KIV Operační systémy

IBM PC/AT & MS-DOS(-alike)

Verze 1.0

80286

- 1. ledna 1982
- 16-bitový procesor
  - Instruční sada x86-16
- První z rodiny x86 procesorů, který měl protected-mode používaný k izolaci jádra od uživatelských procesů

- Nicméně procesor po zapnutí nastartuje v tzv. real-mode, tj. stejně jako 8086 z roku 1978
  - Díky zachování zpětné kompatibility
    stejně jako dnešní nejmodernější x86-64

80186

- Vrstevník 80286, také vyráběný od roku 1982
  - Procesor pro vestavná zařízení
  - Instrukční sada x86-16
- Sice se už nevyrábí, ale zato se dodnes vyrábí RISCové procesory, které mají jeho instrukční sadu
  - HTL80186
  - VAutomation Turbo186
  - CAST C80186EC
  - Mentor Graphics M8086
  - iWave Systems 80186XL

80286 – vybrané registry

- AX, BX, CX a DX – obecné 16-bitové registry, které se dále dělí na dvojice 8-bitových registrů
  - AH, AL … DH, DL
- CS, DS, SS, ES – segmentové registry pro kód, data, zásobník a extra segmentový registr
- SI, DI, BP, SP – indexové registry; source, destination, base, stack
- Flags – stavový registr, výsledky operací
- MSW (386+: CR0-4, atd.) – stav CPU

80286 – adresace

- Adresa v paměti ji dána dvojicí registrů segment:index
  - Např. CS:IP ukazuje na instrukci, která se má vykonat
- Adresa má 20 bitů
  - Segment má 16 bitů
  - Index, tj. offset, má také 16 bitů
  - Adresa = (segment shl 4) or offset

	1100 1100 1100 1100	segment
+	0011 0011 0011 0011	offset
	1111 1111 1111 1111 0011	adresa

80286 – mapa paměti

Adresa	Popis
0000:0000	Tabulka vektorů přerušení; 256 4-bytových adres
0040:0000	Proměnné ROM-BIOSu
A000:0000	Paměť EGA; grafický režim obrazovky
B000:0000	Paměť MDA
B800:0000	Paměť CGA; textový režim obrazovky
C800:0000 až E000:0000	Externí ROM, ROM-BIOS v ní hledá spustitelný kód – ovladače zařízení
E000:0000 až FE00:000	ROM-BIOS: POST, defaultní obsluhy přerušení, SETUP, diagnostika, atd.
F000:FFF0	Instrukce JMP, která se skočí na první instrukci, která se provede po zapnutí/resetu procesoru
F000:FFF5	Datum verze BIOSu
F000:FFFE	Identifikační kód PC

MS-DOS

- Disk Operating System
- Jeden z nejvýznamnějších (tj. ne nutně nejlepších) operačních systémů
- Poskytuje konzoli a souborový systém
  - Další je záležitost ovladačů a náhrady/nástavby „shellu“ – např. Windows 3.1
- Existovalo mnoho DOSů
  - MS-DOS, PC-DOS, DR-DOS, QDOS…
  - MSDOS.SYS, IO.SYS a COMMAND.COM se pak jmenují jinak
  - A dodnes aktivní FreeDOS

80286 – MS-DOS Bootstrap

1. Po zapnutí/hw resetu (tj. hot reset počítače) se procesor uvede do aktivního stavu a do režimu real-mode, tzn. má přístup pouze k 1MB paměti a kód může číst a zapisovat na libovolné místo v paměti
2. Registry CS:IP se nastaví na hodnoty F000:FFF0 a CPU začne vykonávat instrukce od této adresy
   - Tj. skočí na první instrukci ROM-BIOSu, čímž spustí jeho kód – proto se této adrese říká reset vector
   - Cold reset počítače: předání se řízení na adresu určenou reset-vectorem; od 386 je to fyz. lineární adr. 0xFFFFFFF0

80286 – MS-DOS Bootstrap

3. Podle nakonfigurovaného pořadí BIOS hledá disky
4. Při nalezení prvního disku BIOS načte do paměti, adresa 0x7c00, jeho první sektor, tj. prvních 512 bytů, předá řízení CPU na tuto adresu – tj. nastaví CS:IP
   - První sektor se nazývá Master Boot Record (MBR)

Velikost	Popis
440B	Kód
4 B	Signatura disku
2B	0x00 0x00 -
4x16 B	Tabulka rozdělení disku
2B	Signatura MBR

80286 – MS-DOS Bootstrap

5. Kód načtený z MBR má za úkol načíst zbytek zavaděče ze správného/aktivního/vybraného oddílu disku
   - Může být zavaděč přímo operačního systému
   - Anebo manažer, který dá vybrat, který OS se má zavést, je-li jich nainstalováno více
   - Anebo to také může být vir, který infikuje počítač ještě před načtením OS
   - MS-DOS startuje z FAT, active&bootable&primary oddílu

80286 – MS-DOS Bootstrap

6. Poté, co kód z MBR identifikoval použitelný diskový oddíl, pokračuje s načítáním OS do paměti
   - Jedná se o soubory IO.SYS a MSDOS.SYS, které musí být uloženy kontinuálně na začátku oddílu
7. Jakmile jsou sobory načteny, řízení převezme IO.SYS
   - Rozhraní mezi DOSem a I/O subsystémem, které zpřístupní základní periferie
8. IO.SYS předá řízení MSDOS.SYS
   - Jádro OS, které poskytuje abstrakci od HW pomocí poskytovaných služeb

80286 – MS-DOS Bootstrap

9. Pokud existuje, MSDOS.SYS načte a zparsuje CONFIG.SYS
   - Zavede ovladače paměti (XMS, EMS), periferií (CDROM, Myš, zvuková karta, atd.)…
10. MS-DOS.SYS načte a spustí, tj. předá řízení, COMMAND.COM (interpret příkazů, aneb shell)
11. Pokud existuje, spustí se AUTOEXEC.BAT
    - Dávkový soubor s příkazy pro COMMAND.COM
12. C:\ aneb Hotovo!

Windows -9X Bootstrap

- 16 bitové Windows se spouštěly příkazem win.com
  - Zavaděč Windows/386 přepnul procesor do tzv. protected-mode
- Windows 9x se zaváděly tak, že IO.SYS provedl konfiguraci počítače v real-mode a pak spustil win.com po dokončení AUTOEXEC.BAT
  - Komplikované zavedení ovladačů, některá zařízení mají ovladače jen pro real-mode, zatímco Windows běží v protected-mode => potřeba virtualizace – V86 mode

Jádro větší než 1MB

- Např. Linuxové jádro může být větší než 1MB
- Jenomže CPU nastartuje v real-mode s 20-bitovou adresou
- Jak zavést tak velké jádro?
  - 386+ procesory lze přepnout do tzv. unreal-mode, který zpřístupní dostupnou paměť jako 4GB flat address space
    - A pak lze načíst a spustit jádro větší než 1MB
  - Anebo máme UEFI – což nikdy nebyl případ MS-DOSu
    - Ale stále existuje např. FreeDOS

Unreal mode

- Není to oficiálně garantovaná vlastnost, ale existuje protože ji potřebuje System Management Mode
- Aby mohly programy běžet v protected-mode (32-bitová adresa, segmentace, stránkování, izolace procesů), je třeba vytvořit GDT a LDT
  - Globální a lokální tabulky deskriptorů segmentů – organizace paměti
- Segmentům se nastaví maximální velikost (limity) a procesor se přepne zpět do real-mode
  - Respektive do unreal-mode protože limity zůstanou zachovány

Příliš velký program (DOS)

- Co když máme program pro real-mode, který ale vyžaduje více paměti, než kolik jí je volné?
  - XMS, EMS – paměťové manažery, které přepnuly procesor do protected mode a na rozdíl od unreal mode v něm zůstaly
  - Real-mode programům, pak umožnily využít větší paměť po max 64kB velkých oknech, které kopírovaly mezi adresami nad a pod 1MB
- Programy pak dynamicky překrývaly blok paměti různými částmi, moduly, svého kódu => tzv. overlays

Overlays

- Technika, která umožňuje spustit program, který je větší než velikost dostupné paměti
  - Předchůdce virtuální paměti (příští přednáška)
  - Stále se používá u vestavěných zařízení, kde virtuální paměť není k dispozici
- Program se rozdělí do funkčních modulů, overlays, které mají stromovou strukturu
  - V paměti mohou být zavedeny jenom moduly od na cestě kořene, tj. fce main, až k listům
  - Návrh stromů a explicitní zavádění a uvolňování musí zařídit programátor

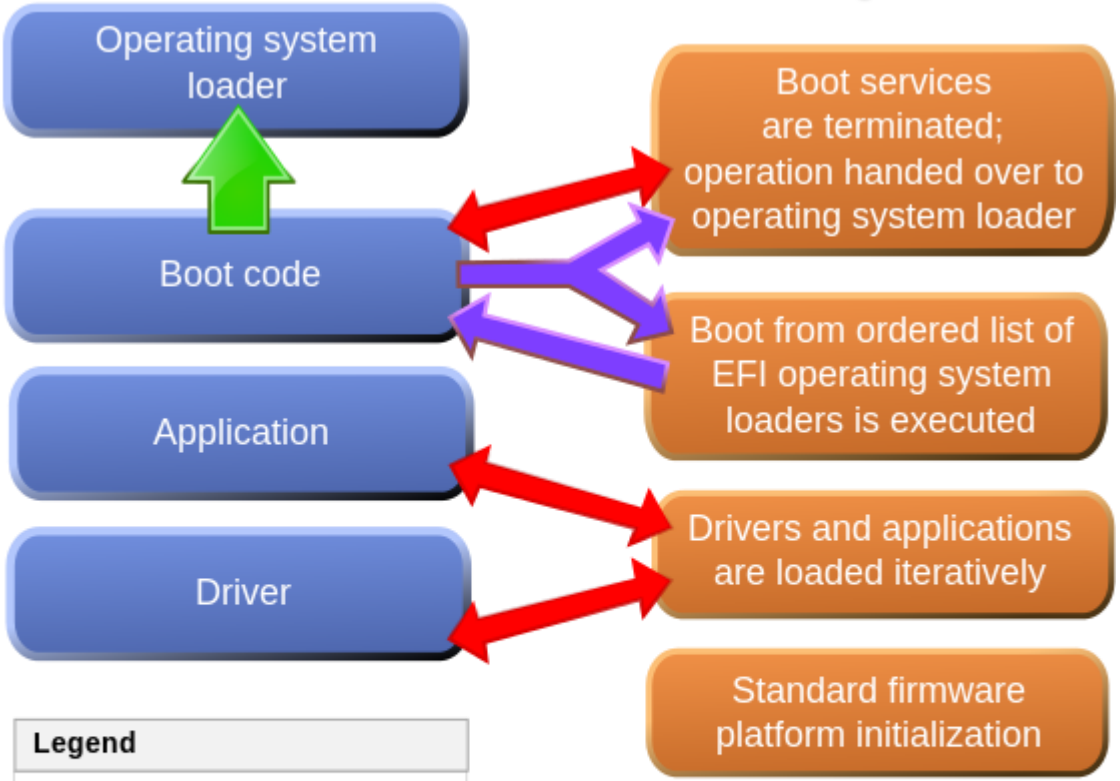
UEFI

- Unified Extensible Firmware Interface
- Nástupce BIOSu, který má řešit jeho nedostatky
- Nespoléhá se na boot (tj. první) sektor, ale definuje boot manager
  - Umí zavádět pouze důvěryhodně podepsaný kód – což nejsou např. viry – BIOS uměl zamezit přepsání MBR
  - Nicméně má legacy mode, ve kterém se chová jako BIOS
- Umí přepnout procesor do cílového režimu, např. protected-mode – zavaděč OS to ale musí očekávat

UEFI - pojmy

- EFI executable – spustitelný soubor v UEFI bytecodu (ne Java bytecode)!
- Guid Partition Table (GPT) – načisto udělaná tabulka diskových oddílů, tak aby se nemusel vláčet omezující balast z minulosti
- EFI System Partition – oddíl naformátovaný souborovým systémem FAT dle specifikace UEFI
- Umíme načíst a spustit soubor z disku – tj. stejný princip jako doposud, jen jiná, novější specifikace

UEFI - Bootstrap



- UEFI načítá ovladače stejně jako to dělal ROM-BIOS a IO.SYS

http://en.wikipedia.org/wiki/File:Efi_flowchart_extended.svg

MS-DOS API

- Chce-li program zavolat službu operačního systému, de-facto volá požadovanou rutinu, která je už někde v paměti – ale jak ji najde?
- Řešením je, aby byla na předem známé adrese, kam se předá řízení procesoru nastavením CS:IP
- Rutin implementujících jednotlivé služby může být mnoho => použije se další registr, např. ax na určení konkrétní služby

MS-DOS volání API

- Služba MS-DOS se zavolá pomocí přerušení 21h
  - Tzn. adresa hlavní rutiny služeb OS je zapsána na 0x21. pozici tabulky vektorů přerušení
  - Např. zjištění verze DOSu vypadá následovně

```
mov ah, 0x30h
int 21h
;po návratu jsou hlavní a vedlejší čísla verze v AL a AH
```

MS-DOS obsluha API

1. Program nastaví příslušné registry vykoná int 21h
2. Do zásobníku, na jehož vrchol ukazuje SS:SP, se uloží registry CS:IP (ukazující ve volajícím programu na další instrukci po int 21h) a registr Flags
   - Provede procesor v rámci zpracování instrukce int 21h
3. Jádro OS získá kontrolu nad CPU a vidí všechny registry volajícího programu
4. Jádro OS provede příslušnou akci a nastaví příslušní registry podle výsledku akce

EGA-BIOS

- Co když budeme chtít využít služeb, které neposkytuje BIOS, ale např. grafický adaptér?
- Např. budeme chtít změnit režim obrazovky na 320x200x256. Program vykoná následující instrukce

```
mov ah, 0       ;služba Změň videorežim
mov al, 13h     ;režim 320x200 256 barev
int 10h
```

- Z hlediska procesoru se stane to samé jako při instrukci int 21h, pouze vykonávaný kód bude v paměti někde jinde

Zápis do video paměti

- Na adrese a000:0000 je první pixel právě nastaveného videorežimu, levý horní roh
- Na každý pixel je jeden byte, byte je index do palety barev
- Přímým zápisem do namapované videopaměti měníme barvy jednotlivých pixelů
- V textovém režimu je znak v levém horním rohu na b800:0000
  - Má dva byty – 1. byte kód znaku, 2. byte atributy, např. barva

VESA

- Videorežim 320x200x256 potřebuje méně než 64kB na uložení indexů pixelů do palety barev
- Videorežim 640x480x16 také
- Ale videorežim 640x480x256 už ne – navíc to dříve nebýval standardní režim, dokud se neobjevila specifikace VESA - SuperVGA
- Před VESA sice bylo možné na některých kartách takový režim aktivovat, ale postup byl proprietární
- Proto vznikl rezidentní software, který VESA emuloval

VESA - stránkování

- Video režim 1280x1024 při 24-bitové barevné hloubce potřebuje více než 64kB paměti
- Řešením bylo stránkování
  - Obraz se rozdělil na několik stránek po 64kB
  - 64kB od A000:0000 umožňovalo přímý zápis a čtení do aktivní stránky
- Aktivní stránka se zvolí funkcí VESA BIOSu
  - Buď pomocí int 10h – opakované volání int 10h je pomalé
  - Anebo call na konkrétní adresu obslužné rutiny; adresu získáme přes int 10h – call je daleko rychlejší než int, který je potřeba pouze jednou

