Otázka č.10 z okruhu 3 – specializace řídící technika

Programování v prostředí 16-ti a 32-ti bitových Windows. Metody rychlého vývoje aplikací, RAD. Funkce, parametry, proměnné, výrazy. Alokace paměti, pole, struktury, ukazatele, reference. Konstruktory a destruktory. Dědění a atributy přístupu. Objekty a překrývání metod a prvků, virtuální metody. Dialogy a jejich elementy. Prostředí DOS, Win16 a Win32, rozdíly a správa paměti. Specifika Win32 - procesy, virtuální paměť, mapování souborů. DLL knihovny, DDE a OLE protokoly.

Programování ve Windows

Systémy Windows jsou založeny na grafickém uživatelském rozhraní (GUI). Každý element (text, editační řádek, tlačítko), který je na obrazovce vidět je okno. Ke každému oknu přísluší tzv. procedura okna, která obsluhuje veškeré události vztahující se k tomuto oknu přicházející ve formě zprávy. Každé okno je jednoznačně identifikováno svým handlem, což je vlastně index do systémové tabulky všech existujících oken v systému.

Programování ve Windows se od programování pod DOSem liší v základním přístupu, neboť Windows pracuje na systému zasílání zpráv. Každá událost (stisk klávesy, posun myši, vznik nového okna, žádost na překreslení okna apod. je převedena na zprávu zaslanou do okna. Jakým způsobem okno zprávu zpracuje, rozhoduje procedura okna. Příkladem jedné z nejdůležitějších zpráv je např. zpráva WM_PAINT, která může být oknu zaslána systémem, pokud je potřeba aktualizovat grafickou podobu okna. (Okno bylo třeba původě zakryto a zakrývající okno bylo posunuto). Aplikace využívá služeb operačního systému pomocí volání Windows API funkcí.

Metody rychlého vývoje aplikací

Z důvodů formálně velmi podobného kódu pro obsluhu jednotlivých oken je výhodné využít množství knihoven a podpůrných nástrojů pro tvorbu aplikací. Příkladem může být objektově orientovaná knihovna od firmy Borland - Object Windows Library, pomocí které je velmi jednoduché napsat základní kostru aplikace. Integrované vývojové prostředí poskytuje i několik průvodců pro vytvoření několika typů aplikací. Po nadefinování několika vlastností se vygeneruje kostra aplikace. Firma Microsoft pro podobné účely zavedla knihovny Microsoft Foundation Class (MFC).

Dalším krokem pro zrychlení a zefektivnění vývoje aplikací jsou nástroje Borland Delphi, C++ Builder, MS Visual C++ (MS Visual Studio), Power Builder a mnohé další. Tato vývojová prostředí kromě objektové orientovaných knihoven podporují vizuální návrhy aplikací. Při programování v těchto prostředích je minimalizováno množství ručně psaného kódu, neboť návrh dialogů se provádí pouhým umisťováním nejrůznějších objektů (komponent) pomocí myši, místo definování vlastností a reakcí na události ve zdrojovém kódu mnohdy stačí jen vyplnit parametry do formuláře příslušejícího ke konkrétnímu objektu.

Pokud se týká vlastního rozboru a návrhu aplikace, jsou k dispozici nástroje jako CASE, Rational Rose a další. Tyto nástroje dovolují přehledně definovat strukturu aplikace, vazby mezi jednotlivými moduly, definovat datové toky, mají podporu pro logický a fyzický návrh apod. Nástroje z balíku Rational navíc obsahují podporu pro definici požadavku na SW, poloautomatickou podporu pro dokumentaci zdrojových kódů, testování modulů i kompletních produktů, podporují týmovou práci a mnoho dalších věcí.

Rozdíly DOS, Win16, Win32

DOS:

- jednoúlohový OS, není podpora pro běh více úloh (výjimka tzv. rezidentní programy běžící na pozadí)

- běh v reálném módu CPU-kompatibilita s PC XT (16/20 bitové adresování), správa jen 640 KB paměti, pro přístup ke zbylé paměti nutné použít služeb EMS / XMS, pro speciální účely existují 32 bitové DOS-extendery např. DOS4GW, jež pracují v chráněném režimu CPU (náročné programy, hry apod.)
- neexistence jiného standardního uživatelského rozhraní než příkazová řádka
- přímý přístup ke všem technickým prostředkům PC, pro každé nestandardní zařízení je potřeba použít speciálního přístupu (např. nastavení grafických módů VGA s vysokým rozlišením)
- nelze použít virtuální paměť

