KIV/ZEP - 2011

Správa paměti, úniky paměti.
Práce s příliš velkými daty.

Fyzická a virtuální paměť

- 16-bitové procesory pracují v reálném módu
  - adresa, se kterou se pracuje v programu, má 16 bitový segment a 16 bitový ofset
    - registry CS, DS, ES, SS
    - např. instrukce *rep movsb* přesouvá CX bytů z adresy DS:SI na adresu ES:DI
  - lineární adresa vypočtěna jako segment*16 + ofset
  - teoreticky lze tedy adresovat 1MB paměti
    - sběrnice i286 má 20 bitů
  - lineární adresa odpovídá fyzické adrese paměťového chipu
  - jedna aplikace má přístup do paměti druhé aplikace
    - chybující aplikace má za následek pád celého systému

- chráněný mód
  - poprvé na Intel 80286
  - procesor startuje v reálném módu
  - program (typicky OS) může přepnout procesor z reálného do chráněného módu
    - vyžaduje nastavení různých tabulek (GDT – Global Descriptor Table, LDT – Local DT, IDT – Interrupt DT,...)
  - jakmile v chráněném módu, registry CS, DS, ES, ... obsahují index do GDT nebo LDT
  - položky GDT (případně LDT) tvoří:
    - lineární adresa začátku segmentu (bázová adresa)
    - velikost segmentu v bytech
    - atributy segmentu včetně vyžadovaného oprávnění (RPL)

Fyzická a virtuální paměť

Fyzická a virtuální paměť

- chráněný mód
  - lineární adresa = bázová adresa + ofset
  - lze adresovat 16MB na i286, 4GB na i386+ a mnoho TB na 64-bitovém systému
  - lineární adresa odpovídá fyzické adrese paměťového chipu

- chráněný mód
  - kód běžící v rámci nějakého segmentu smí přistupovat k paměti jiných segmentů jen, pokud jeho oprávnění (CPL) je vyšší než oprávnění RPL přistupovaného segmentu
  - není-li tato podmínka splněna, dojde k přerušení běhu kódu a vyvolání speciální obslužné rutiny (INT13 GP)
    - rutina součástí OS, provede ukončení běhu aplikace, vyhození výjimky apod.
  - rutiny OS jsou odděleny od aplikací