VESA – Linear Frame Buffer

- I když je stránkování video paměti pomocí call rychlejší než pomocí int, je stále pomalé
- Rychlejší je přímý přístup do video paměti, jako tomu bylo u videorežimů, kterým stačilo 64kB
- VESA umožňuje získat adresu Linear Frame Buffer (LFB)
  - Obdoba A000:0000 u EGA, ale adresa LFB není pevně daná
  - Je třeba LFB „povolit“ a následně získat adresu od grafické karty
  - Blok paměti, který nelze použít pro data a kód programů

Zřetězení obsluhy přerušení

- Např. chceme-li sw emulovat VESA

1. Program si do vlastní proměnné načte adresu stávajícího vektoru přerušení – int 10h u sw VESA
2. Program zapíše do tabulky vektorů přerušení adresu své rutiny, která bude přerušení nově obsluhovat
   - Např. u sw VESA obsluhuje služby s AH=4Fh (VESA extension)
   - Nová rutina přerušení může, případně musí (např. int 08h hodiny), zavolat (už ne int!) předcházející obsluhu přerušení
3. Program skončí službou OS Terminate and Stay Resident (TSR)

Ukončení TSR

- TSR program měl smysl pouze tehdy, pokud obsluhoval některé přerušení
- V paměti mohl být načteno několik TSR obsluhujících stejné přerušení
- Ale co když se měl jeden z nich ukončit, a nebyl to ten poslední?
  - Byl to problém, protože neexistoval standardizovaný protokol, jak vyjmout ze zřetězeného seznamu obsluh přerušení někoho uprostřed

Interrupt request (IRQ)

- Instrukce int je sw-vyvolané přerušení
- Pokud přerušení vyvolá hw, pak se bavíme o IRQ
  - Každé IRQ má svoji prioritu – Level aka IRQL
- Při IRQ procesor zastaví vykonávání aktuálního programu, uloží Flags, CS a IP, a začne vykonávat příslušenou obsluhu přerušení dle tabulky vektorů přerušení
- Programmable Interrupt Controller (PIC) mj. překládá číslo IRQ na index do tabulky vektorů přerušení

Časovač

- Např. tik hodin je IRQ0 (nelze změnit ani maskovat) a vyvolá obsluhu přerušení int 08h
  - Proběhne každých 55ms
- Na tomto přerušení závisí mnoho důležitých činností a je proto nutné, aby
  a) Bylo co nejrychlejší
  b) Zavolalo původní obsluhu přerušení
- Pomocí časovače se implementuje preemptivní multithreading (a následně multitasking)

Časovač - implementace

- Nejprve se do proměnné oldVec8 uloží adresa původní obsluhy přerušení, takže obsluha může vypadat následovně:

```
pushf                              ;simulace volání obsluhy přerušení
call dword ptr cs:[oldVec8] ;pro původní obsluhu
…                                  ;naše vlastní činnost
iret                               ;návrat do přerušeného programu
```

I/O Porty

- Input/Ouput base address – adresa prvního portu
- Periferie lze také ovládat pomocí portů – jedno zda blikáme s LED klávesnice, nebo programujeme PIC
  - Zápis odešle příkaz; instrukce out
  - Čtení čte stav nebo výsledek operace; instrukce in
- Např. při obsluze časově závislých činností v int 08h je nutné poslat řadiči přerušení informaci, že přerušení již skončilo

```
mov al, 20h          ;signál Konec přerušení
out 20h, al          ;port řadiče přerušení 8259
```

Viry

- MS-DOS umožnil ovládat počítač a jeho periferie
- Ale špatně nebo záměrně napsaný program mohl číst a přepisovat jeho vnitřní proměnné, a libovolně měnit činnost systému
  - Např. na přerušení se mohl pověsit vir, který se spouštěl z infikovaného MBR disku a infikoval MBR disket, a díky zavedení před jádrem OS mohl utajit své soubory na disku filtrováním systémových volání
    - Antiviry musely skenovat paměť, což byla příležitost pro polymorfní viry, které měnily svůj kód v paměti za běhu

KIV Operační systémy

Režim jádra a uživatelský režim

Verze 1.0

Chceme, aby…

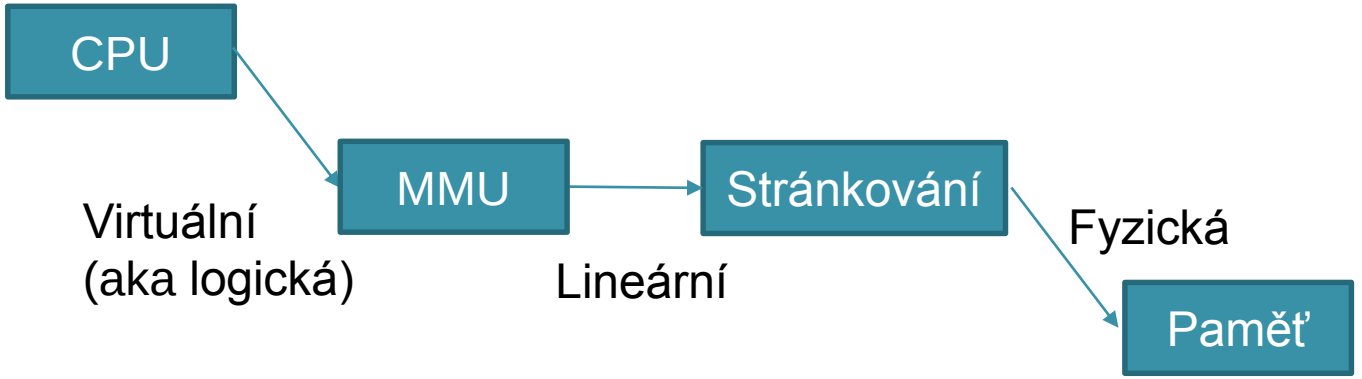
- ..mohlo zároveň běžet několik procesů
- ..proces nemusel vědět, že zároveň s ním běží další procesy
- ..bylo jedno, kde v paměti běží který proces
- ..proces nemohl přistupovat do paměti jiného procesu či jádra
- ..výsledkem nebylo zpomalení počítače
- ..výsledkem bylo zefektivnění práce na počítači

Memory Management Unit

- Výše uvedené cíle implikují, že procesy budou pracovat se svým formátem (tj. virtuální) adresy do paměti, která bude následně převedena na fyzickou adresu do paměti
- Protože sw implementovaný převod by byl pomalý, převod virtuální na fyzickou adresu obstará hw
  - Konkrétně Memory Management Unit (MMU)
  - Detaily převodu zadá MMU přímo OS

segment:offset

- x86 pracuje s virtuální adresou ve formátu segment:offset
- Z ní je třeba získat tzv. lineární adresu, a teprve tu lze převést na fyzickou adresu v režimu, ve kterém dokážeme od sebe izolovat jednotlivé procesy a jádro



Protected mode

- V prvé řadě musíme zabránit uživatelským procesům, aby nemohly modifikovat kód a data jádra
- Jelikož x86 adresuje pomocí segmentu a offsetu (vůči segmentu), řešením bylo připojit dodatečné informace ke konkrétním segmentům
  - => už nepracujeme se segmentem, ale se segment deskriptorem
  - Procesor běží v tzv. Protected mode

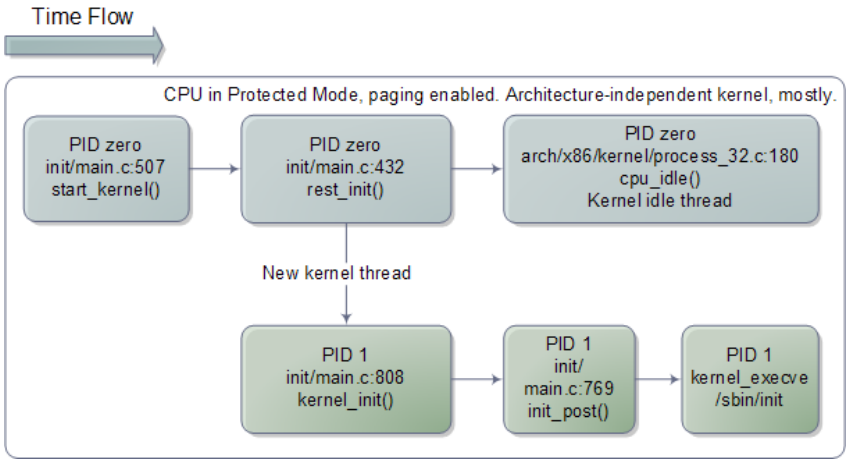
Protected mode - přepnutí

- Poprvé ho implementoval 80286, komplikované přepnutí
- 80386 ho rozšířila a zjednodušila jeho přepnutí

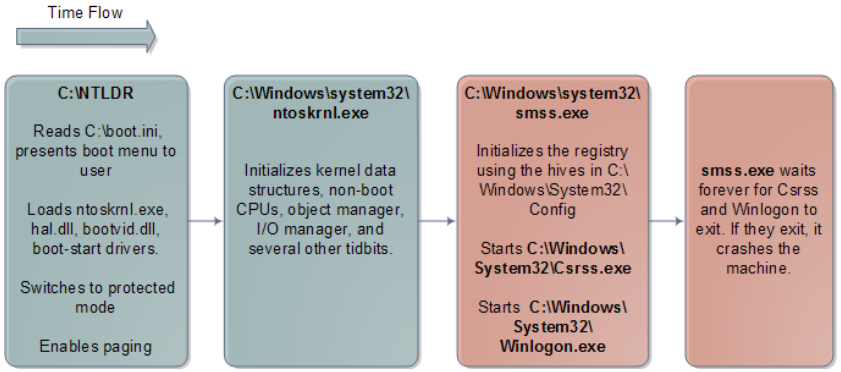
```
EnablePM:
        cli                     ;zamaskování přerušení
        mov eax, cr0
        bts eax, 0              ;přepneme do
        mov cr0, eax            ;chráněného režimu
        lgdt [newGDT]           ;nastavíme selektory
        lidt [newIDT]           ;přerušení a vyjímky
        mov eax, selSS
        mov ss, eax             ;zásobník
        ljmp selCS, @pm         ;nastav segment selektor kódu:EIP
pm:
        ret                     ;a vrátíme se v pm
```

Protected mode – zavedení OS

- Linux



- Windows NT

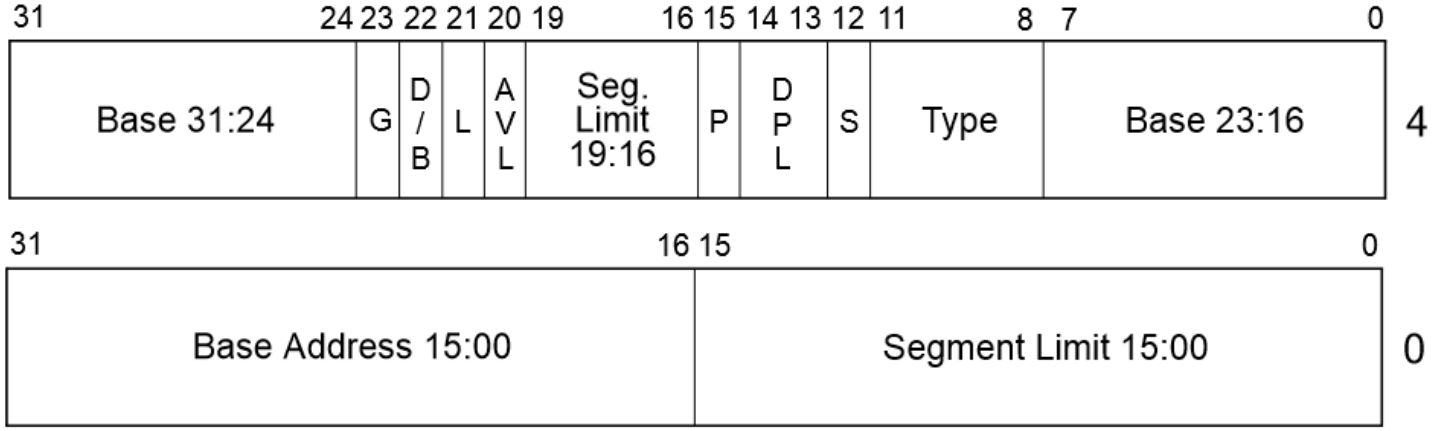


http://duartes.org/gustavo/blog/post/kernel-boot-process/

Segment Descriptor

- Segment má:
  - Základní (base) adresu (lineární adresa)
  - Velikost (limit)
  - Typ (kód, data, atd.)
  - Přístupová práva
  - Další vlajky

Segment Descriptor



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1

Izolace jádra

- Pro izolaci je důležitá hodnota bitů CPL/DPL
- Číslo oprávnění/privilege level aktuálního kódu
  - 00 - většinou jádro
  - 01 –může být jádro, když na 00 poběží hypervizor virtualizace
  - 10 – může být ovladač, který nesmí do jádra
  - 11 – většinou uživatelský proces
- Kód s nižším číslem oprávnění může přistupovat do segmentu s vyšším číslem oprávnění
- Opačně to nelze => izolace jádra od uživ. procesů

Izolace procesů

- Před vlastní inicializací protected mode musí jádro OS vytvořit tabulku deskriptorů segmentů
- Tabulky existují dvě
  - Globální používaná jádrem
    - uložena v registru gdtr privilegovanou instrukcí lgdt
  - Lokální používaná konkrétním jedním procesem
    - Uložena v registru ldtr privilegovanou instrukcí lldt
    - Tj. při přepnutí kontextu může lldt vykonat pouze jádro s CPL=0 a tudíž si uživatelský proces nemůže měnit jemu přidělenou paměť

Segment deskriptory v kódu

- Uživatelský proces může mít až 6 segmentů

  s deskriptory uloženými v cs, ds, ss, es, fs a gs
  - V 64-bitovém long-mode jsou nulové až na fs a gs
- Adresa v paměti má prefix požadovaného segmentu
  - call cs:[adresa_funkce]
  - mov eax, ds:[adresa_retezce]
  - mov ecx, ss:[ebp-offset_parametru_funkce]
    - V ebp bývá hodnota esp při volání funkce, tzv. frame pointer

Výhody segmentace

- Izoluje procesy a jádro
- Lze sdílet segmenty – např. read-only programový kód sdílených knihoven
- Lze relokovat i pouze jeden segment
- Není nutné alokovat nevyužitou paměť
- Tabulka deskriptorů se vejde do MMU

Nevýhody segmentace

- Segmenty mohou mít různou délku => jak je poskládat do paměti?
- Segmenty mohou být velké => fragmentace paměti
- Jak efektivně implementovat sdílenou paměť mezi procesy?
- Jak efektivně odkládat paměť z RAM na disk, abychom zvýšili celkovou dostupnou paměť počítače?

