného režimu prebieha rovnako, ako v predošlom prípade (okrem prerušení). Namiesto alokovania dátového deskriptora odkazujúceho sa na videopamäť s bázovou adresou textového režimu *(0xB8000)* si v tomto prípade alokujeme dátový deskriptor s bázovou adresou grafického režimu *(0xA0000)*. Po inicializácii, ešte pred vstupom do chráneného režimu zavoláme štandardnú rutinu ***VGA BIOSu***, ktorá prepne textový režim na grafický režim a obrazovka sčernie. Po následnom prepnutí do chráneného režimu vykreslíme na obrazovku 15 farebných obdĺžnikov pomocou priameho zápisu do videopamäti za použitia nami vytvoreného deskriptora. Následne sa program prepne späť do reálneho režimu a ukončí sa.

5-multi.cpp:

Program *5-multi.cpp* tiež vychádza z programu *2-gdt.cpp* a rozširuje jeho možnosti. Tento program nám prezentuje jeden zo spôsobov využitia multitaskingu v chránenom režime, a to konkrétne kooperatívny multitasking (teda nie preemptívny multitasking s hardvérovou podporou). Inicializácia chráneného režimu je úplne rovnaká, ako v predchádzajúcom prípade, no tento program navyše obsahuje 4 funkcie, ktoré predstavujú 4 nezávislé úlohy. Program ich cyklicky strieda jednu za druhou (algoritmus FCFS – First Come, First Served). Každá z úloh inkrementuje svoju premennú a výsledok zobrazuje na obrazovke. V programe je dobre vidieť striedanie jednotlivých úloh. Po určitej dobe sa program prepne späť do reálneho režimu a ukončí sa.

7.2 Dôležité funkcie

V tejto kapitole popíšem niektoré dôležité funkcie, ktoré sa v programoch používajú a ktoré sú nevyhnutné pre beh chráneného režimu.

int ***get_mode****()*:

Táto funkcia na zistenie režimu práce procesora používa neprivilegovanú inštrukciu ***smsw***, ktorá do operandu ***AX*** uloží hodnotu ***MSW***. Najnižší bit tejto hodnoty určuje, v akom režime sa procesor nachádza *(bit **PE**)*.

void ***set_descriptor****(unsigned long base,unsigned long limit,TYPE type,int selector)*:

Táto funkcia vyplňuje v tabuľke ***GDT*** konkrétny deskriptor udaný hodnotou selektora, pričom tomuto deskriptoru nastavuje bázovú adresu, limit a typ deskriptora. Typ môže byť buď kódový deskriptor, alebo dátový deskriptor. Po vykonaní tejto funkcie je možné so selektorom pracovať.

void ***set_interrupt****(void far *proc,int no)*:

Táto funkcia vyplňuje v tabuľke ***IDT*** konkrétny deskriptor typu brány, pričom pozícia deskriptora v tabuľke je udaná konkrétnym prerušením a deskriptor je naplnený hodnotou adresy obsluhy prerušenia.

void near ***show_message****(char near *msg)*:

Táto funkcia zobrazí v chránenom režime správu na obrazovke. Používa pritom priamy zápis do videopamäti s použitím deskriptora, ktorý bol dopredu vytvorený. Číta reťazec znak po znaku a dokiaľ znak nie je nulový, tak ho zobrazí aj s farebným atribútom na obrazovke.

void near ***go_PM()***:

Toto je najdôležitejšia funkcia všetkých programov, lebo prepína procesor z reálneho do chráneného režimu. Najprv zakáže asynchrónne prerušenia, potom nastaví bit ***PE*** v ***CR0*** registri na 1 a tým povie procesoru, že má prevádzkovať chránený režim. Hneď potom inicializuje systémové registre ***GDTR*** a prípadne aj ***IDTR***, aby procesor vedel, kde sa nachádzajú jeho systémové tabuľky. Pre plnohodnotný chránený režim musí ale procesor ešte vykonať vzdialený skok do nového kódového segmentu, v tomto prípade pomocou improvizovanej inštrukcie ***retf*** a v novom kódovom segmente ešte inicializuje všetky segmentové registre tak, aby boli kompatibilné s chráneným režimom. Tieto nové segmenty sú aliasy k starým segmentom v reálnom režime, ako už bolo povedané.

void near ***go_RM()***:

Táto funkcia je v podstate opačná k funkcii *go_PM*, pretože prepína procesor z chráneného režimu do režimu reálneho. Obdobným spôsobom najprv vynuluje bit ***PE*** v ***CR0*** registri a tým dá procesoru najavo, že ďalej bude prevádzkovať reálny režim. Pomocou inštrukcie ***retf*** vykoná vzdialený skok do pôvodného reálneho kódového segmentu a inicializuje všetky segmentové registre na pôvodné hodnoty. Pokiaľ bol zmenený register ***IDTR***, treba ho vrátiť na pôvodnú hodnotu, teda predovšetkým nastaviť jeho bázovú adresu na 0, lebo na tejto adrese sa začína tabuľka vektorov prerušení v reálnom režime.