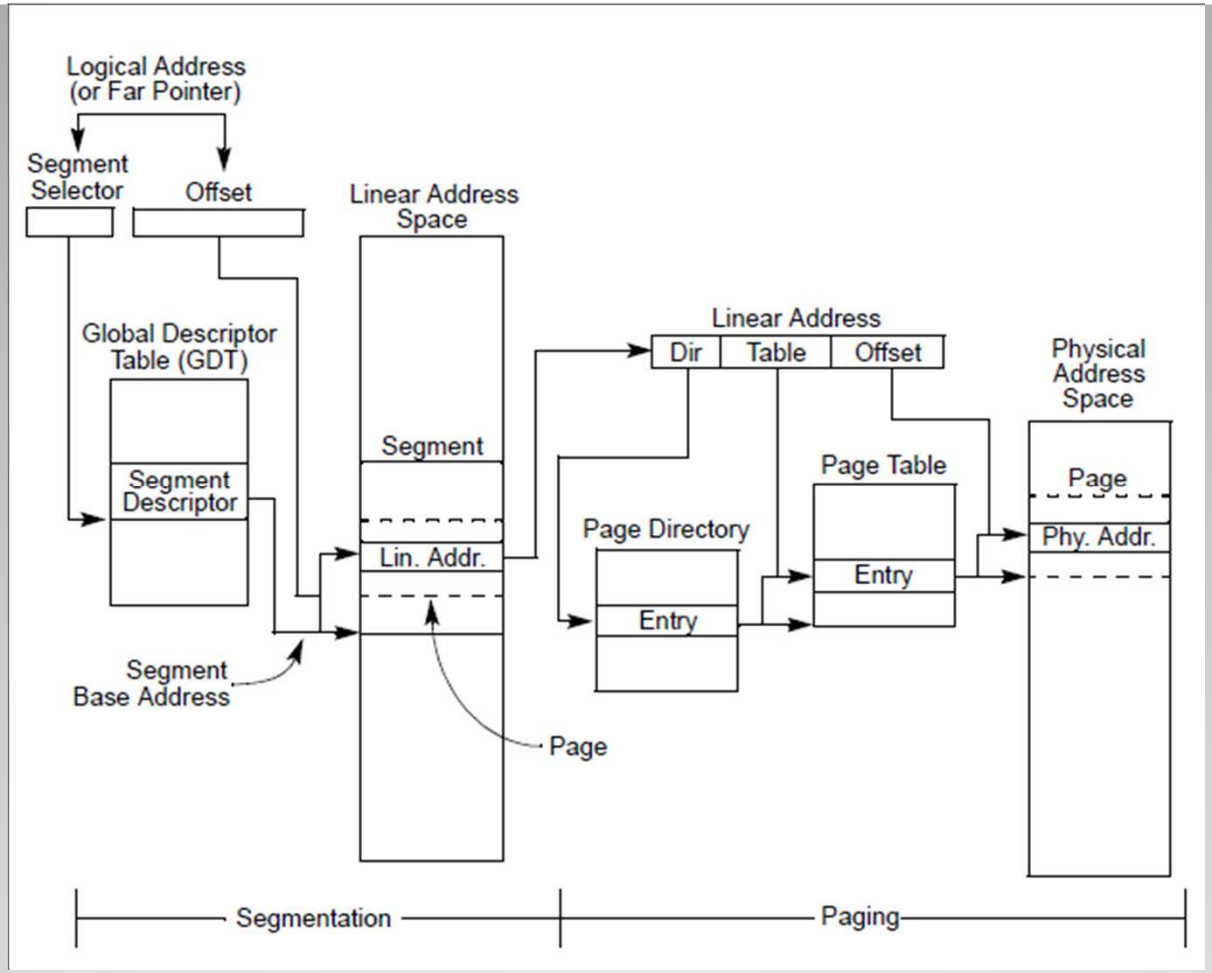
Fyzická a virtuální paměť

Fyzická a virtuální paměť

- virtuální mód
  - počínaje Intel 80386
  - nadstavba nad chráněný mód
  - lineární adresa neodpovídá fyzické adrese
  - lineární adresní prostor mnohem větší než prostor fyzických adres
    - tzv. virtuální paměť
  - prostor lineárních adres rozsekán na stránky
    - stránka = 4KB na většině CPU

Fyzická a virtuální paměť

- virtuální mód
  - pro každou stránku existuje záznam ve speciální tabulce, který obsahuje
    - příznak, zda je stránka mapována do fyzického prostoru paměťového chipu
    - fyzickou adresa začátku stránky (je-li mapována)
    - příznaky zabezpečení (pouze pro čtení, zápis, ...)
  - pokud kód přistupuje k adrese, která je na stránce, která není ve fyzické paměti, CPU zavolá speciální rutinu (INT 14 Page Fault)
    - OS načte stránku ze „swapovacího" souboru
  - poznámka: tabulka může být pro každou aplikaci jiná → dvě aplikace mohou pracovat se stejnou virtuální adresou a přitom každá s jinými daty



Fyzická a virtuální paměť

Fyzická a virtuální paměť

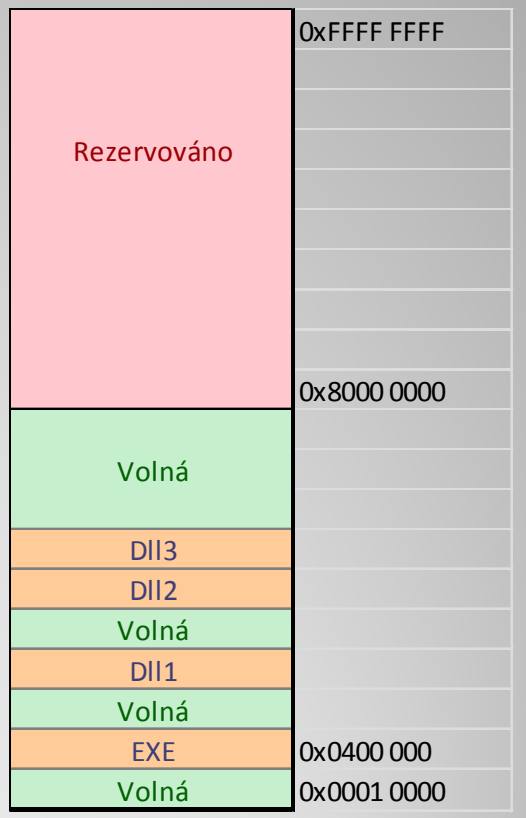
- moderní OS používají virtuální mód
- typicky nastavují tzv. „flat mód“ adres: bázová adresa = 0, velikost segmentu = 4GB (32-bitové CPU) nebo 2^{64} B (64-bitové CPU)
  - výhody
    - všechny aplikace mají stejný prostor virtuálních adres, ty jsou však mapovány na různá místa
    - pracuje se jen s ofsety, tzn. jednodušší a rychlejší kód
  - nevýhody
    - 32-bitové aplikace (naprostá většina) mají k dispozici pouze 4GB virtuální paměti, ačkoliv OS by vlastním stránkovacím způsobem zvádl poskytnout více

Adresní prostor ve Win32

- skutečnost je ještě horší!
- aplikace ve Win32 má k dipozici necelé 2GB, ostatní rezervováno pro systém
  - virtuální adresy 0x00010000 – 0x7FFFFFFF
- do tohoto prostoru se musí vejít kód aplikace a případných DLL knihoven, zásobník (jak aplikace tak DLL) a samozřejmě data
- každá aplikace vyžaduje minimálně jednu DLL knihovnu (kernel32.dll, user32.dll)