Stránkování

- Odstraňuje nevýhody segmentace
  - x86 umožňuje kombinovat segmentaci a stránkování
  - Nelze povolit stránkování bez protected mode
  - x86 ji umožňuje použít v protected (32-bit) a long (64-bit) mode
- Celá paměť se rozdělí do stránek o pevné velikosti
  - 4kB, 2MB, 4MB a 1GB
  - Virtuální stránka: page (o velikosti frame)
  - Fyzická stránka: frame (o velikosti page)

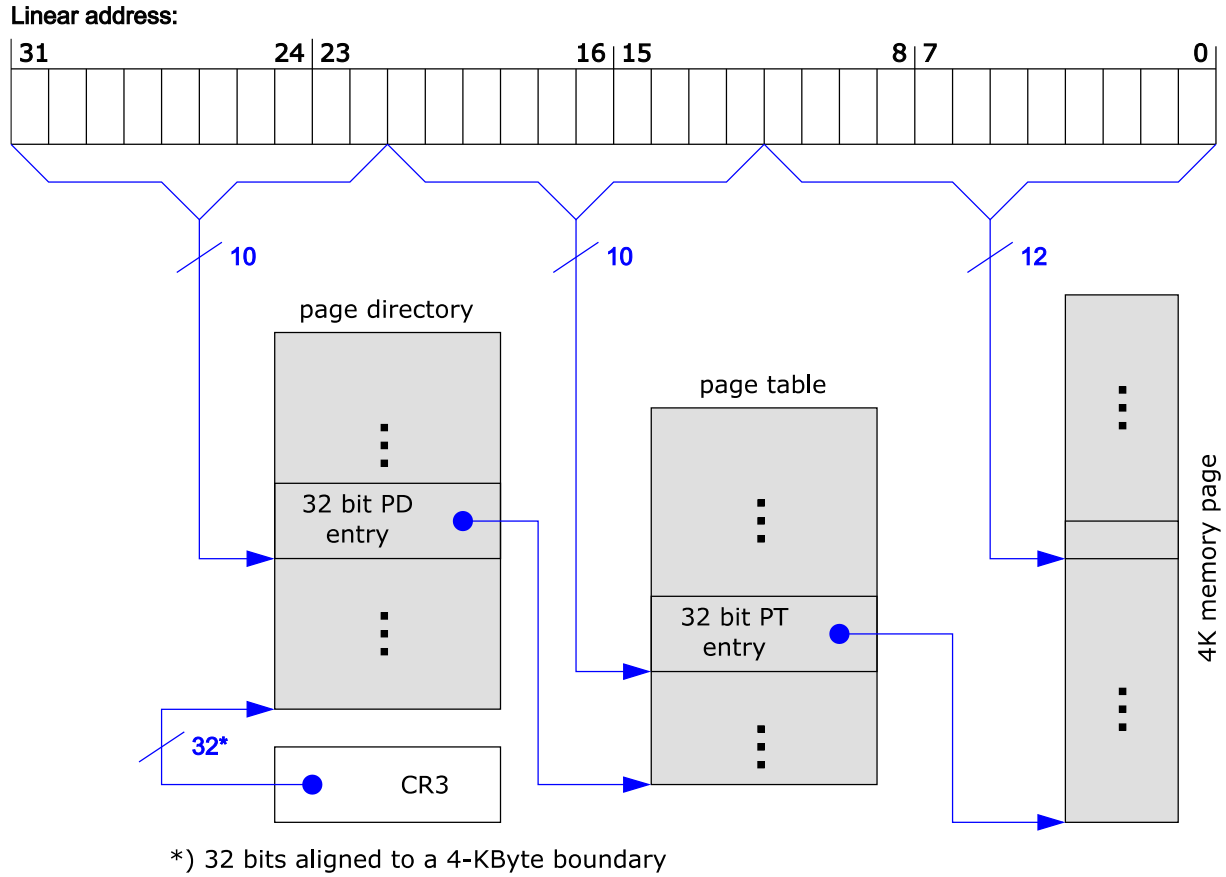
Page Table

- Jádro OS vytvoří tabulku stránek
  - Řídící registr (x86) cr3 ukazuje na tuto tabulku stránek
  - Pouze jádro s CPL=0 může měnit obsah cr3
    - Tj. dosáhneme stejného efektu jako s GDT a LDT
    - Procesor nastaví cr3 při přepnutí kontextu
- Každá stránka má své vlajky, mj. zahrnující
  - Jádro vs. Uživatelský proces
  - Writeable, NX – do not execute
  - Present (zda je fyzicky v RAM), Dirty a Accessed

Hierarchická Page Table

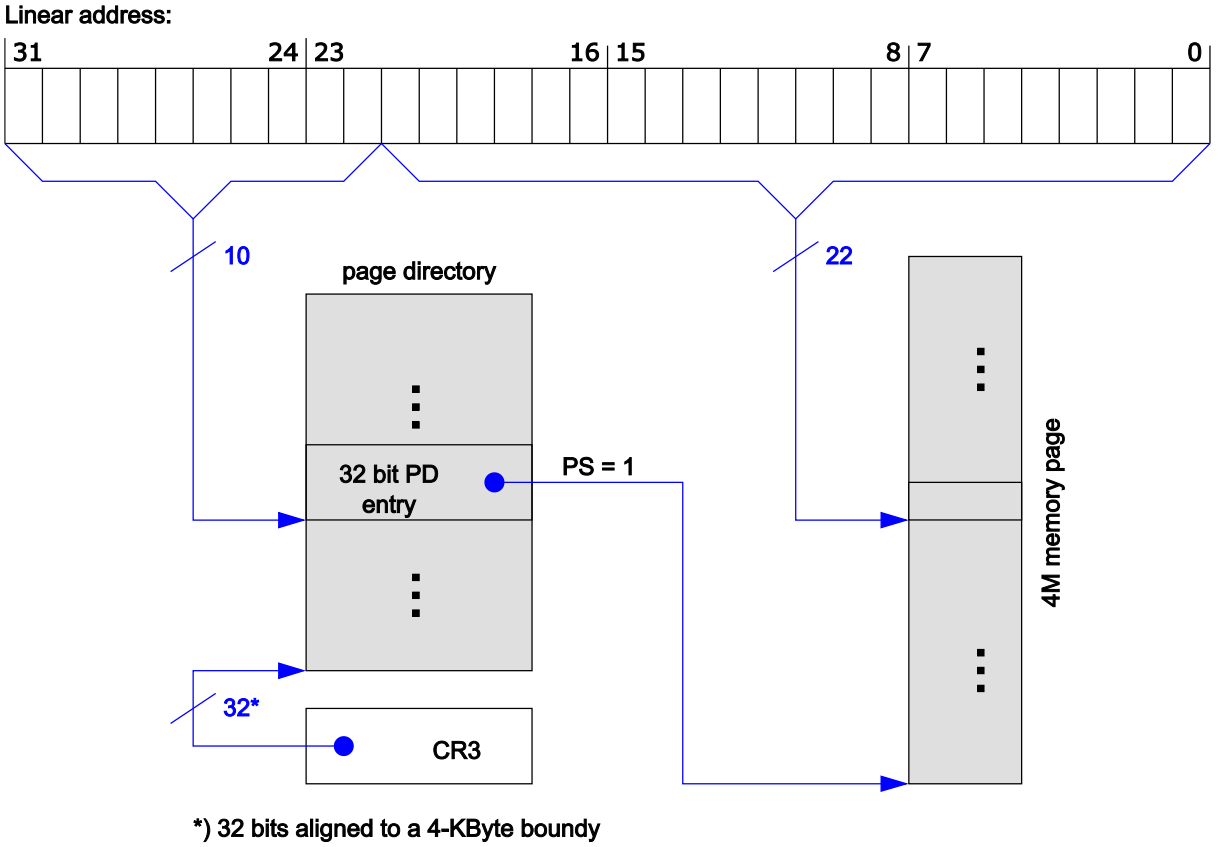
- Jenom několik málo procesů využije celý dostupný paměťový prostor, např. 2^32
- Řešením je vytvořit vnější/nadřízenou/page directory tabulku stránek, která bude odkazovat na další tabulku stránek
  - Paměťový prostor procesu pak nemusí být spojitý ve smyslu, ale mohou v něm být díry na místech, které program nepoužívá

Stránkování 4kB



http://en.wikipedia.org/wiki/Physical_Address_Extension

Stránkování 4MB



http://en.wikipedia.org/wiki/Physical_Address_Extension

Translation look-aside buffer

- Běžící proces často potřebuje jenom omezené množství stránek
  - Jak tedy zrychlit převod virtuální adresy na fyzickou?
  - Pomocí asociativní pamětí Translation look-aside buffer (TLB)
    - Je rychlá, ale také je malá

Page index	Frame index

Stránkování s TLB

- Běžící proces často potřebuje jenom omezené množství stránek
  - Jak tedy zrychlit převod virtuální adresy na fyzickou?
  - Pomocí asociativní pamětí Translataion look-aside buffer (TLB)
    - Musí se vyprázdnit při změně kontextu
    - Je rychlá, ale také je malá
      - Větší hit rate když je větší, ale také je pak pomalejší
    - Má zásadní vliv na výkon

Page index	Frame index

Sdílená paměť

- Efektivní cesta jak sdílet data mezi různými procesy
- Procesy jsou zodpovědné za konsistenci dat ve sdílené paměti
- OS pouze zajistí sdílení paměti
- Do tabulky stránek procesu OS přidá page entry, která ukazuje na ten samý rámec (fyzická stránka) jako page entries v tabulkách stránek ostatních procesů, které tímto „trikem“ sdílejí tu samou paměť

Sdílení kódu

- Pokud několik procesů používá sdílený programový kód, proč ho do paměti nahrávat vícekrát?
- Příslušné stránky s kódem se označí jako read-only a namapují se do paměťových prostorů příslušných procesů
  - Tj. stále jde o sdílenou paměť
- Příkladem jsou dynamické knihovny
  - dll
  - so

Copy on write

- Co když několik procesů na začátku sdílí stejná data, která považují za privátní, ale mění je jenom zřídka?

1. Příslušná stránka se označí jako read-only
   - Dokud z ní proces jenom čte, nic se neděje
2. Jakmile se proces pokusí zapsat do read-only stránky, procesor vygeneruje vyjímku, kterou zachytí OS
3. OS pak alokuje novou stránku, zkopíruje do ní původní read-only stránku, a aktualizuje tabulku stránek procesu
4. Po návratu z obsluhy proces normálně zapíše data už do nové stránky, aniž by o něčem vůbec věděl

Swapování

- Chceme-li poskytnout procesům více paměti, než kolik máme fyzicky RAM, nezbývá než paměť z RAM dočasně odložit do jiné paměti – flash nebo pevný disk
- Working Set – dynamická množina stránek, které má proces aktuálně ve fyzické paměti
- Chce-li dát OS více paměti některému procesu a není volný rámec, zmenší Working Set jiného procesu odebráním stránky, přičemž uloží obsah rámce na disk

Page Fault

1. Proces chce číst data ze stránky, která není v RAM
2. MMU nedokáže přeložit virtuální adresu na fyzickou adresu
   - => vygeneruje vyjímku PageFault
3. OS uloží rámec některé jiné stránky, třeba i jiného procesu, na disk a načte do něj stránku požadovanou aktuálním procesem
4. OS příslušně upraví tabulky stránek dotčených procesů
5. Po dokončení proces dál normálně pokračuje aniž by něco poznal

Page Fault - následky

- Ve skutečnosti OS nebude měnit working sets při každém Page Fault
  - Namísto toho se požadavky mohou nasbírat a vyřídit později – hromadně
  - Page Fault-ovaný proces se mezitím může pozastavit
- Swapování má extrémně negativní vliv na výkon systému
  - Zaměstnává procesor, disk a příslušnou i/o sběrnici

Mapování souborů

- Chceme-li zrychlit práci se souborem, je možné OS požádat o namapování souboru do paměti
- Proces pak zapisuje a čte přímo z rychlé paměti, namísto práce s pomalejším diskem
- OS namapuje soubor do paměti po stránkách jako u swap souboru
- Při ukončení práce se souborem se pak na disk zapíší pouze ty stránky, které mají nastavený dirty bit
  - Pokud by měl soubor např. několik GB, proč zapisovat vše?

Změna privilege level

- Uživatelský proces běží ve svém paměťovém prostoru
- Pokud volá funkci jádra operačního systému, které vyžaduje vyšší úroveň oprávnění, např. CPL=0, je nutné nějak změnit privilege level
- Dělá to procesor v okamžiku, kdy obsluhuje přerušení ať už int nebo IRQ
  - Úroveň oprávnění určí z adresy/vektoru obsluhy přerušení
  - Obsluha přerušení běží se zvýšeným CPL tak dlouho, dokud neudělá iret

Physical Address Extension (PAE)

- Éra 32-bitových x86 s více než 4GB RAM
- Ačkoliv procesor, tj. i jádro OS, dokázalo adresovat pouze 4GB RAM, tabulky stránek mohly adresovat více než 4GB RAM
- V podstatě šlo první kroky ke 64-bitovým tabulkám stránek – u PAE s 36-bitovou fyzickou adresou
  - Zavedeno s Pentium Pro
  - Nicméně, už 386 by teoreticky zvládnula 64TB virtuální paměti – technická realizace v praxi je ovšem něco jiného

80386 virtuální adresový prostor

- Virtuální adresy mají 48-bitů – dáno MMU
  - 16 bitů segment selektor, 32 bitů offset v rámci segmentu
- 16+32= 48 bitů virtuálního adresy
  - Ale 2 bity selektoru jsou použity na privilege level
  - 1 bit selektoru indikuje globální/lokální tabulku
  - => 46 k adresování použitelných bitů => 2^46=64TB
- Jedná se ale jenom o teoretický limit! V praxi nikdy nebylo použito.

Segmentace a stránkování

- Ačkoliv je segmentace považovaná za historický artefakt, nedá se říci, že by byla nepoužívaná
  - Např. Vx32 user-level sandboxing na x86 ji používá ke spouštění nedůvěryhodných programů na FreeBSD, Linuxu a Mac OS
  - Implementace TLS ve Windows – viz dále

Segmented Paging

- Tabulky stránek jsou segmentovány
- Virtuální adresa je logical_page:offset
- logical_page je segment_number:segment_offset

Paged Segmentation

- Segmenty se skládají ze stránek
- Virtuální adresa je seg:offset
- Offset je page_number:page_offset
- Way to go on x86

HW vs SW

- Proč to zatím vypadá jako specifikace hw?
- Protože ať už je to Linux, UNIX, Mac OS či Windows, všichni nakonec dělají to samé
  - A takhle už víme, co to je, aniž bychom se upnuli na konkrétní implementaci

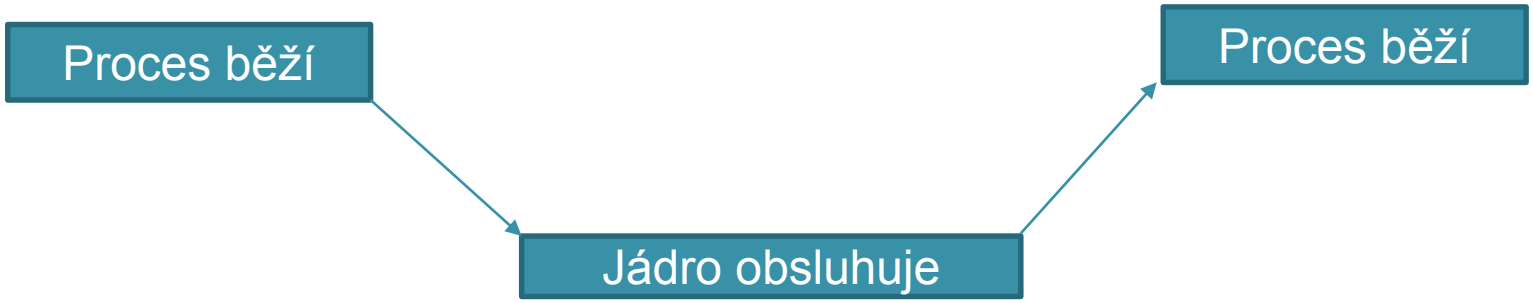
KIV Operační systémy

Obsluha volání služeb OS, přerušení a vyjímek

Verze 1.0

Monolitické jádro

- Všechny služby jádra OS běží s CPL=0
- Ovladače mohou běžet jako moduly jádra také s CPL=0
- K drahému přepnutí kontextu dojde jenom 2x
  - Při volání služby OS
  - Po dokončení služby OS
    - Pozor, řízení se může předat jinému procesu



Monolitické jádro - velikost

- Jádro může být příliš velké, než aby se vešlo do paměti
  - Nebo nechá příliš málo volné paměti
  - Problém zejména u Embedded systémů
  - U PC s Linuxem to zase takový problém není
- Např. AIX a MULTICS umí dynamicky načítat a uvolňovat moduly

Kernel Panic

- Když selže jádro, co budeme dělat?
  - Nejspíš už nic.
- Vlastní jádro bývá obvykle dobře odladěné
- Daleko větší problém představují ovladače běžící s CPL=0
- Chyba v ovladači pak s sebou vezme celé jádro
  - Bez ohledu na to, jak má být dotčené jádro dobré
- Příčinou může být i vadný hw

Mikrojádro – proč?