Aplikace typu Win16

- představitelem jsou nadstavby OS DOS Windows 3.x (nelze ještě mluvit o plnohodnotném OS)
- pracují jak v reálném, tak i chráněném režimu CPU, 16.bitová správa paměti, odkládací soubor
- grafické rozhraní, pro přístup k technickým prostředků PC využívá ovladačů

Aplikace typu Win32

- představitelem jsou OS Windows 95/98 a Windows NT, popis Win32 je zaměřen na Windows 95, ty vůči Win NT postrádají:
    - ochranu mezi aplikacemi, pád jedné může ohrozit jinou
    - dělení běhu úlohy na více procesorů
    - chybí OpenGL-3D, u u některých „resource“ stále limit na 64k textu
- Preemptivní multitasking
- Multi thread
- Do systému zakomponována obsluha **exceptions** **(**strukturované výjimky)
- Kromě GUI vestavěná možnost i Console aplikací
- Každá aplikace vlastní paměť „flat“ model 4Gbyte
- sdílení paměti jen přes „FileMapping“, ev. čtená data lze dát do DLL
- Virtuálnì mapovaná paměť
- Vše 32 bitů s výjimkou: **char**, **BYTE** (8), **short**, **WORD** (16)
    - Handle 32 bitù, oproti dřívějším 16
    - Pointry pouze **near** (32 bitù), **far** ignorováno
    - GUI souřadnice 32 bitů, pouze myš vrací 16 bitové souřadnice
    - message wParamer, lParameter - 32 bitù => někde vůči Win16 změna dekódování parametrů
- Registrace harwaru a aplikací
- Ikony 16x16 dole na lištì, 32x32 na pracovní ploše
- Nové prvky dialogů, například: List view, progress bar, slide, spin, tree view...
- Jména souborů 255 znakù a mohou obsahovat tečky a mezery
- možnost OVERLAPPED souborů

Modely paměti

DOS small – pouze 2 paměťové segmenty + far HEAP, typicky 200-300Kb podle velikosti pgm.
CS … kódový segment – vlastní program
DS … datový segment – (ne) / inicializovaná data, za daty následuje near HEAP, proti němu od konce segmentu roste zásobník (stack SP)

Win16 large– více CS, jeden DS, jeden SS (stack segment), heap 16 Mb

DOS huge – jako large, ale více datových segmentů, ovšem v kompilátoru v Borland C++ 3.1 je chyba

Win32 fat – segmenty neexistují – umisťuje je OS
kód programu | glob data inic/neinic | stack 1Mb s možností růstu | Heap až 4 GB

Pointry

Ve **small** modelu jsou pointry implicitně *near*, ale na far heap se musí ukazovat *far* pointrem, v **large** modelu by se měl používat *far* pointer, *near* může být v určitých situacích nebezpečný, ve **flat** modelu se obě klíč. slova *far* i *near* ignorují, vždy se ukazuje 32 bitovými pointry.

typ pointru	DOS16	WIN16	UNIX, WIN32
near	16 bit offset	16 bit offset	32 bit offset
far	16 seg : 16 seg	16 seg : 16 off	32 bit offset
huge	16 seg:16off – překryvné segm.	v Borland C nelze	v Borland C nelze

Funkce, parametry, proměnné, výrazy

Následující popisy se budou převážně opírat o programovací jazyk C++.

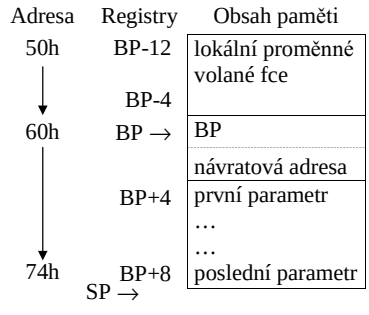
Priority operátorů ve výrazech v C++

1. Highest	() [] -> :: .	Function call Array subscript C++indirect component .selector C++scope access/resolution C++ direct component selector
2. Unary	! ~ + - ++ -- & * sizeof new delete	Logical negation (NOT) Bitwise (1's) complement Unary plus Unary minus Preincrement or postincrement Predecrement or postdecrement Address Indirection (returns size of operand, in bytes) (dyn.alloc. C++ storage) (dyn.dealloc.C++ storage)
3. Member access	.* ->*	C++ dereference C++ dereference
4. Multiplicative	* / %	Multiply Divide Remainder (modulus)
5. Additive	+, -	Binary plus, minus
6. Shift	<<, >>	Shift left, right