7.3 Uvedenie do prevádzky

Demonštračné programy sú napísané v jazyku C++ a s použitím vkladaného assembleru. Pre preloženie zdrojových súborov treba mať nainštalovaný prekladač ***Borland C++*** (napr. verziu 3.1), ktorý je voľne dostupný.

Preklad súborov:

```
bcc -3 -B 1-test.cpp
bcc -3 -B 2-gdt.cpp
bcc -3 -B 3-idt.cpp
bcc -3 -B 4-vga.cpp
bcc -3 -B 5-multi.cpp
```

Po preklade dostaneme binárne súbory s príponou ***exe*** spustiteľné pod operačným systémom ***DOS***. Aby boli programy funkčné, treba ich spustiť buď priamo pod operačným systémom ***DOS*** alebo v emulátore, ako je napr. ***DOSBox***. Dôležité je, aby sa procesor nachádzal v reálnom režime a nie v režime virtuálnom, teda aby neboli nainštalované niektoré špeciálne ovládače, ako napr. ***EMM386.EXE***.

8. Záver

Neustále zvyšujúce sa nároky na výkon, spoľahlivosť a bezpečnosť výpočtovej techniky sa premietajú aj do dizajnu architektúry procesorov. Dnešné procesory už sú zas o kus ďalej a využívajú 64-bitovú inštrukčnú sadu. Cez to všetko neustále podporujú 32-bitový chránený režim a sú spätne kompatibilné so starým procesorom 8086, z ktorého pôvodne vyšli. Napriek tomu dnes sú ešte stále hojne používané 32-bitové procesory, ktoré prevádzkujú predovšetkým chránený režim.

Chránený režim je v súčasnosti používaný vo všetkých moderných operačných systémoch pre jeho výhodné vlastnosti. Už prvé verzie systému ***Windows*** ho používali vo väčšej či menšej miere, napr. ***Windows 3.1*** obsahoval 2 módy práce tohto systému – štandardný mód 286 a rozšírený mód 386. Pre systém ***Linux*** je chránený režim samozrejmosťou.

Chránený režim je síce veľmi výhodný ako pre užívateľské aplikácie, tak aj pre operačný systém, avšak si vyžaduje dodatočný softvér pre svoju správu a jeho chod a spracovanie procesorom je o dosť komplexnejšie, ako je tomu v prípade reálneho režimu. Je to dané jeho pomerne náročným dizajnom. Jeho zložitosť oproti reálnemu režimu je daňou za jeho nenahraditeľné schopnosti, ktoré umožňujú chrániť operačný systém pred spustenými aplikáciami a aj aplikácie medzi sebou navzájom.

Čitateľ, ktorý si túto prácu preštudoval, by mal pochopiť základné princípy chráneného režimu, jeho výhody, ale aj nevýhody pri použití, predovšetkým pri jeho použití pod 16-bitovým operačným systémom, ktorý nevie o chránenom režime nič a jeho kód je naprogramovaný čisto vo štýle reálneho režimu. Čitateľovi by malo byť jasné, aké sú dôvody na používanie chráneného režimu, akým spôsobom v ňom funguje správa pamäti a akým spôsobom sú spracované prerušenia.

Použitá literatúra

1. Ladislav Vágner: AT help v 1.50, Elektronický manuál, 1996.

2. Intel 80386, http://en.wikipedia.org/wiki/Intel_80386, 1.5.2009

3. x86 Memory Segmentation, http://en.wikipedia.org/wiki/Segment_register, 9.2.2009

4. Control Register, http://en.wikipedia.org/wiki/Control_register#CR0, 10.3.2009

5. DJ Delorie: DPMI Function by Functional Group, http://www.delorie.com/djgpp/doc/dpmi/ch5.g.html, 1996

6. Brandejs, M.: Mikroprocesory Intel 8086 – 80486, Grada, 1991

7. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture, 2007