- Když přesuneme co nejvíce kódu mimo CPL jádra, pak zvyšujeme šanci, že jádro přežije
  - A následně můžeme restartovat pouze ten kód, který selhal
- Mikrojádro obsahuje pouze základní, nezbytné služby
  - Plánovač
  - Alokace paměti (ale ne celý správce paměti)
  - Meziprocesová komunikace

Mikrojádro – proč?

- Když „nejaderný“ kód OS poběží mimo CPL(plánovač má CPL jádra), pak ho můžeme napsat jako preemptivní a plánovat jako běžný proces
  - Požadavky na vykonání služeb se pak dají seskupovat a tím se zvýší efektivita jejich obsluhy systémem
    - Monolitické a hybridní jádra mají Bottom-Half, viz dále, které mají stejný cíl
  - Kód pak může běžet na různých CPU,
    - Dokonce i na jiném počítači u distribuovaných OS

Mikrojádro – skutečný výkon

- Mikrojádro je pomalé kvůli příliš velkému počtu přepínání kontextu
  1. Proces volá službu OS – přepne se kontext do CPL mikrojádra
  2. Mikrojádro určí příslušný obslužný kód mimo CPL mikrojádra a předá mu řízení => přepnutí kontextu
  3. Když se dokončí obsluha mimo CPL mikrojádra, předá se opět řízení do kontextu s jiným CPL
     - Může být opět mikrojádro a z něj pak následně přepnutí do CPL uživatelského procesu

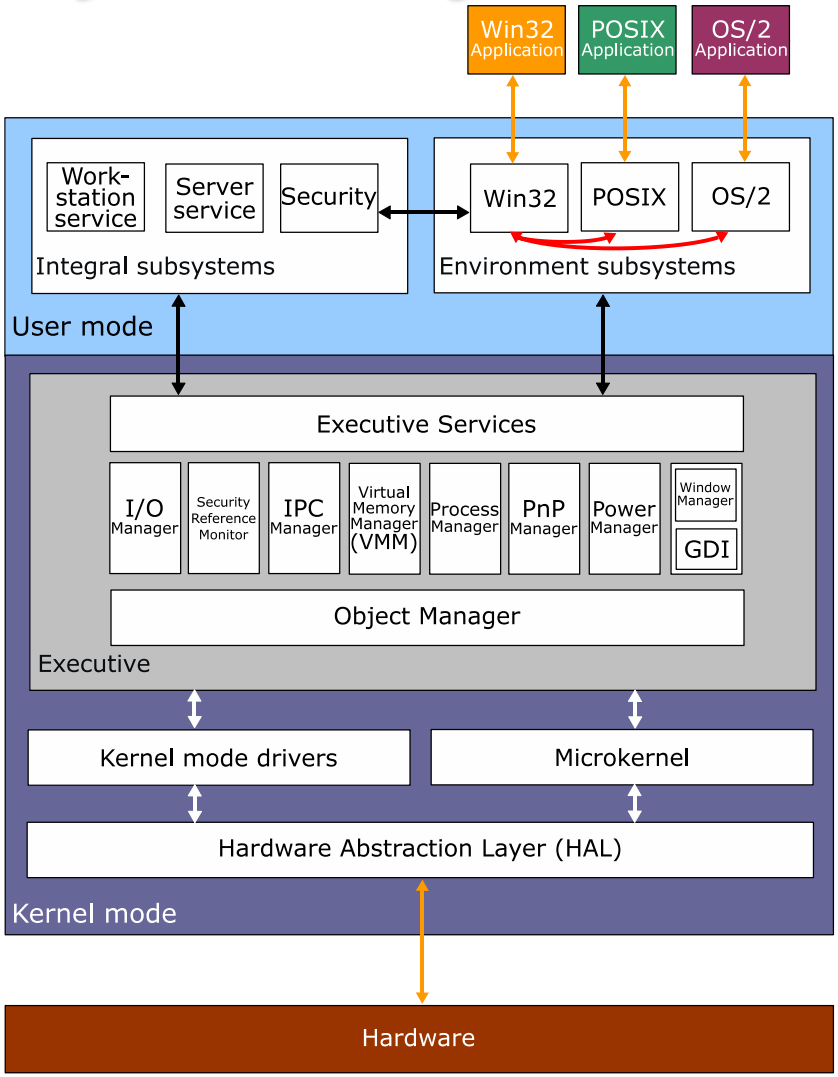
Mikrojádro – optimalizace

- Protože mikrojádro přeposílá požadavky jinam, optimalizace by spočívala v tom, jak při předání výsledků ušetřit alespoň jedno přepnutí kontextu
  - Případně jak rovnou volat kód mimo CPL mikrojádra
    - Taková možnost musí ale počítat s tím, že kód mimo mikrojádra mohl být nahrán znovu na novou adresu v paměti
- Nejrychlejší mikrokernel měla AmigaOS v době, kdy ještě Amiga neměla obdobu Protected-Mode
  - Pak už na tom byla stejně jako jiné mikrokernely

Hybridní jádro

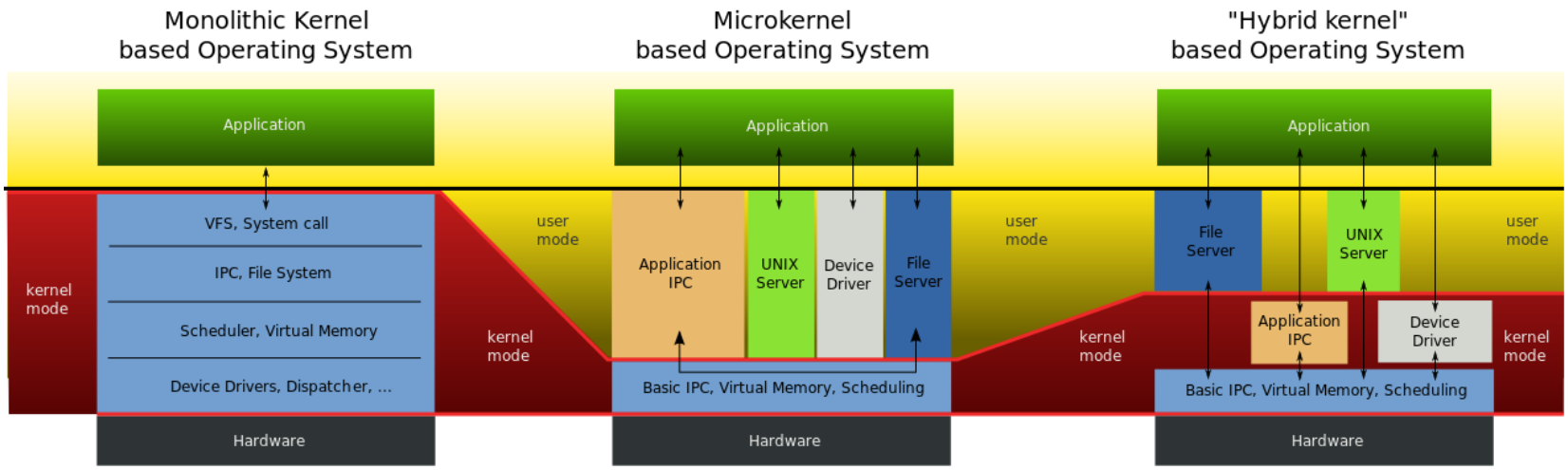
- Protože je mikrojádro pomalé, vývoj OS začne monolitickým jádrem
- Výkonnostně kritické části kódu pak zůstanou v jádře, zatímco ostatní se přesunou na úroveň s méně privilegovanou CPL
  - Např. ovladače třetích stran, které často padají a zákazníci si myslí, že je to špatným OS

Hybridní jádro – Win2000



http://en.wikipedia.org/wiki/Hybrid_kernel

Srovnání jader



http://en.wikipedia.org/wiki/Monolithic_kernel

GetTickCount

- RTL knihovny často nabízejí funkci podobného názvu, která vrací počet tiků od resetu procesoru
- Získání korektní hodnoty není na moderních procesorech triviální => proto se vyplatí, aby RTL knihovna zavolala příslušnou funkci OS

- Pozn. nebavíme se o High-Resolution Timer

GetTickCount - pozadí

- 80286 měla na adrese 0000:046c real-mode 4-bytovou proměnnou ROM-BIOSu, která udávala počet 55ms tiků od resetu procesoru
- Pozdější procesory mají tick-counter registr, který je dnes větší a má daleko nižší režii přístupu, ale..
  - Registry nejsou synchronizované mezi jednotlivými jádry v systému - resp. není to garantované
  - Power-saving na konkrétním jádru ovlivní hodnotu čítače
  - Out-of-order může samotnou čtecí instrukci vykonat příliš brzy
- => ať si takové varianty ošetří jádro namísto volajícího threadu

GetTickCount – kód proces

- Pro jednoduchost předpokládejme x86 uniprocesor
- Čítač tiků přečte instrukce RDTSC
- V uživatelském procesu tak kód může vypadat takto:

```
size_t tickcount = GetTickCount();
```

- Přičemž předpokládejme, že se návratová hodnota vrátí v registru EAX

GetTickCount – koncept volání

- Instrukci RDTSC lze vykonat v uživatelském režimu
- Kód příslušné funkce namapuje OS do adresového prostoru příslušného procesu
  - Sdílení kódu
  - Dynamické knihovny
- Obsluha takové funkce tedy vůbec nevyžaduje přepnutí kontextu => rychlost dokončení služby OS
  - Což neznamená, že k němu nemůže dojít vlivem časovače

GetTickCount – volání služby

1. GetTickCount se přeloží jako call instrukce, která předá řízení na adresu, na které je příslušný RTL kód, nejpravděpodobněji kód od výrobce překladače
2. RTL kód zatím blíže neurčeným způsobem zná adresu, kam OS namapoval svůj sdílený kód (dynamická knihovna), který dělá požadovanou činnost - RTL kód udělá příslušný call
3. Kód OS mj. zapíše tick count do EAX a udělá ret
4. RTL kód udělá ret
5. Proces zná počet ticků (zanedbali jsme errno)

Dynamická knihovna

- Naprostá většina programů používá knihovny
  - Statické knihovny se během překladu stanou součástí výsledného kódu spustitelného programu
  - U dynamických knihoven se to nestane, knihovna bude existovat jako samostatný soubor
    - Soubor může na disku existovat jenom jednou
    - Soubor může mít do paměti namapováno několik procesů
      - Viz koncept sdílení kódu

Dynamic loading

- Proces explicitně stanoví kdy a která knihovna se má načíst do paměti – tj. proces zavolá jednu z následujících funkcí, které předá cestu ke knihovně
  - dlopen – Linux, MacOS
  - LoadLibrary – WinAPI
- OS volajícímu procesu inkrementuje reference counter, kolikrát už danou knihovnu načetl
  - Pokud knihovna nebyla dosud načtena, OS alokuje procesu paměť, do které nahraje knihovnu
  - OS vyřeší, která část nově alokovaná data bude označena jako spustitelná a která jako datová

Library Hijacking

- Během načítání knihovny nemusí být cesta k ní jednoznačná
  - Např. bude-li to pouze jméno souboru bez cesty, systém bude soubor hledat v adresáři se spustitelným souborem
    - A když tam nebude, tak v adresářích specifikovaných nějakou proměnnou – např. PATH ve Windows
    - Bude-li program pod Windows specifikovat pouze jméno souboru knihovny v cestě dané PATH, lze podvrhnout falešnou knihovnu do adresáře k jeho .exe souboru, a tato podvržená knihovna tak bude načtena namísto té v cestě PATH

Dynamic unloading

- Během procesu dynamic loading získá proces deskriptor načtené knihovny
- Proces explicitně stanoví, kdy knihovna identifikovaná příslušným deskriptorem, uvolní z paměti
  - dlclose, FreeLibrary
  - Uvedené funkce sníží reference counter knihovny o jedna
    - OS uvolní knihovnu z paměti teprve až reference counter = 0
      - Ukazatele do této paměti se tak stanou neplatnými

Dynamic GetAddress

- Máme-li knihovnu načtenou v paměťovém prostoru procesu, potřebujeme ještě získat adresy požadovaných funkcí
  - Programátor procesu musí znát prototyp těchto funkcí
- Proces zavolá funkci, které předá název funkce, a OS mu vrátí pointer na adresu této funkce
  - dlsym
  - GetProcAddress
- Programátor knihovny označí, které funkce exportovat

Dynamic knihovna - inicializace

- Dynamická knihovna není nic jiného než spustitelný program – má svůj main, který se spustí při jejím načtení
  - dllmain pod Windows
  - .interp sekce v ELF formátu pod Linuxem
- Dynamická knihovna má tak možnost inicializovat se
  - A stejně tak má možnost deinicializace před uvolněním
    - dllmain
    - sekce .fini

Příklad - Win

```
HMODULE lib = LoadLibrary(“knihovna.dll”);

TFunc *func = GetProcAddress(“funkce”);

func();

FreeLibrary(lib);
```

Dynamic linking - proč

- Dynamic loading se hodí např. pro načítání plug-inů
- Ale co když budeme namapovat dynamickou knihovnu hned při spuštění programu?
  - Buď v případě, že program knihovnu vyžaduje a tudíž nemá smysl řešit její podmíněné načítání
  - Anebo v případě, kdy knihovna patří OS, který jejím prostřednictvím poskytuje služby procesu
- V hlavičce programu se označí, které knihovny má OS dynamicaly load rovnou při zavádění programu

Dynamic linking - PLT

1. Ve zdrojovém kódu se proměnné ukazující na symboly (funkce, proměnné, atd.) knihovny označí
   - Např. pomocí __declspec (dllimport) pod Windows
   - Značky jsou uvedeny v samostatné sekci v programu
     - Procedure Linkage Table (PLT)
     - Např. .rel.text pod Linuxem
   - Značka říká, že se má na příslušnou adresu zapsat hodnota získaná dlsym/GetProcAddress
     - Zařídí zavaděč knihovny - OS

Dynamic linking - PLT

```
$ cat a.c
extern int foo;

int function(void) {
    return foo;
}
$ gcc -c a.c
$ readelf --relocs ./a.o

Relocation section '.rel.text' at offset 0x2dc contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000004  00000801 R_386_32          00000000   foo
```

https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html

Dynamic linking - PIC

- Do paměti se může zavést několik knihoven, tj. každá knihovna se může pokaždé zavést na jinou virtuální adresu
  - Potřebujeme tzv. Position Independent Code (PIC)
  - Překladač musí zajistit, aby se veškerý kód adresoval relativně k program counteru (IP registr)
  - Alternativě se používala relokovatelný kód (např. před Vista), kdy zavaděč programu modifikoval kód knihovny ještě před spuštěním tak, aby šla spustit z adresy, kam byla zavedena