7. Relational	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to
8. Equality	==, !=	Equal to, Not equal to
9.	&	Bitwise AND
10.	^	Bitwise XOR
11.	|	Bitwise OR
12.	&&	Logical AND
13.	||	Logical OR
14. Conditional	?:	(a?x:y means"if a then x,else y")
15. Assignment	= *= /= %= += -= &= ^= |= <<= >>=	Simple assignment Assign product Assign quotient Assign remainder (modulus) Assign sum Assign difference Assign bitwise AND Assign bitwise XOR Assign bitwise OR Assign left shift Assign right shift
16. Comma	,	Evaluate

Volání funkcí

Při volání funkce je potřeba do volané funkce předat parametry, návratovou adresu a poskytnout prostor pro lokální proměnné. Pro všechny tyto účely slouží zásobník. Ať už jsou parametry předávány z hlediska jazyku hodnotou nebo odkazem, na nejnižší úrovni se předává vždy hodnota. (při předávání parametru odkazem se vlastně předává hodnota odkazu, tj. hodnota pointeru na proměnou)



obr. 1: Před voláním cdecl fce

V programu je možné volat funkci podle způsobu ukládaní parametrů na zásobník dvěma základními přístupy. Jedno volání je typu _cdecl, což je standardní volání jazyka C, druhý typ volání je tzv. PASCALské volání. Od sebe se navzájem liší pořadím parametrů předávaných na zásobní a umístěním návratové adresy. U pascalského volání je pořadí parametrů obrácené a návratová adresa se ukládá na zásobník jako první. Za vyčištění zásobníku odpovídá volaná funkce. Pascalské volání je důležité pro většinu volání funkcí přes rozhraní Windows API . Dále je popisováno jen volání _cdecl. Při volání funkce se nejprve vypočítává poslední parametr

funkce, který se ukládá na vrchol zásobníku, dále se vyhodnocují pozpátku parametry až k prvnímu a ukládají se na zásobník. Nakonec se na zásobník uloží návratová adresa a na zásobníku se vyhradí místo pro lokální proměnné volané funkce. Za vyčištění zásobníku odpovídá volající funkce. Tento mechanismus dovoluje předávání proměnného počtu parametrů.

Typy pointerů

```
int *pi;          // pointer na prom. typu int
int * const cpi;  // konstantní pointer na int
int const* pci;   // pointer na konstantní int
cpi++;            // nelze
pci++;            // lze
*pci = 5;         // nelze
const char *const s="TEXT";  // nic 's' nepřepíše
```

Pointry a pole

Přístup k prvkům pole přes [], násobné závorky se vyhodnocují zleva doprava a mají nevyšší prioritu. Kombinace referencí a dereferencí * / & v kombinaci s unárními operátory ! ++ -- se asociují zprava doleva, tedy *pc++ je to samé jako *(pc++).

Pole definované jako *char data[3][9] = {{"prvni\n"},{"druhy\n"},{"treti\n"}};* znamená alokaci tří řetězců po devíti znacích. Fyzický obsah paměti je pak: *prvni\n\x0\x0\x0druhy\n\x0\x0\x0treti\n\x0\x0\x0*

Reference

Reference je vlastně odkazová proměnná, tento mechanismus se používá, pokud je potřeba dvě proměnné svázat, nebo předávat proměnné do funkce odkazem (aby funkce měla při návratu vliv na proměnné vložené jako parametry). Syntaxe pro definici takové funkce je:
void prohod(int &a, int &b) { int c=a; a=b; b=c; } volání fce: int i,j; prohod(i,j);
Referenční proměnné: *int &ix = 6;* // lze překladač vytvoří pomocnou proměnnou s hodnotou 6 a ix na ní bude odkazovat. je to ekvivalentní zápisu:
*int ix = 6; int *pi = &ix;* $\Rightarrow$ *(*pi==6)* je pravda **pi* $\Leftrightarrow$ *ix;*
jiný příklad:
float f = 10.5;
int &iy = f; // lze, překladač vytvoří pomocnou prom. *f* typu int, ale při změně *f* se **iy* nemění.
ovšem v případě:
*int f; int *py = &f; *py = 5* // jsou proměnné svázány, $\Rightarrow$ *(f == 5)* je pravda.

DLL knihovny

V DLL knihovnách (Dynamic linked library) jsou uloženy skupiny funkcí, které mohou být vykonávaným programem připojeny až za běhu, tedy až pokud je kód v nich obsažený skutečně zapotřebí. Tvoří opak k tzv. staticky linkovaným knihovnám, které jsou již v době kompilace připojeny k programovému modulu. Výhodou DLL knihoven je to, že jejich programový kód lze sdílet několika aplikacemi, tudíž se může ušetřit operační paměť. (toto neplatí ve Win NT, kde je kód dyn. link. knihovny pro každou aplikaci znovu kopírován do odděleného paměťového prostoru – kvůli bezpečnosti a stabilitě). Ve Win 3.x a 9x je např. i jádro systému KERNEL32.DLL (KRNL286.EXE, KRNL386.EXE) jednou velkou DLL knihovnou, protože by bylo zbytečné, aby každá aplikace skupinou standardních funkcí opakovaně zaplňovala operační paměť. Výhoda DLL knihoven je též v možnosti modularizace programů, protože při opravě či aktualizaci jednoho modulu je možné vyměnit jen knihovnu. Drivery pro různá zařízení jsou v podstatě též DLL knihovny, výrobce pro každé zařízení dodává specifický ovladač, díky němuž systém k zařízení následně přistupuje.