Adresní prostor ve Win32

- EXE i DLL knihovny překládány tak, že se zaváděny na nějakou fixní adresu
  - výchozí nastavení je obvykle 0x04000000 pro EXE a 0x10000000 pro DLL knihovny
  - aplikace dokáže alokovat naráz nejvýše tolik paměti, jak veliký je největší volný blok virtuálních adres!
    - často jen 1GB po spuštění



Adresní prostor ve Win64

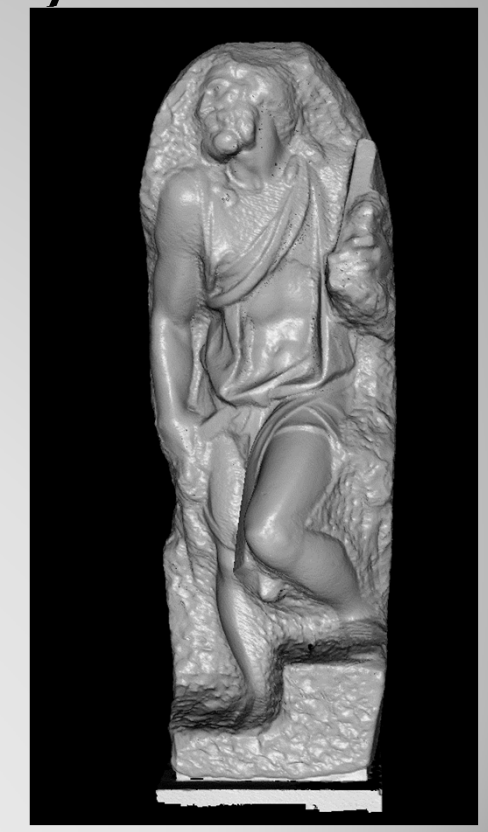
- není-li EXE překládán s parametrem *LARGEADDRESSAWARE*, tak adresní prostor stejný jako ve Win32, i když aplikace je 64-bitová
- jinak má aplikace k dispozici 8TB
  - ale swapovací soubor typicky mnohem menší, takže stejně alokovat tolik nemůže
  - situace mnohem lepší, ale ...

Fragmentace paměti

- většina aplikací pracuje s adresou přímo
  - tj. virtuální adresa začátku pole bytů + 1000 je virtuální adresa 1001. prvku
  - mnohem rychlejší než nepřímý přístup přes indexer nějaké třídy
- problém se časem stupňuje v důsledku fragmentace paměti
  - příklad:512 MB volný prostor, alokuji blok A = 150 MB, alokuji blok B = 250 MB, uvolním blok A, chci alokovat blok C = 200 MB, ale to již nejde, došla paměť!

Rozsáhlá data

- reálná data jsou často rozsáhlá
  - výsledky měření, simulací, musí být přesné
- řešení 1: data-stream algoritmy
  - v paměti vždy pouze malá část dat (ostatní na disku)
  - možný jeden resp. několik málo průchodů
- řešení 2: redukce dat
  - nelze vždy (ztráta přesnosti)



Stanford University Computer Graphics Laboratory: St. Matthew - 186,810,938 bodů, 12 GB (min)

- řešení 3: komprese dat
  - velká režije, nemusí vždy postačovat
- řešení 4:vlastní správa paměti
- řešení 5: použít úspornější algoritmus
  - pozor aplikace může žrát moc paměti kvůli únikům paměti

Rozsáhlá data

Úniky paměti

- pravidlo: paměť alokovaná aplikací musí být dealokována, když již jí není třeba
- není-li paměť uvolňována, dochází k únikům paměti, které mohou vést k pádu celé aplikace nebo zpomalení celého OS
- jazyky bez „garbage collector“ (např. C/C++, Delphi, ...)
  - v kódu musí být zavolání příslušné rutiny pro uvolnění paměti
  - častá chyba: opomenutí volání

Úniky paměti

- jazyky s „garbage collector“ (např. Java, C#, VB, ...)
  - paměť uvolňuje GC
    - pokud na ní není reference
  - **může docházet k únikům?**

- **ANO,** protože
  - GC se volá automaticky často až, když nedokáže obsloužit další požadavek na alokaci
    - příklad: aplikace má k dispozici 1 GB, na začátku své činnosti alokuje 512 MB, pak zruší referenci a dále po dobu několika hodin střídavě alokuje položky o několika KB, tj. po celou dobu žere zbytečně více než 512 MB ve swapovacím souboru
    - řešení: volat GC manuálně
      - zejména pracuje-li se s velkými daty

Úniky paměti

- **ANO,** protože
  - častá chyba:reference není ztracena

```
// Can you spot the "memory leak"?
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }
```

```
    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

Úniky paměti

Úniky paměti

- **ANO,** protože
  - častá chyba:reference není ztracena
  - řešení: nastavovat reference na *null*
- jak poznat, že v programu je únik paměti?
  - Java vyhodí výjimku OutOfMemory, ačkoliv se alokují jen malé bloky
  - specializované utility (např. VLD pro C++)
  - dávat si pozor
- dobré pravidlo: kolekce (hash tabulky, pole, ...) nedělat jako *static*