Dynamic linking - GOT

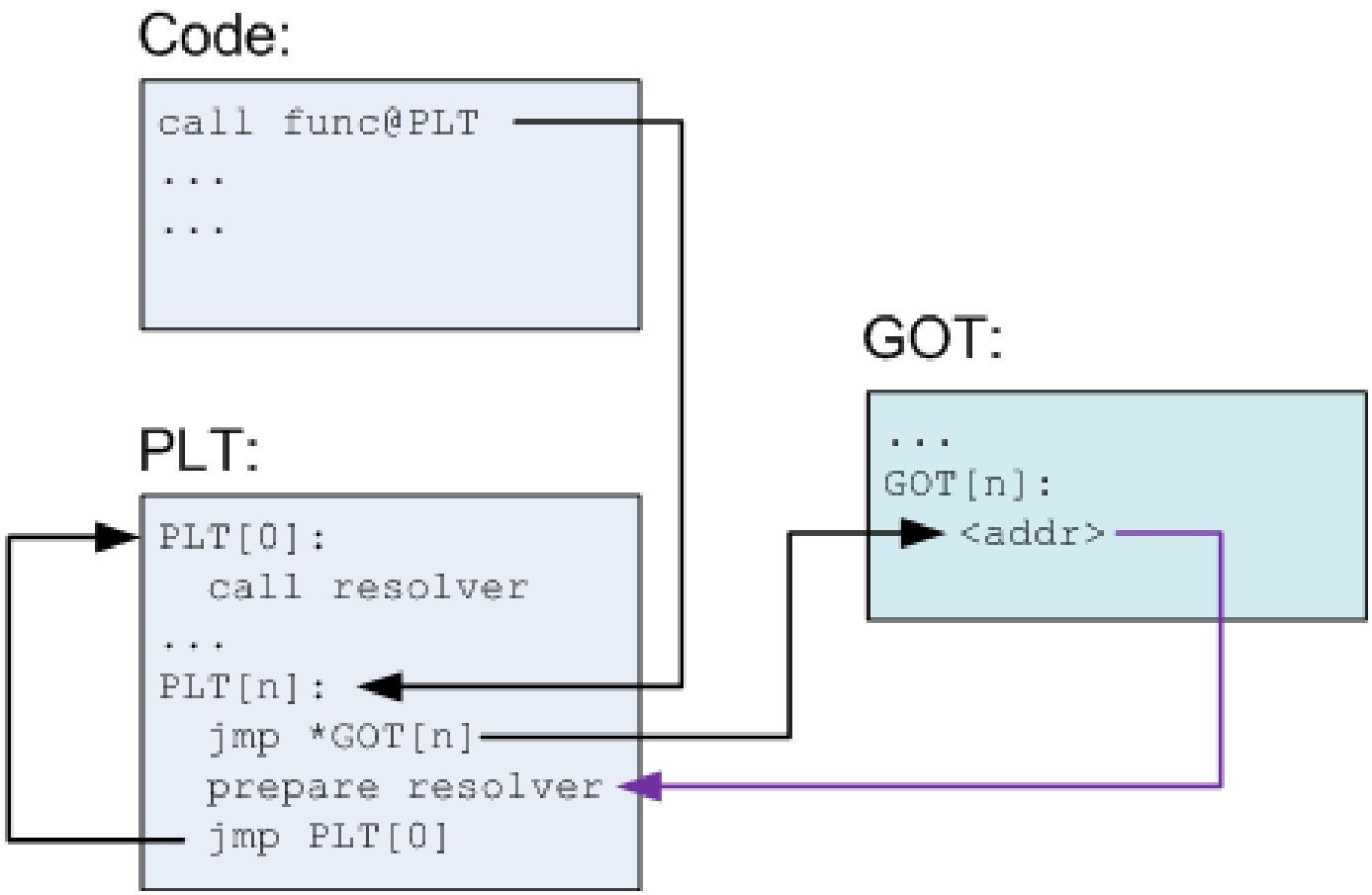
- Relokace vyžaduje přepsání kódu knihovny jejím zavaděčem
- Lepší je vytvořit tabulku, Global Offset Table (GOT), ve které budou adresy symbolů knihovny, a které (adresy) tam zapíše zavaděč knihovny
  - Kód tedy bude symbol dereferencovat 2x
  - GOT bude privátní pro každý proces
    - A nezměněný kód knihovny tak bude sdílený pro všechny procesy

Lazy Binding

- Adresy symbolů knihovny není nutné resolvovat ihned, ale až bude třeba

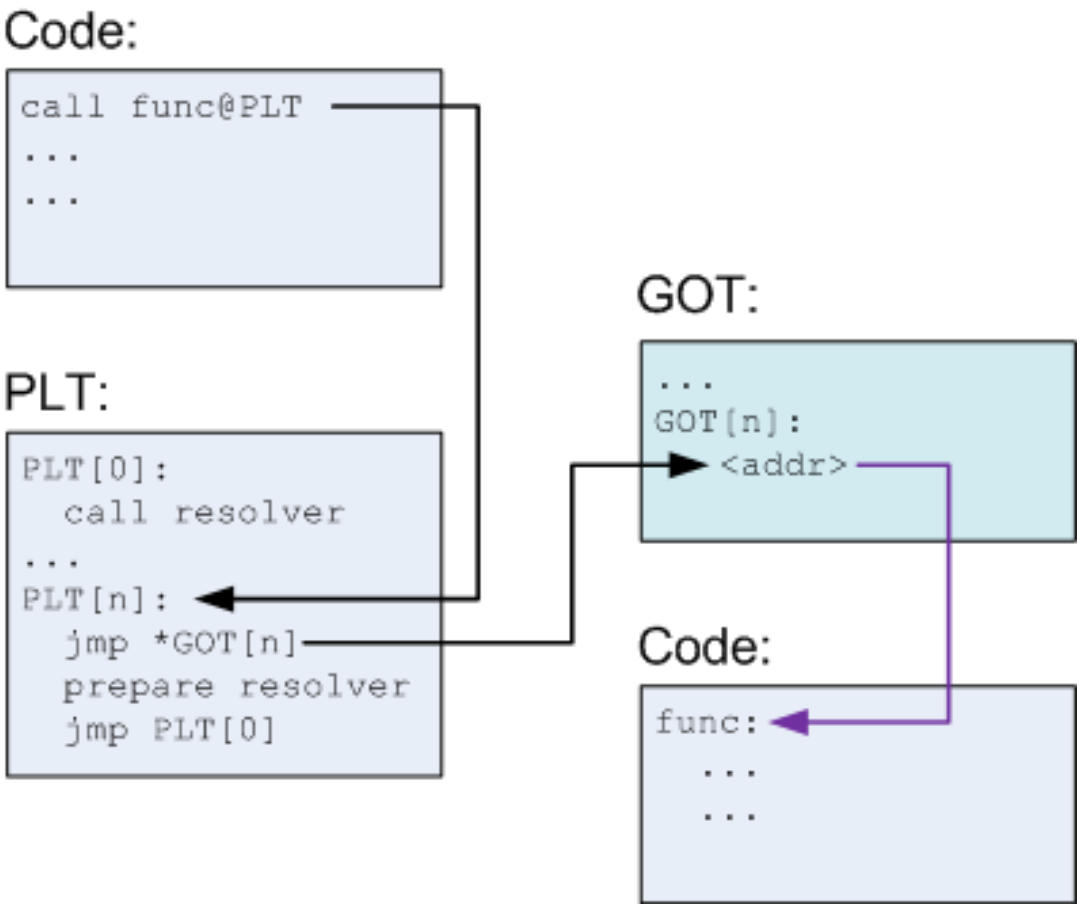
1. Proces volá funkci knihovny
   - Každá knihovnou exportovaná funkce má své ordinální číslo n – tj. volá se adresa PLT[n]
2. PLT[n] ale zatím ukazuje na rutinu zavaděče, která se zavolá jako první a resolvuje adresu rutiny do GOT než zavolá vlastní funkci
   - Než je funkce zavolána, GOT se upraví tak, aby se příště už rovnou volala funkce knihovny, ne rutina zavaděče

Lazy Binding



http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/

Lazy Binding poté



http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/

Randomizace adres kódu

- Některé útoky se spoléhají na to, že se předem ví, co se bude nacházet na konkrétních adresách
  - Např. je možné změnit adresu v GOT tak, aby ukazovala na jinou funkci, než na kterou má
- Jednou z možností obrany je randomizace adres kódu tak, aby adresy byly náhodné a nedaly se uhádnout
  - Což jde tím lépe, čím větší je virtuální adresový prostor

Volání služby OS přes int

- Některé služby OS nelze vyřídit pouze v uživatelském adresovém prostoru, ale musí se změnit CPL na CPL jádra a předat jádru řízení programu
  - Toto dokáže sw přerušení - x86 instrukce int
- Proces buď sám vygeneruje int, s číslem přerušení dedikovaného OS pro tyto účely, aneb zavolá rutinu v dynamické knihovně OS, která sama posléze vygeneruje příslušný int
- Stejné princip jako u volání MS-DOS API

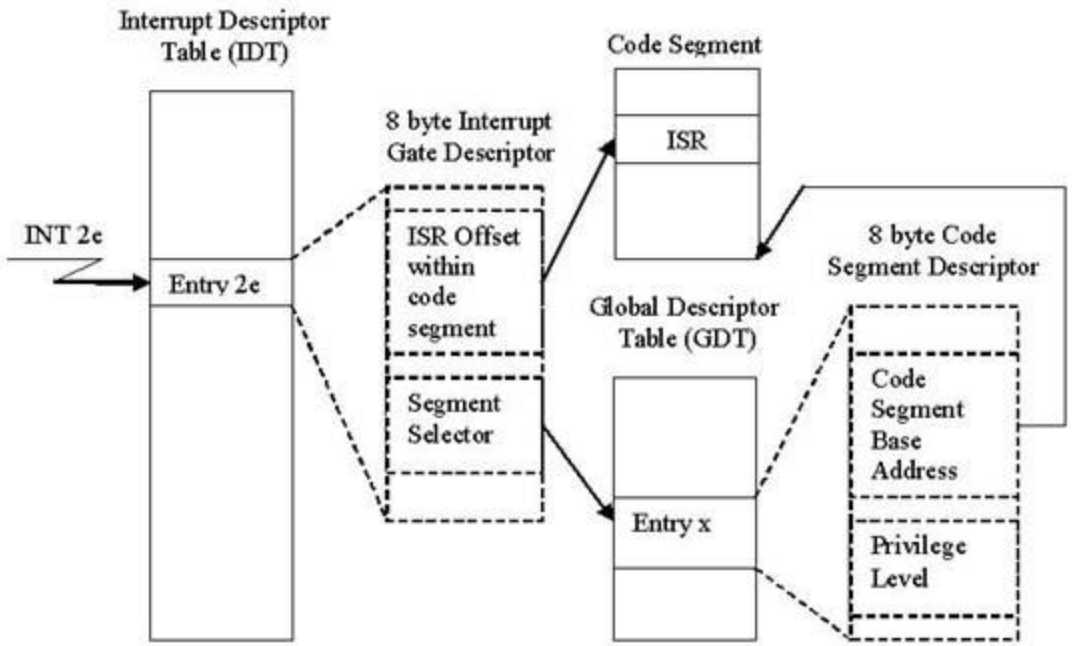
Předání parametrů jádru

- Při použití int se parametry předávají v registrech, nebo přes zásobník
- Vybrané registry mohou obsahovat virtuální adresu do virtuálního adresového prostoru procesu, kde je připravena nějaká struktura blíže specifikující, co má OS udělat

Volání služby v jádře - enter

- Procesor použije číslo přerušení jako index do tabulky vektorů přerušení, aby získal adresu obsluhy přerušení
  - Tabulka nemusí být na adrese 0000:0000 jako po startu v real mode, ale její pozici udává registr IDTR
- Procesor nastaví CS:EIP na získaný vektor přerušení
  - V zásobníku jsou uloženy registry přerušeného vlákna flags, cs a eip; cs:eip ukazují na instrukci, která se má vykonat po návratu
- Jádro OS považuje zásobník přerušeného vlákna za nedůvěryhodný, protože v něm nemusí být místo, a tak bude používat svůj zásobník

Volání služby v jádře - enter



http://www.codeguru.com/cpp/w-p/system/devicedriverdevelopment/article.php/c8035/How-Do-Windows-NT-System-Calls-REALLY-Work.htm

Volání služby v jádře - main

- Obsluha přerušení vykoná pouze nezbytnou část požadované činnosti
  - Lze-li něco odložit na později, odloží se to – viz Bottom Half dále
  - V takovém případě ale nelze vrátit řízení přerušenému vláknu, protože operace nebyla dokončena => vlákno se musí uspat a řízení se předá jinému vláknu
- Jádro také předá řízení jinému vláknu tehdy, když bylo cílem vlákno uspat nebo ukončit
- Plánovač vybere jiné vlákno, které je ve stavu runnable

Volání služby v jádře - exit

- Po dokončení obsluhy přerušení už jádro ví, kterému procesu předá řízení
- Před ukončením obsluhy přerušení tedy jádro vybere zásobník vlákna, kterému předá řízení
  - Tj. v zásobníku jsou registry CS:EIP a flags tohoto vlákna
  - Pokud jde o jiné vlákno, musí se obnovit i zbývající registry
- Obsluha přerušení udělá instrukci iret, která nastaví registry CS:EIP a flags z aktuálního zásobníku SS:ESP
  - A z CS:EIP procesor určí svůj režim - CPL

Task State Segment

- x86 (IA-32) struktura, kam se ukládá kontext přerušené činnosti procesoru
  - V x86 terminologii task, jinak jde o vlákno
  - Lze vytvořit pro každé vlákno v systému
  - Používá se např. k implementaci syscall jako rychlejší varianty int
- Deprecated na x86-64

Syscall

- int generovaný programem je synchronní událost vůči běhu programu => lze toho využít
- syscall/sysret jsou speciální instrukce, které toho využívají a dokáží přepnout procesor do režimu jádra cca 3x rychleji než int
  - int – původní, pomalý mechanismus volání jádra
  - sysenter/sysexit – protected mode, compatibility mode, vyžaduje TSS
  - syscall/sysret – long mode
    - Jádro musí nastavit zásobník v době, kdy může dojít např. nemaskovatelnému přerušení

syscall – sdílená stránka

- Kdy lze použít int, sysenter/sysexit či syscall/sysret závisí na dostupném hw a jádru OS
  - V každém případě se ale jedná o netriviální činnost, které by mělo být volající vlákno ušetřeno
- OS namapuje do virtuálního adresového prostoru procesu speciální stránku, která zavolá službu jádra jádrem očekávaným způsobem
  - Kód na této stránce poskytne jádro OS, volající vlákno pouze udělá call na adresu této stránky

syscall - volání

- libc má funkci syscall
- ntdll.dll má KiFastSystemCall a KiFastSystemCallRet
- Adresa musí být fixní, aby byla známa RTL v době jejího psaní
  - Buď fixní adresa, např. 32-bit Win
    - call dword ptr [adresa]
  - Nebo se použijí registry fs (Win) či gs (Linux), přičemž segmentový registr odkazuje na Thread Control Block
    - call fs:[offset od začátku TCB]