Každá knihovna obsahuje vstupní bod – speciální funkci DLLEntryPoint() (ve Win32), která je při připojení k aplikaci standardně vyvolávána. V této funkci jsou prováděny inicializační kroky.

DDE protokol

DDE protokol (Dynamic Data Exchange) slouží k výměně dat mezi dvěma aplikacemi, která probídá na principu klient – server.

1. Registrace: MakeProcInstance() + DdeInitialize(...);
   DDE server - aplikace, zdroj dat - DdeNameService(...)
   DDE klient - žadatel dat
2. Zahájení konverzace inicializuje klient, může jich zahájit víc,
   DdeConnect(*idInstance*, Service, Topic, 0 /* *jazykový kontext* */);
3. Funkce pro přenos dat
   DdeClientTransaction(...)
      XTYP_ADVSTART / XTYP_ADVSTOP
      XTYP_POKE, XTYP_REQUEST, XTYP_EXECUTE,...
   HSZ DdeCreateStringHandle(...)
   int DdeCmpStringHandles(...)
4. Konec konverzace: DdeDisconnect(...)
5. Odhlášení: DdeUninitialize(idInst) + FreeProcInstance()

Starší způsob:

SendMessage: WM_DDE_INITILIZE / WM_DDE_ACK
PostMessage: WM_DDE_TERMINATE
WM_DDE_POKE
WM_DDE_REQUEST / WM_DDE_DATA
WM_DDE_ADVISE x WM_DDE_UNADVISE
WM_DDE_EXECUTE

Data se přenášela přes atomy.

Object Linked Exchange - OLE2 – popis na bázi OWL knihovny

OLE je určeno pro sdílení objektů mezi aplikacemi (typicky je to např. tabulka z Excelu vložená do Wordu. Při práci či úpravách objektu v hostitelské aplikaci se volají metody aplikace, odkud objekt pochází.

- Typ aplikace „**Automating Application**“ - druzí mohou používat její metody
  - registrace aplikace jako „automation server“
  - „Automate class - popíšeme metody a data
- „**OLE container**“ - schovává v sobě cizí data
- „**Automation controller (client)**“ - používá cizí metody a data
  - vytvoření TOleAllocator allocator(0);
  - deklarace „proxy“ tříd pro OLE objekty TAutoProxy.
  - implementace „proxy“ tříd
  - vytvoření „proxy“ objektù
- „**OLE server**“ - tvoří a spravuje datové objekty pro druhé programy
  - vytvoření TOleAllocator allocator(0);
  - registrace Registar, seznam dokumentù, IUnkwown
  - vytvoření nezávislého okna - „View Window“
- v hlavní okně obsloužit WM_OCEVENT zprávy

Základní objekty:

- **IUknown -** povinná metoda
  - QueryInterface - dotaz na existenci metody
  - AddRef ++ , Release --

- **IDispatch** - zjištění informace a manipulace s metodami a daty
  - GetTypeInfoCount - je-li IDispatch, pak vždy 1
  - GetTypeInfo
  - GetIDsOfNames - získání identifikátoru
  - Invoke - provedení metody
- **IClassFactory** - vytváří objekty
  - CreateInstance - vytvoří objekt
  - LockServer - zabrání zrušení serveru, pokud existuje objekt

Objektově orientované programování

Třídy a Objekty

Třídy a Objekty jsou základními stavebními kameny objektově orientovaného programování, které zobrazují abstrakce reálného světa. Objekty reprezentují základní entity za běhu programu, třídy je možno chápat jako jejich předlohu. Jaké atributy bude mít objekt a jakými metodami je definováno v popisu třídy objektu. Třídy mají:

- rozhraní, které definuje části objektů dané třídy, přístupné zvenčí
- tělo, které implementuje operace rozhraní
- instanční proměnné, které obsahují stav objektu dané třídy

Data a operace jsou podle přístupu rozlišena do třech kategorií:

- ***Public*** - data a operace přístupné zvenčí pro kohokoliv, kdo chce přistupovat k objektům dané třídy
- ***Protected*** - přístupné pouze v rámci dané třídy (a tříd deklarovaných jako spřátelené třídy) a podtříd
- ***Private*** - přístupné pouze v rámci dané třídy (a tříd deklarovaných jako spřátelené třídy)

Konstruktory a destruktory

C++ umožňuje automaticky provádět operace, které jsou nutné pro vznik a zánik objektu, neboli volat tak zvaný konstruktor a destruktor. V tomto směru se objektové C++ liší od objektového PASCALu, v němž je možné též deklarovat konstruktory a destruktory, ale musíme si je obsluhovat sami. V C++ překladač sám vkládá kód pro inicializaci a ukončení našich datových struktur.