Vyjímky

- Vyjímka je přerušení generované procesorem, když dojde k
  - Trap – např. breakpoint
  - Fault – opravitelná chyba, program může pokračovat
    - Např. Page Fault
  - Abort – neopravitelná chyba
    - Např. Double Fault – dojde k Page Fault, ale handler je ve stránce, která není v RAM
  - Některé vyjímky přidávají na zásobník extra hodnotu s chybovým kódem, který je třeba odstranit před návratem z obsluhy přerušení, aby se na vrcholu byly cs, ip a flags

Exception handler

- V případě, že procesor nedokáže zavolat obsluhu vyjímky, dojde k Double Fault
  - Pokud ani pro ni nebude obsluha, dojde k Triple Fault
    - Což je samo o sobě známkou, že je něco špatně v obsluze první vyjímky
      - Vyjímku nelze zamaskovat
        - => buď budeme psát perfektní, bezchybný kód
        - Anebo alespoň spolehlivé obsluhy vyjímek

Exception handler – jádro OS

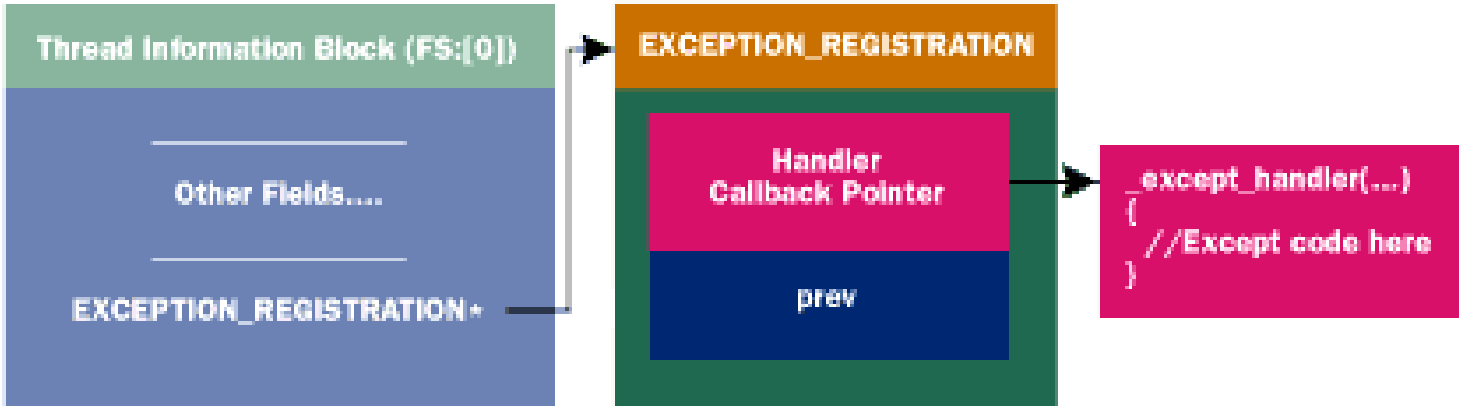
```
exc_0d_handler:
    push gs
    mov gs,ZEROBASED_DATA_SELECTOR
    mov word [gs:0xb8000],'D '
    ;; D in the top-left corner means we're handling
    ;;  a GPF exception right ATM.

    ;; your 'normal' handler comes here
    pushad
    push ds
    push es
    mov ax,KERNEL_DATA_SELECTOR
    mov ds,ax
    mov es,ax

    call gpfExcHandler

pop es
    pop ds
    popad

    mov dword [gs:0xb8000],'  D-'
    ;; the 'D' moved one character to
the right, letting
    ;; us know that the exception
has been handled properly
    ;; and that normal operations
continues.
    pop gs
    iret
```

http://wiki.osdev.org/Troubleshooting

Dělení nulou

- Pokud by OS neměl handler vyjímek, došlo by k restartu počítače
  - OS ho má – tj. dojde-li k vyjímce v uživatelském procesu, OS vyjímku zachytí
- Např. dělení nulou je Fault
  - Pokud proces nenastavil svůj handler, OS program ukončí
  - Pokud ho proces nastavil, OS mu předá řízení
  - Opět se k tomu použije Thread Control Block

Try catch



https://www.microsoft.com/msj/0197/exception/exception.aspx

KIV Operační systémy

Vlákna

Verze 1.0

Vlákna

- Chceme-li v operačním systému souběžně spouštět několik procesů, musíme implementovat vlákna
- Vlákna zvyšují uživatelův dojem, že je systém responsivnější
  - V popředí běží aktuální vlákno, se kterým uživatel interaguje
  - V pozadí běží ostatní vlákna, která vyvíjí další činnost v době, kdy vlákno na popředí neběží
    - Nebo běží na dalších procesorech v systému

Vlákno

- Vlákno je sekvence instrukcí, která může být spravována plánovačem OS
  - Unit of CPU scheduling
- Z pohledu programátora je to často funkce, která se vykonává asynchroně k funkci main
  - Přičemž funkce main běží ve vlastním vláknu, které vytvořil OS po zavedení procesu do paměti, aby ho mohl spustit
  - Běžící proces je dynamická kolekce vláken s alespoň jedním vláknem
- Proces vlastní vlákna

Time slicing - uniprocessor

- Na jednom procesoru může v jeden okamžik běžet pouze jedno vlákno
- Chceme-li uživateli předstírat, že jich tam běží více, vlákna se musí střídat
- Time Slicing je technika, kdy se čas procesoru rozdělí na časová kvanta, která se postupně přidělují jednotlivým vláknům
  - Vždy běží vlákno, která má momentálně přidělené kvantum strojového času

Kooperativní multitasking

- Otázka zní, jak velké bude časové kvantum?
- Jednou z možností je, že vlákno poběží tak dlouho, dokud se dobrovolně nevzdá procesoru
  - Procesoru se vzdá systémovým voláním
  - Jádro OS pak vybere další vlákno, které má běžet a předá mu řízení
  - Výhodou je, že naplánování dalšího vlákna není časově kritické, protože se neděje v obsluze přerušení
  - Nevýhodou je, že vlákno může procesor uzurpovat příliš dlouho a OS se může jevit jako/být zaseknutý – Windows 3.1

Preemptivní multitasking

- Jádro OS má nainstalovanou obsluhu přerušení, které generují hodiny
- Během obsluhy přerušení hodin neběží vlákno, ale obsluha přerušení
  - => obsluha, tj. jádro, je schopné uložit stav aktuálního vlákna a nahradit ho stavem jiného vlákna
    - Tj. po návratu z obsluhy přerušení poběží jiné vlákno bez ohledu na to, jak dlouho by chtělo původní vlákno počítat
    - Celá akce však musí být rychlá, je to v obsluze přerušení

Kontext vlákna

- Při změně vlákna se vždy uloží stav aktuálního vlákna a obnoví se dříve uložený stav nově vybraného vlákna
- Kontext vlákna je dán jeho proměnnými
  - Některé proměnné jsou v registrech
  - Jiné proměnné jsou např. v zásobníku, kam ukazuje SS:RSP
  - Další proměnné jsou v paměti, kam mohou ukazovat další registry
  - Ukazatelem na instrukci, která se má vykonat – CS:RIP
  - A stavovou proměnnou OS, která říká, zda proces běží, je pozastaven, atd.

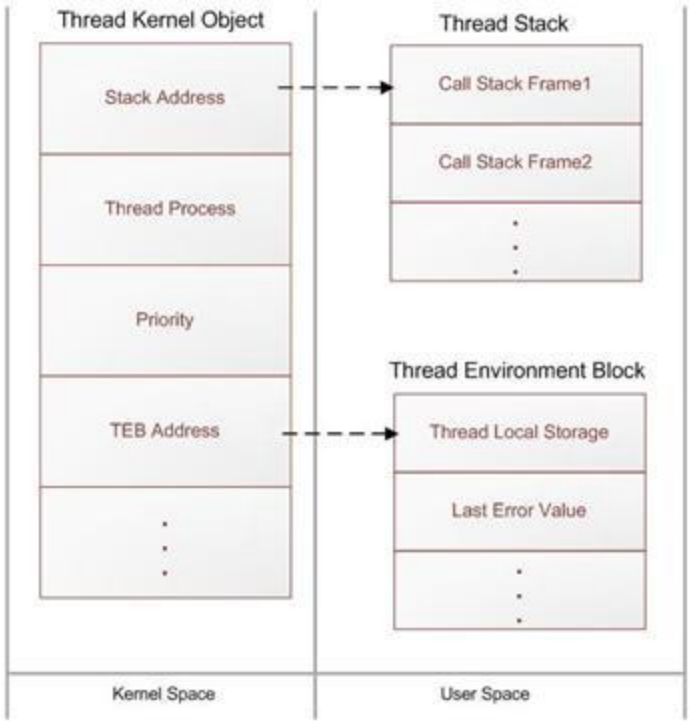
Thread Control Block

- Jádro spravuje thread podle jeho TCB
  - Pod Win/Linux přístupný přes fs/gs registry na x86, x86-64
- TCB obsahuje:
  - Uložené hodnoty registrů procesoru
  - Priorita
  - Stavovou proměnnou, čas dosavadního běhu vlákna
  - Ukazatel na Process Control Block
  - A další specifické info, jako jsou např. seznam obsluh vyjímek, skok na stránku s funkcí syscall, atd..

Windows User Thread

- Vlákno, které patří uživatelskému procesu, se sestává ze tří komponent:
  - Kernel object
    - Čas vytvořené, běhu, ukazatel na TEB, stav, priorita, počet změn kontextu, afinita, atd.
  - Zásobník
  - TCB, zde TEB aka Thread Environment Block
    - Thread Local Storage, obsluhy vyjímek, poslední chyba, impersonace, vlastněné kritické sekce, atd.

Windows User Thread

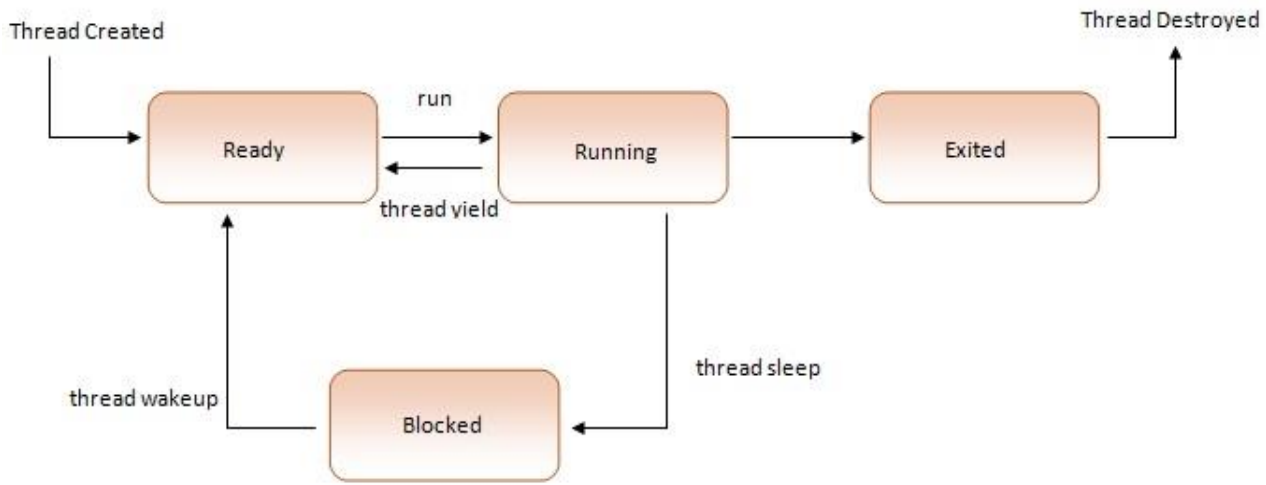


http://www.codeproject.com/Articles/662735/Internals-of-Windows-Thread

Stav vlákna

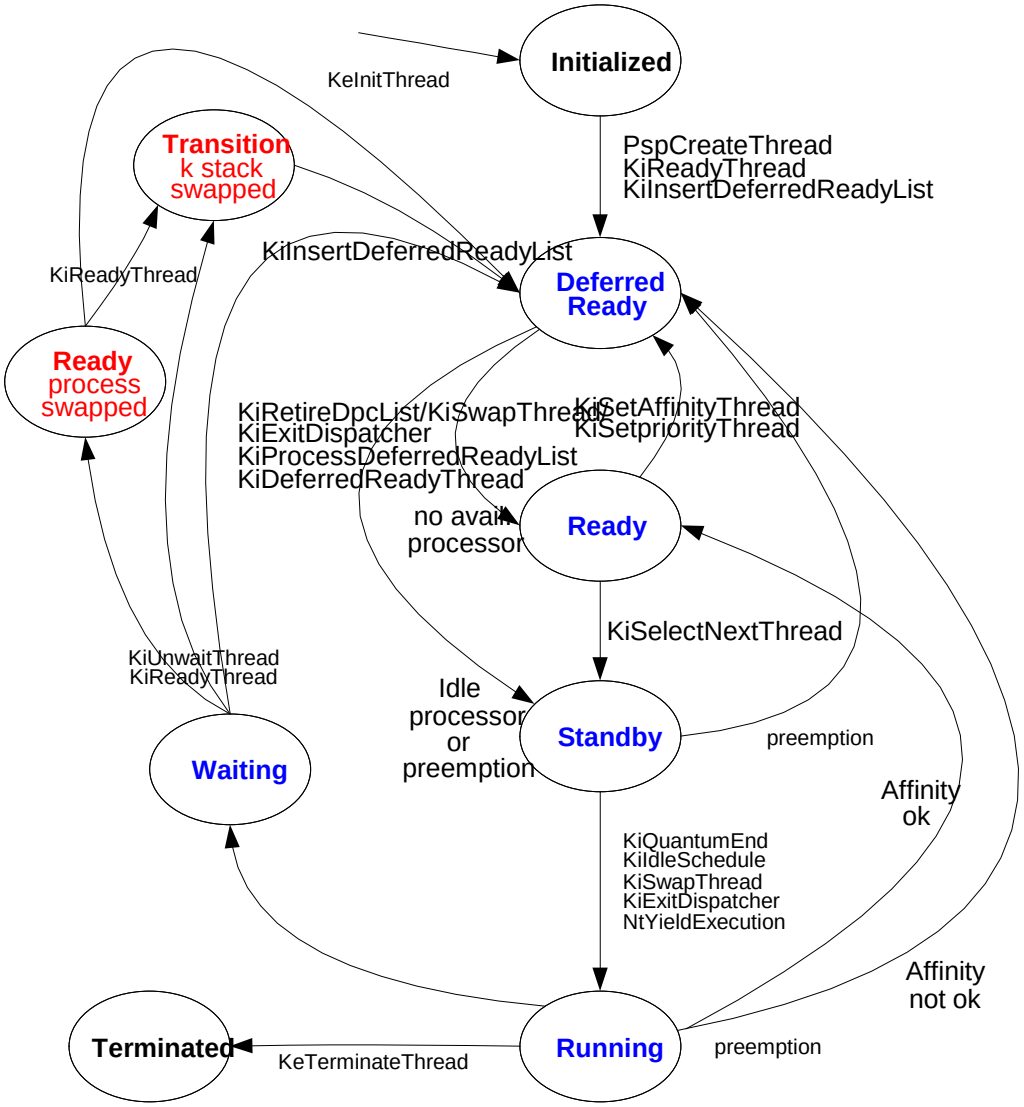
- Ve zjednodušeném pohledu je vlákno
  - Runnable – může být naplánováno a spuštěno
  - Running – vlákno běží
  - Blokované – buď se uspalo, nebo třeba čeká na podmínkové proměnné, nebo na dokončení I/O operace
  - Ukončené
- Ve skutečnosti je to složitější, protože OS potřebuje vědět, proč vlákno čeká
  - Určitě je rozdíl např. mezi PageFault a kritickou sekcí

Stav vlákna



http://www.codeproject.com/Articles/662735/Internals-of-Windows-Thread

Windows SMP Thread State



Kernel Thread Transition Diagram
DavePr@Microsoft.com
2003/04/06 v0.4b

http://i-web.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/Lectures/03-ThreadScheduling/ThreadScheduling.ppt

Plánování vláken

- OS neplánuje procesy, ale vlákna, protože proces je jenom kontajner
  - Pokud nejde např. o Linux, který nedělá rozdíl mezi procesem a vláknem; všechno je pro něj runnable task
- Vlákna s nejvyšší prioritou běží nějčastěji
  - Ale zároveň se občas musí ke slovu dostat i vlákna, která mají nízkou prioritu
  - Vlákno asociované s aktuálním vstupem, např. UI dialog, má dočasně zvýšenou prioritu, jako divadlo na uživatele, které mu zajistí větší dojem z responsivnosti systému

Round Robin

- Žádná priorita, jenom seznam vláken
  - Vláknům se přiděluje pouze časové kvantum 20 – 50 ms
  - Když se jedno vlákno přeruší, vezme se další runnable na seznamu
- Čím menší kvantum, tím více ztraceného času při přepínání vláken
- Čím větší kvantum, tím se zase OS zdá uživateli pomalejší
- Hodně procesů, které by měly mít nízkou prioritu může vyhladovět procesy, které by ji měly mít vysokou

Round Robin s prioritou

- Několik seznamů vláken
- Co seznam, to jedna priorita
- Nejprve běží vlákna ze seznamů s vyšší prioritou, dokud takový seznam není prázdný
  - Vlákno skončilo, nebo je blokováno
- Takže ve výsledku je možné vyhladovět procesy s nízkou prioritou

Plánovač Linuxu

- Používá velká časová kvanta pro důležité procesy
- Modifikuje velikost přidělovaných kvant podle využití CPU
- Snaží se držet procesy na stejném CPU
  - Každé CPU má svou fronton procesů, ze které plánuje procesy ke spuštění
  - Pokud má některý CPU frontu příliš dlouhou, přebývající procesy jsou přesunuty na jiný CPU
    - Každý proces má (affinity) masku, která říká, na kterých procesorech může běžet a na tom se nic nemění
- Completely Fair Scheduler, O(1)

Základ plánování

1. Vybere se fronta s největší prioriou a spustitelným procesem
   - V systému jsou dvě sady (active & expired) 140 front, každá pro jednu statickou prioritu
     - 0 – 99 jsou real-time procesy
     - 100 – 139 jsou normální procesy, nastavuje se pomocí nice()
     - 5 integerů tvoří bitmapu jejíž bity říkají, která fronta má spusitelný proces
2. Vybere se v ní první spustitelný proces a vypočítá se jeho časové kvantum
3. Spustí se a až vyčerpá své kvantum, dá se expire fronty
4. Zpět do prvního bodu

Výpočet kvanta

- SP – statická priorita
  - SP<120: Quantum = (140 – SP)*20
  - SP>=120: Quantum = (140-SP)*5
  - Proces s větší prioritou (tj. menším číslem) dostává větší časová kvanta

Priorita	SP	Nice	Quantum
Nejvyšší	100	-20	800 ms
Vysoká	110	-10	600 ms
Normální	120	0	100 ms
Nízká	130	+10	50 ms
Nejnižší	139	+20	5 ms

Dynamická priorita (DP)

- Vypočítává se ze statické priority a doby, po kterou proces neběžel
- Proces dostane bonus <0, 10>
  - 0 – sníží prioritu o (bonus) 5
  - 5 - neutrální (bonus 0)
  - 10 – zvýší prioritu o (bonus) 5
  - Pokud je ovšem process interaktivní, pak platí: bonus – 5 >= SP/4 - 28
- DP = max (100, min(SP - bonus + 5, 139))

KIV Operační systémy

Symetrický multiprocesor

Verze 1.0

Bootstrap procesor x86

- Po zapnutí počítače a inicializaci BIOSu je aktivní pouze jeden procesor, respektive jedno procesorové jádro – tzv. BSP aneb bootstrap procesor
- Zavaděč OS musí zjistit, zda systém obsahuje i další procesorová jádra a aktivovat je
  - Pokud se mu to nepodaří, aktivuje se uniprocesorové jádro

MP Floating Pointer Structure

- Začíná signaturou _MP_ a popisuje dostupné procesory
- Hledá se
  - V 1. kB Extended BIOS Data Area
  - Posledním kB základní paměti
    - „640K ought to be enough for anybody." – Bill Gates, 1981
  - V adresním rozsahu ROM-BIOSu

MP Config

- Jedna z položek MP Floating Pointer Structure je MP Config Pointer ukazující na seznam dostupných procesorů

MP Config

Processor Entry			
Field	Offset	Length	**Description/Use**
Entry Type	0	1B	**Since this is a processor entry, this field is set to 0.**
Local APIC ID	1	1B	**This is the unique APIC ID number for the processor.**
Local APIC Version	2	1B	**This is bits 0-7 of the Local APIC version number register.**
CPU Enabled Bit	3:0	1b	**This bit indicates whether the processor is enabled. If this bit is zero, the OS should not attempt to initialize this processor.**
CPU Bootstrap Processor Bit	3:1	1b	**This bit indicates that the processor entry refers to the bootstrap processor if set.**
CPU Signature	4	4B	**This is the CPU signature as would be returned by the CPUID instruction. If the processor does not support the CPUID instruction, the BIOS fills this value according to the values in the specification.**
CPU Feature flags	**8**	**4B**	**This is the feature flags as would be returned by the CPUID instruction. If the processor does not support the CPUID instruction, the BIOS fills this value according to values in the specification.**

MP Config

- Jedna z položek MP Floating Pointer Structure je MP Config Pointer ukazující na seznam dostupných procesorů
  - BSP
  - AP
    - aka auxiliary processor
    - aka application processor

SMP Bootstrap x86

1. Po zapnutí počítače jsou všechny procesory v reálném režimu
2. BIOS vybere BSP a ostatní procesory zastaví
3. Kód běžící na BSP prohledá paměť, zda najde _MP_
4. Pokud nenajde, zavede se jednoprocesorové jádro
5. Pokud našel, inicializuje APIC BSP
   - Děje se v protected-mode
6. Kód běžící na BSP postupní vzbudí AP pomocí Init-IPI (Inter-Processor Interrupt)

SMP Bootstrap x86

7. Kód běžící na AP ho přepne do protected-mode a začne svoji další činnost synchronizovat s kódem běžícím na BSP
8. Jakmile jsou inicializovány všechny AP, BSP přepne I/O APIC do symetrického I/O režimu
   - Routovací tabulka, která přesměruje přerušení od sběrnic periferií na některý lokální APIC
9. Pokračuje se vlastní inicializací SMP jádra

SMP úskalí plánování

- SMP sice znamená, že každý procesor je stejný, a tudíž že lze každé vlákno SMP OS naplánovat na libovolný procesor, ale v praxi to rozhodně není dobrý nápad
- Každý procesor má svoji cache, ve které jsou uložena data vláken, která na procesoru naposledy běžela
- Cache podstatným způsobem přispívá k rychlému běhu vláken
- Pokud vlákna budeme naivně migrovat mezi procesory, vlákna o tuto výhodu přijdou a celý systém se zpomalí

SMP Synchronizace

- Vlákna běžící na jednotlivých procesorech mají pouze jednu možnost, jak se synchronizovat
  - Atomické instrukce
  - Add, Sub, CompareExchange aka TestAndSet
  - Prefix lock, který zamkne sběrnici
  - x86 – mov strojového slova zarovnaného na adresu beze zbytku dělitelnou velikostí strojového slova (např. sizeof eax či rax)

SpinLock

- Potřebujeme-li, aby pouze jeden z procesorů vykonával kritickou sekci, pak v paměti potřebujeme proměnnou, jejíž stav říká, zda je kritická sekce obsazená, či nikoliv
- Pokud bude obsazená jiným procesorem, pak příchozí procesor čeká ve smyčce, dokud mu ji jiný procesor neodemkne (tj. nemá smysl na uniprocesoru)
  - Pak si ji sám zamkne
- Co kdyby čekalo více procesorů?
  - => Operace s proměnnou musí být atomické

SpinLock - zamčení

```
mov edx, DWORD(-1) //-1 zamčeno
                          //otestujeme stav zámku, 0 = odemčeno
spin:  mov eax, [lockState]
        test eax, eax
         jnz spin

        //zkusíme ho zamknout s -1
      lock cmpxchg [lockState], edx
      //nepředběhl nás jiný procesor?
      //původní lockState je v eax
        test eax, eax
jnz spin
```

SpinLock - odemčení

mov DWORD PTR [lockState], 0

- A je to.

- Odemčení je jednoduché. Ale nedalo by se také udělat něco se zamčením? Přece jenom, soustavné točení se ve smyčce jenom spotřebovává energii. A co když máme mobilní zařízení?

SpinLock – úspora energie

```
//Intel® 64 and IA-32 Architectures Optimization Reference Manual 2011
if (!acquire_lock()){
  /* Spin on pause max_spin_count times before    backing off to sleep */
  for(int j = 0; j < max_spin_count; ++j)
    /* intrinsic for PAUSE instruction*/
    _mm_pause();
    if (read_volatile_lock()) {
      if (acquire_lock()) goto PROTECTED_CODE;
     }
   }

  /* Pause loop didn't work, sleep now */
  Sleep(0);
  goto ATTEMPT_AGAIN;
}
PROTECTED_CODE:
  do_work();
  release_lock();
```

KIV Operační systémy

Meziprocesová synchronizace

Verze 1.0

Meziprocesová synchronizace

- Meziprocesová nebo mezivláknová?
- V Linuxu je thread uvnitř OS reprezentován s PCB, u jiných OS s TCB
- Takže lze univerzálně říci, že se dále budeme bavit o synchronizaci operačním systémem plánovatelných entit
- Výhodou je, že s tímto přístupem pokryjeme i synchronizaci threadu uvnitř procesu
  - Tj. synchronizujeme thready v alespoň jednom procesu

Celočíselný semafor

- Abstraktní datový typ, který kontroluje přístup ke sdílenému prostředku (sdílí ho více vláken)
  - O vlastním, sdíleném prostředku ale neví nic
  - Má limit, kolik threadů může naráz přistupovat ke sdílenému prostředku
    - Binární semafor má tuto hodnotu nastavenu na 1
  - Má počítadlo, kolik threadů už prostředek sdílí
  - Má frontu čekajících threadů, které by chtěly prostředek sdílet

Acquire

- … též známé jako wait, down či P(passering v originále, „***P***usť mě dovnitř“)
- Thread žádá, aby byl vpuštěn semafor a počítadlo semaforu bylo sníženo o *n*, kde *n* bývá zpravidla 1
- Operační systém, který semafor poskytuje, musí atomicky zajistit:
  - Test, zda může být počítadlo sníženo o *n* a zůstat nezáporné
  - Pokud ano, sníží se počítadlo a thread běží dál
  - Pokud ne, počítadlo se nesníží a thread se zablokuje

Acquire – uniprocesor

- Operace acquire musí proběhnout atomicky, takže OS musí nějakým způsobem implementovat kritickou sekci, ve které změní stav semaforu a procesu
- Na uniprocesoru může rovnou měnit příslušné struktury
  - Je však třeba pohlídat, aby během kritické sekce nedošlo k přepnutí kontextu
    - Dočasně lze pozastavit přepínání kontextu (interrupt od hodin se bude ignorovat – buď se nastaví sw vlajka, že jeho obsluha nebude ovlivňovat přepínání kontextu, nebo se hw interrupt zamaskuje)

Acquire – multiprocesor

- Na multiprocesoru nestačí zamaskovat přerušení pro jeden procesor – muselo by se to udělat pro všechny, což opět vyžaduje synchronizaci => tudy cesta nevede
- sw vlajka by musela být v globální paměti všech procesoru a nastavovala by se pomocí atomických operací => to už s nimi můžeme rovnou změnit počítadlo semaforu
- Kód snížení počítadla je založený na cyklu spinlocku

Acquire – spinlock

```
            mov eax, [counter]
spin:
            mov edx, eax
            sub edx, 1              ; n=1
            jns trylock             ; pokud je číslo záporné, není místo a musíme
            call  doblock           ;thread zablokovat, dokud se neuvolní
trylock:    lock cmpxchg [counter], edx    ; zkusíme nastavit novou hodnotu
                                    ;počítadla (edx) je-li stále eax==[counter]
            ;nepředběhnul nás jiný procesor?
            jnz spin                ; předběhnul-li, aktuální [counter] je teď v eax
```

Acquire – uspání threadu

- Není-li možné thread vpustit dále za semafor, OS ho musí uspat
  - Stav threadu se změní na blokovaný
  - Thread se přidá do seznamu threadů čekajících na daný semafor
    - Pokud by bylo n>1, musí se do seznamu přidat i n
  - A do TCB se přidá semafor do seznamu entit, nad kterými je thread blokovaný
- TryAcquire - Namísto toho, aby se thread v Acquire uspal, TryAcquire vrátí příslušnou chybovou hodnotu

TryAcquire - SpinCount

- Spinlock acquire lze vykonávat v uživatelském adresovém prostoru
- Lze se tedy pokoušet o získání přístupu přes semafor předem stanovenou dobu, bez přepnutí do režimu jádra, a pak
  - Acquire zavolá jádro a to thread uspí
  - TryAcquire vrátí řízení uživatelskému kódu threadu bez volání jádra, a to může dělat jinou, uživatelskou činnost
  - Např. viz RTL_CRITICAL_SECTION.SpinCount u WinAPI

Release

- … též známé jako signal, up či V(vrijgave v originále, „pusť mě ***V***en“)
- Thread informuje, že opouští kritickou sekci, a že se má počítadlo semaforu zvětšit o nějaké m, zpravidla m=n=1
- Funkce OS analogicky k Acquire atomicky zvýší počítadlo o m, ale pak se ještě podívá, zda na opouštěném semaforu není blokován nějaký thread, který by mohl pokračovat

Release – vzbuzení threadu

- V základě by stačilo:
  - z neprázdné fronty čekajících threadů na daném semaforu vyjmout ten první
  - z příslušné v TCB odkazované fronty tohoto threadu vyjmout daný semafor
  - a nastavit stav thredu na runnable
- Jenže…
  - Co když thread žádal o Acquire s n>1?
  - Co když je thread uspán ještě z jiného důvodu?