Konstruktor deklarujeme jako metodu mající název shodný se jménem celé struktury. Pro příklad budeme pracovat s objektem zásobník - bude to tedy metoda:

ZASOBNIK(int velikost)

Destruktor definujeme jako metodu mající jméno struktury, před níž je předřazen znak ~. Pro zásobník bude mít destruktor tvar:

~ZASOBNIK()

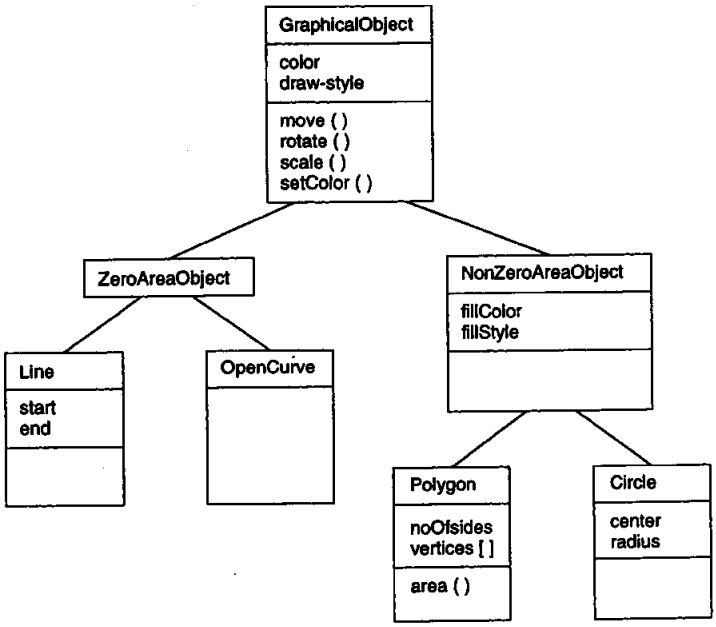
Konstruktory a destruktory mají následující vlastnosti:

1. Nesmějí vracet žádné parametry, dokonce ani typy void.
2. Metoda konstruktoru může mít formální argumenty. Má-li je, píšeme je do závorek za jméno proměnné. Například: ZASOBNIK cisla(100); je vytvoření datové struktury cisla typu ZASOBNIK s předáním hodnoty 100 jako argumentu jejímu konstruktoru. Kdybychom napsali jen: ZASOBNIK cisla; znamenalo by to, že chceme volat konstruktor bez parametrů, ale takový v našem příkladu nebudeme definovat, a proto by předchozí řádek vyvolal chybové hlášení: „Chybí konstruktor ZASOBNIK().“
3. Strukturu, která má definovaný konstruktor s jedním parametrem, lze vytvořit i zápisem: ZASOBNIK cisla = 100; Oba zápisy, se závorkami „cisla(100)“ a rovnítkem „cisla = 100“, jsou rovnocenné.
4. Konstruktor nemůžeme volat sami, jeho zařazení do kódu závisí výhradně na překladači, který ho vloží při deklaraci příslušné struktury.

5. Můžeme definovat víc konstruktorů, které se od sebe vzájemně liší použitými argumenty (o tom v části „Překrývání funkcí a metod“).
6. Destruktor nemá žádné formální argumenty a je vždy pouze jediný.
7. Destruktor lze volat jako běžnou metodu, ale s výjimkou speciálních situací se to nedělá. Vložení operací pro zrušení objektu se nechává na překladači.
8. Pokud neuvedeme metody konstruktoru nebo destruktoru, překladač místo nich vytvoří náhradní, které vytvoří či zruší objekt, ale neprovádějí žádné inicializace proměnných.
9. Překladač vytváří dva náhradní „default“ konstruktory:
   - bez argumentů, který existuje pouze tehdy, když jsme žádný jiný konstruktor nedefinovali
   - „copy“ konstruktor: TRIDA(TRIDA & ExistujiciTrida), který slouží k inicializaci z dat existující třídy stejného typu. Tento konstruktor je automaticky vytvořen, je-li použit a není námi definován.

Dědičnost

Dědičnost (inheritance) je koncept unikátní pro objektově orientované technologie. Dědičnost je relace mezi dvěma třídami, která umožňuje založit definici jedné třídy na jiné existující třídě. Jestliže třída B dědí od třídy A, pak B je podtřídou a třída A je nadtřídou. Podtřída B pak má obecně dvě části: odvozená část je zděděna z A a doplněna novou inkrementální částí, obsahující definice a kód příslušný přidanou třídou B. Dědění jedné třídy lze provést i děděním z většího počtu základních tříd, nová třída má pak vlastnosti dané sjednocením vlastností základních tříd.



obr. 2: Ukázka dědičnosti

Obecně je možné vlastnosti z A zděděné v B předefinovat (překrýt). Toto se týká jak datových prvků tak i metod. Tímto způsobem lze měnit i viditelnost operací (tj. je-li metoda v A public, může být v B předefinována jako private) popř. stejnojmenné operace z A nahradit v B jinými, přejmenovat je apod. Inheritanční relace mezi třídami vytváří objektové hiearchie. Jejich hlavní silou je fakt, že veškeré operace jejich podtříd je možné soustředit do jejich nadtříd, tj. vytvářet abstraktní třídy, které lze ve vlastním programu využít tím způsobem, že vytvoříme novou třídu jako specializaci jedné třídy nebo více tříd.

Dědičnost přináší ještě jednu významnou vlastnost – polymorfismus. V objektově orientovaném návrhu aplikace se projevuje v té podobě, že se můžeme stejným způsobem odvolávat na různé objekty v různém čase, tj. pracujeme s objektem, který je černou skříňkou, jejíž skutečný obsah je dán okolnostmi (např. obsahem zasílaných zpráv) až za běhu programu. Samozřejmě, že takto alternující objekty musí splňovat jisté podmínky.

Polymorfismus objektů: Mějme objekt x ze třídy B, objekt y ze třídy A, B je podtřídou A. Pak lze všude, kde je očekáván objekt třídy A použít objekt třídy B, stejně tak je možné přiřazení x:=y. Tímto přiřazením se dynamický typ objektu x změní z B na A (staticky zůstává).

Polymorfismus metod: vyžaduje tzv. dynamické vázání (jedná se o tzv.virtuální metody), které umožňuje dané operaci přiřadit kód skutečně prováděné metody až za běhu programu. Mějme opět třídy A a B jako v předchozím případu s tím, že v B budeme redefinovat zděděnou metodu M(), x má statický

typ B, dynamický A, nebo B. Pak skutečně volaná metoda pro x.M() závisí na tom, zda má objekt v danou chvíli dynamický typ A (pak se volá metoda A::M()),nebo B (pak se volá metoda B::M()).

Správa paměti

Operační systém a běžící aplikace často potřebují větší množství paměti, než je fyzicky v počítači instalováno. Proto je procesor vybaven systémem logického adresování, který umožňuje pracovat s větším paměťovým prostorem, než je velikost fyzické paměti. Běžící programy nikdy nepotřebují celý paměťový prostor najednou, je proto možné vždy jen část logického paměťového prostoru namapovat do fyzické paměti. Pro mapování se používají dva postupy

- Stránkování paměti
- Segmentace paměti

Stránkování – základní metody, pojmy

Fyzická paměť je rozdělena na logické stránky, viz obr. 3 a obr. 4, Nulová externí fragmentace, interní fragmentace neexistuje, Velikost stránky je 2, 4 kB, až 64 MB, overhead. Frame table (je/není alokována a kterému procesu). Kopie page table pro uživatele. Systémové volání. Doba kontextového přepnutí.

Stránkování – vytváření adres INTEL

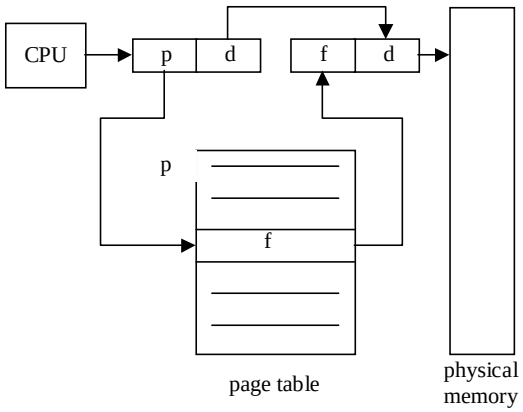
INTEL 486 a vyšší používají segmentaci se stránkováním. Maximální počet segmentů na jeden proces je 16 K, každý z nich může být velikost 4 GB. Velikost stránky jsou 4 KB. Celkový popis architektury je na obr. 5. Logický adresový prostor je možné rozdělit na dvě části, každá je tvořena 8 K segmentů:

- Privátní, popsaný pomocí local descriptor table (LDT)
- Sdílený, popsaný pomocí global descriptor table (GDT)

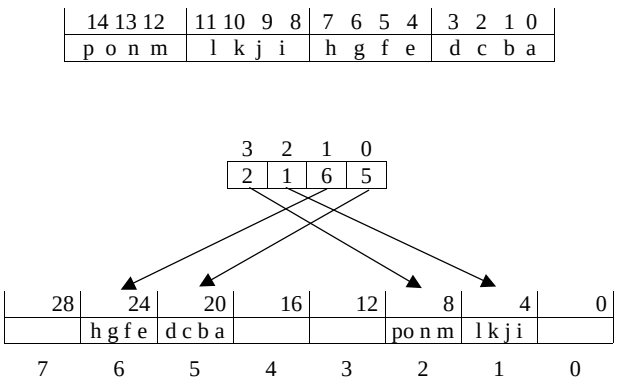
Každá položka v LDT, GDT je tvořena 8 byty a obsahuje podrobnou informaci o každém segmentu (počáteční adresa, délka segmentu, granularity, protection, writable, accessed).

Logická adresa je dvojice (selector, offset), kde selector je 16-ti bitové číslo skládající se z INDEXU (13), určení descriptor table (1) a protection (2).

Procesor má k dispozici 6 segment registrů, které umožňují použití 6 segmentů. Existuje 6 8-mi bytových mikroprogramních registrů, obsahující odpovídající descriptor z LDT, GDT (cache pro LDT, GDT).



obr. 3: Technické prostředky pro stránkování

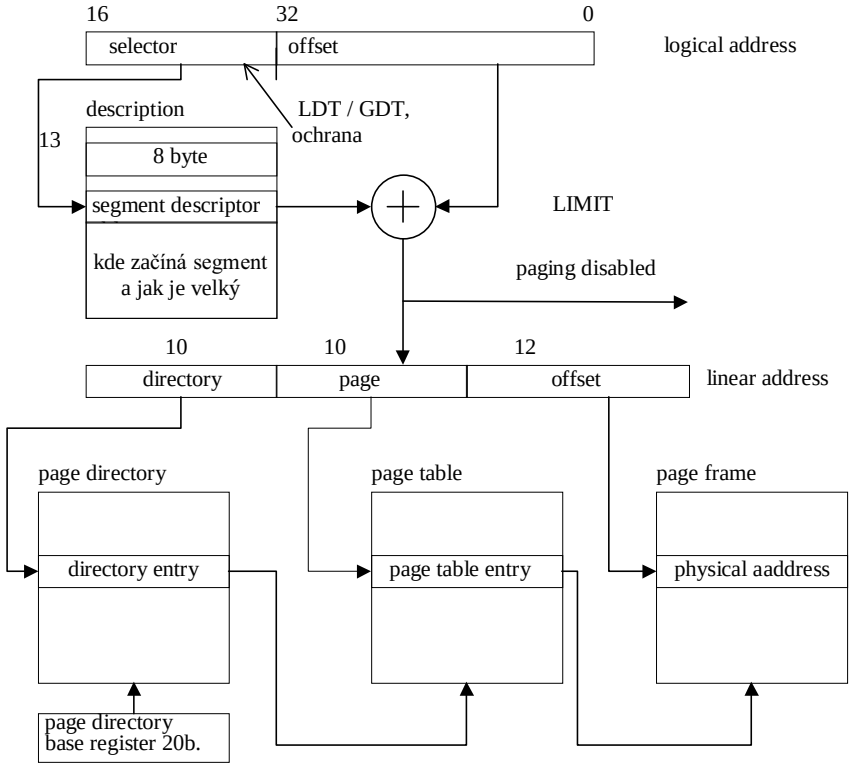


obr. 4: Příklad stránkování

Fyzická adresa má 32 bitů a je vytvářena následujícím způsobem. Select registr ukazuje na příslušnou položku v LDT, GDT. Počáteční adresa a limit se využívají pro generování lineární adresy. Limit se

používá pro ověření platnosti adresy. Pokud adresa není platná vygeneruje se příslušné přerušení. V případě platnosti adresy se připočítá offset k base, čímž získáme lineární 32 bitovou adresu, která se přepočítává na fyzickou adresu.

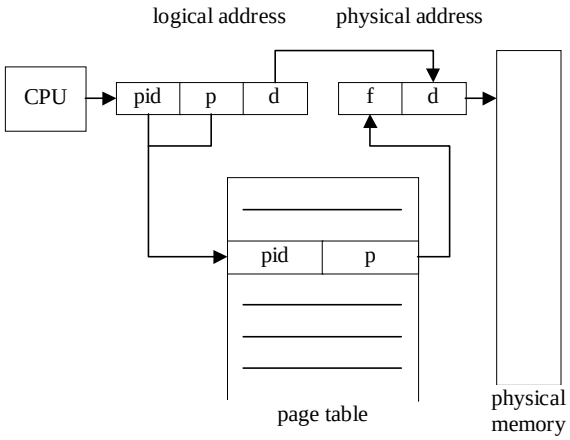
Každý segment je stránkován, stránka má velikost 4 KB. Tabulka stránek může obsahovat až 1 milion položek. Položky jsou tvořeny 4 byty, tudíž každý proces může vyžadovat až 4 MB fyzického adresového prostoru pro vlastní tabulku stránek. Není nutné alokovat tabulku stránek spojitě v operační paměti. Využívá se dvouúrovňové stránkování. Lineární adresa je rozdělena do 20 bitového čísla stránky a 12 bitového offsetu. Pro zavedení stránkování pro tabulku stránek, je číslo stránky dále děleno na 10 bitový ukazatel adresáře stránek a 10 bitový ukazatel do tabulky stránek viz obr. 5. Tabulka stránek může být odložena, potom invalid bit, který se používá v položce adresáře stránek indikuje přítomnost stránky v paměti nebo v dislokovaném prostoru. Jestliže tabulka stránek je odložena, operační systém může použít 31 bitů pro specifikaci umístění tabulky v diskovém prostoru. Tabulka stránek může být umístěna do operační paměti na základě požadavku.



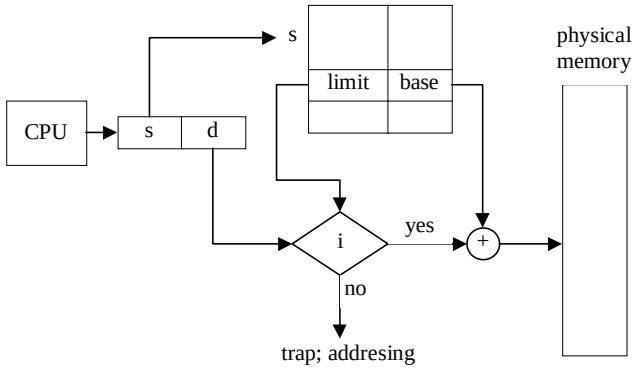
obr. 5: Tvorba fyzické adresy pro architekturu INTEL

Segmentace

Logický adresní prostor se skládá ze segmentů. Adresa = číslo segmentu, offset. Segment = data, kód, zásobník, globální proměnné, lokální proměnné, sdílená paměť, mapování souboru do operační paměti. Adresování je zobrazeno na obr. 6 a obr. 7.



obr. 6: Inverzní tabulka stránek



obr. 7: Technické prostředky pro stránkování

Tabulka segmentů je vytvářena rychlými HW registry – segment base register (STBR), segment table length register (STLR). Ochrana a sdílení paměti je prováděna na úrovni segmentů – datový / kódový segment – problém v programech – špatný index => programové chyby.

Hlediska porovnávání správy paměti

- HW podpora
- Výkonnost HW / SW
- Fragmentace, pevná délka alokace – interní fragmentace, proměnná – externí fragmentace
- Relokace. Externí fragmentace – pakování
- Odkládání, běh více procesů, větších než je velikost operační paměti
- Sdílení kódu a dat mezi procesy, musí být stránkování nebo segmentace
- Ochrana – stránkování nebo segmentace – execute, read, read-write

Alokace paměti

Pro alokaci paměti je možné použít standardních C knihovních funkcí (malloc() apod). Pro alokace ve Windows je však někdy vhodnější použít volání API funkcí GlobalAlloc() / GlobalFree() spolu s funkcemi GlobalLock(),GlobalUnLock(). GlobalAlloc umožňuje definovat parametry alokace paměti – zda má být pamět sdílena, přesunovatelná, vyplněna nulami po alokaci apod. Výsledkem této funkce je handle alokované paměti, před přístupem do takto alokované paměti je ji nutné zamknout funkcí GlobalLock(). Analogicky se postupuje při dealokaci paměti.

Použití tohoto způsobu je někdy nezbytné, neboť některé API funkce jako parametr pro provedení určitých operací požadují handle paměti a ne pointer na ní.

Vypracoval: Roman Mázl, xmazl@lab.felk.cvut.cz, tel. 0603 / 745574