Release – vzbuzení threadu

- Co když thread žádal o Acquire s n>1?
  - Pak je třeba vybrat thread, který byl uspán s n menším nebo rovným počítadlu semaforu
  - Jenže, co když ho předběhne thread s menším?
    - Pak se musí fronta uspaných threadů projít znovu hledat thread s vyhovujícím n.
    - Takže to má ve výsledku takovou režii, že je lepší podporovat n=m=1

Release – vzbuzení threadu

- Co když je thread uspán ještě z jiného důvodu?
  - Např. je-li uspán z debuggeru
  - Thread nebude převeden do stavu runnable, dokud ho bude něco blokovat
  - A tím pádem musí OS z fronty uspaných threadů vybrat další, který by bylo možné zkusit odblokovat
  - Teoreticky by bylo možné odblokovat všechny na semafor čekající thready, protože by se zase v případě neúspěchu uspaly – ale je to moc velká a zbytečná režie navíc
    - => metoda vzbouzení má vliv na celkovou režii

Producent-konzument

- Kruhový buffer, binární semafor pro přístup k buffer[bufsize] , semafor pro zápis a semafor pro čtení
  - Oba inicializovány na limit bufsize, počítadlo pro čtení na 0 a pro zápis na bufsize
  - Binární semafor lze nahradit atomickými operacemi

Producent:	Konzument:
P(pro_zápis),P(buffer)	P(pro_čtení), P(buffer)
vlož_do_bufferu	vyber_z_bufferu
V(pro_čtení), V(buffer)	V(pro_zápis), V(buffer)

Mutex

- Sice má navenek tu samou funkcionalitu jako binární semafor, ale:
  - Může mít vlastníka – jenom ten thread, který ho zamknul ho může odemknout
  - Může poskytovat inverzi priorit
  - Může zabránit ukončení threadu, který mutex uzamknul

Roura

- Roura je buffer, který má dva souborové deskriptory, jeden pro zápis a jeden pro čtení
- A když už má roura souborový deskriptor, může mít i souborové jméno
  - Pojmenovaná roura je pak persistentní, jinak roura zaniká s posledním procesem, který ji mohl používat
- Roura se často využívá k přesměrování výstupu jednoho konzolového programu na vstup druhému

Roura – zápis a čtení

- Buffer roury má omezenou velikost, takže musíme ošetřit, aby thready zapsaly jenom tolik, kolik je v ní místa
- Aplikujme úlohu producent konzument
  - Buffer bude kruhový
  - Producent bude zapisovat n bytů
  - Konzument bude vybírat m bytů
  - => a známe řešení/implementaci na bázi semaforů
    - Které ovšem musíme ošetřit pro specifické případy – např. když producent bude chtít zapsat více bytů, než kolik je velikost bufferu
    - Producentů i konzumentů může být několik

stdin, stdout, stderr - Linux

- V POSIXovém systému uděláme close požadovaného handle
- OS pak jeho číslo použije jakmile vytvoříme nový souborový deskriptor, anebo budeme duplikovat handle

```
dup2(fileno(newstdinopenedfile), STDIN_FILENO);
dup2(fileno(newstdoutopenedfile), STDOUT_FILENO);
 dup2(fileno(newstderropenedfile), STDERR_FILENO);
fclose(newstdinopenedfile);
fclose(newstdoutopenedfile);
fclose(newstderropenedfile);
```

stdin, stdout, stderr - WinAPI

```
PROCESS_INFORMATION piProcInfo;
STARTUPINFO siStartInfo;
BOOL bSuccess = FALSE;


 ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );
 siStartInfo.cb = sizeof(STARTUPINFO);
 siStartInfo.hStdError = g_hChildStd_OUT_Wr;
 siStartInfo.hStdOutput = g_hChildStd_OUT_Wr;
 siStartInfo.hStdInput = g_hChildStd_IN_Rd;
 siStartInfo.dwFlags |= STARTF_USESTDHANDLES;


 bSuccess = CreateProcess(NULL, szCmdline, NULL, NULL, TRUE,  0, NULL,
                                        NULL, &siStartInfo, &piProcInfo);
```

Zprávy

- Významná forma synchronizace MS Windows
  - Zejména u GUI
    - Všechny vizuální prvky jsou window, která přijímají a posílají zprávy
    - Přičemž v main je hlavní smyčka zpráv
      - Konzolové aplikace ji nepotřebují, ale mohou použít
- Jeden thread doručí zprávu druhému threadu
  - Může i nemusí čekat, až ji příjemce zpracuje
- OS spravuje frontu příchozích zpráv per thread

Zprávy – hlavní smyčka

```
int wmain() {
        CreateWindow(….)
        while(GetMessage( &msg, NULL, 0, 0 )) {
          TranslateMessage(&msg);
           DispatchMessage(&msg);
        }
        return msg.wparam;
}
```

- Každé window má svou WindowProcedure, která zprávy přijímá

Zprávy – WindowProcedure

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
    switch (uMsg)    {
    case WM_SIZE:     {
            int width = LOWORD(lParam);  // Macro to get the low-order word.
            int height = HIWORD(lParam); // Macro to get the high-order word.

            // Respond to the message:
            OnSize(hwnd, (UINT)wParam, width, height);
        }
        break;
                return DefWindowProc(hwnd, uMsg, wParam, lParam);
        }
}
```

Zprávy – odesílání

- PostMessage – odešle zprávu, nezajímá ho výsledek
- SendMessage – odešle zprávu, ale je blokován, dokud ji příjemce nezpracuje a nevrátí výsledek (int)
- WM_COPY data – jeden z parametrů je ukazatel na blok paměti, který je při doručení do jiného procesu přístupný v paměti procesu příjemce
  - Lze použít při SendMessage
- Jak to implementovat?

PostMessage – implementace

- OS volá WindowProcedure a serializuje zprávy jí zpracovávané
- Po dokončení WindowProcedure musí OS provést vyjmutí zprávy z fronty zpráv
- Při PostMessage
  - Zpráva se pouze přidá do fronty příjemce, odesílatel pokračuje s vykonáváním kódu
  - Až ji příjemce zpracuje, OS ji vyjme z jeho fronty zpráv

SendMessage – implementace

- Se zprávou musí být svázaný nějaký synchronizační prostředek, nad kterým se odesílatel uspí/bude blokován, dokud příjemce nezpracuje odesílanou zprávu
  - V principu jde o to samé, jako u semaforu
- Až příjemce zprávu zpracuje, tj. kód OS dostane řízení po návratu z WindowProcedure, OS překopíruje eax příjemce do eax odesílajícího (tj. zkopíruje návratovou hodnotu), a zruší blokaci odesílajícího nad SendMessage

WM_CopyData– implementace

- Speciální zpráva, umožňující předat velké množství dat
- WinAPI říká, že:
  - odesílající nemá modifikovat odesílaný blok paměti, dokud SendMessage neskončí
  - Příjemce nemá tento blok paměti modifikovat, jako by pro něj byl read-only
- Windows jsou sice privátní OS s uzavřeným kódem, ale možnou implementaci si už lze dovodit

WM_CopyData– implementace

- Při SendMessage OS dočasně namapuje stránky odesílajícího procesu do adresového prostoru příjemce, a nechá je jako read-only
- Před voláním WindowProcedure příjemce ale OS musí upravit pointer ukazující na předávaná data tak, aby tento pointer ukazoval na předávaný blok paměti na adresy, kam byly stránky namapovány
- Po návratu z WindowProcedure OS stránky odmapuje
- Chce-li si příjemce data ponechat, musí si je sám zkopírovat – což říká i dokumentace WinAPI

Signály

- Významná forma synchronizace v POSIXu
- Obsluha signálu je rutina, identifikovaná číslem, která se vyvolá při události, jíž toto číslo odpovídá
  - Podobnost s tabulkou vektorů přerušení
- Např. uživatel konzolové aplikace stiskne Ctrl+C
  - OS transformuje stisk této klávesy na signál SIGINT (signal to interrupt the process) a naplánuje vykonání příslušné obsluhy
  - Co se stane dále, to závisí na tom, co daná obsluha signálu dělá

Signály – default handler

- Co by se stalo, kdyby chtěl OS vykonat obsluhu signálu, ale proces by pro něj nenastavil obslužnou rutinu?
  - Buď by došlo na Segmentation Fault (což je signál SIGSEGV, u MS vyjímka Access Violation), anebo by se začal vykonávat náhodný kód, což by nejspíš také skončilo vyjímkou
- OS každému procesu při jeho vytvoření poskytne tzv. default handler pro každý signál
  - Dokud proces nenastaví svou obsluhu, vykonává se obsluha OS

Signály – user handler

- Např. default handler SIGINT ukončí proces
- Když ale proces bude provádět nějakou činnost, např. kopírování souboru, jeho programátor může chtít, aby Ctrl+C pouze ukončilo aktuální činnost
- => proces si nainstaluje vlastní obsluhu SIGINT, která pouze ukončí aktuální činnost

Signály – ignorování

- Většinu signálů lze také ignorovat – tj. pokud nastanou, neprovede se žádná obsluha
  - Ani uživatelská, ani OS
- Vyjímkou jsou dva signály, které nelze ignorovat, ani pro ně nastavit vlastní obsluhu
  - SIGKILL – ukončí proces
  - SIGSTOP – zastaví proces

Signály – Process-Directed

- Podle POSIXu všechny jádrem plánované thready mají mít stejný PID
- Takže pokud se má vykonat obsluha signálu, OS vybere thread, který ji vykoná – Process-Directed signal
- Nicméně, thread má možnost pomocí pthread_sigmask možnost ignorovat signály pro sebe a své potomky
  - Takže je možné zpracovat signály v jednom, konkrétním threadu
  - Množina threadů, ze které OS vybírá, se omezí na jeden prvek
  - Bude-li prázdná, ukončí se celý proces

Signály – Thread-Directed

- Vedle Process-Directed signálů jsou také Thread-Directed
- Funkce *kill má jako jedne z parametrů číslo signálu
- Funkce kill pošle signál procesu
  - Např. SIGKILL
- Funkce pthread_kill pošle signál konkrétnímu threadu
  - Např. SIGSEGV je signál pro konkrétní thread, ve kterém došlo k neoprávněnému přístupu do paměti
  - Jestliže thread nemá definovanou obsluhu pro signál, jehož defaultní akcí je ukončení procesu, není ukončen thread, ale celý proces

Signály – fork, exec

- Potomek nebude mít signalizován žádný signál, i kdyby jeho rodič měl
- Uživatelské obsluhy a ignorování signálů je zděděno
- Exec přepíše stávající kód novým kódem
  - Přepíše i kód stávajících uživatelských obsluh signálů
  - => všechny obsluhy signálů jsou nastaveny do výchozího stavu
    - Jinak by se stalo to, co je popsáno na slidu „Signály – default handler“

Signály – implementace

- Per thread/proces, jádro si udržuje
  - seznam obsluh signálů,
  - spinlock, který chrání přístup k nim,
  - 64-bitovou masku ignorovaných signálů
  - 64-bitovou masku signálů čekajících na obsluhu
  - Obousměrný spojový seznam signálů čekajících na obsluhu
    - Každá položka ještě obsahuje OS-specifické info
- Standardní signál vždy čeká jenom jeden
  - Ještě jsou rt-signály

Real-Time signály

- Existuje 32 standardních signálů (0-31) s předdefinovaným významem a 32 real-time signálů (32-63) bez předdefinovaného významu; akce neobslouženého rt signálu je terminate process
  - Real-time indikuje, že mají být obslouženy ASAP – ne real-time
  - Používá je např. Native POSIX Thread Library pro běh programů využívajících POSIX threads
- Od standardních signálů se liší
  - Ve frontě může být několik instancí jednoho rt signálu
  - Jsou doručovány v garantovaném pořadí
  - Obsahují další systémové info

Signály vs. zprávy

- Konzolová aplikace ve Windows může nastavit obsluhu např. pro Ctrl+Z
- Ekvivalentem SIGKILL je ve Windows GUI zpráva WM_QUIT
- Linux má 64 signálu, navržené s ohledem na konzolové aplikace, Windows 2^sizeof(int) zpráv navržené s ohledem na GUI aplikace
- GUI v Linuxu používá jiné mechanismy, např. signály a sloty dle Qt, které se dají provozovat i pod Windows

KIV Operační systémy

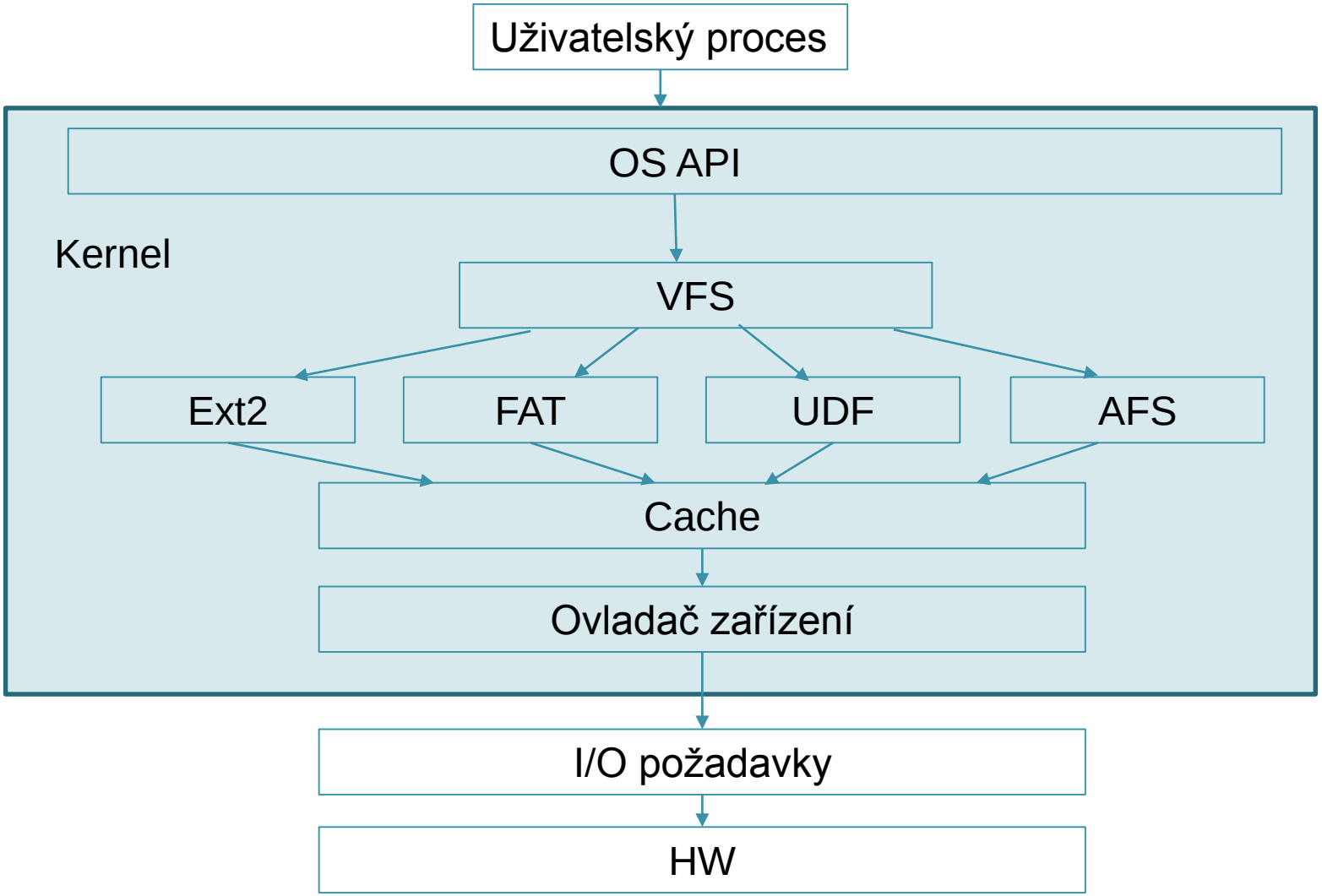
Souborový systém

Verze 1.0

Virtual File System

- Thread volá jednotné API pro práci se soubory
  - Adresář je jenom speciální typ souboru
- Souborový systém je abstrakce nějakého souvislého bloku paměti, které ho umožňuje organizovat do souborů – tj. do menších, pojmenovaných bloků paměti
- Typicky je blok paměti hw realizovaný diskem, ale může to být i síťový protokol nebo část RAM
- Každý blok paměti může mít jiný souborový systém
- OS musí zpřístupnit jednotné API – Virtual File System

Virtual File System



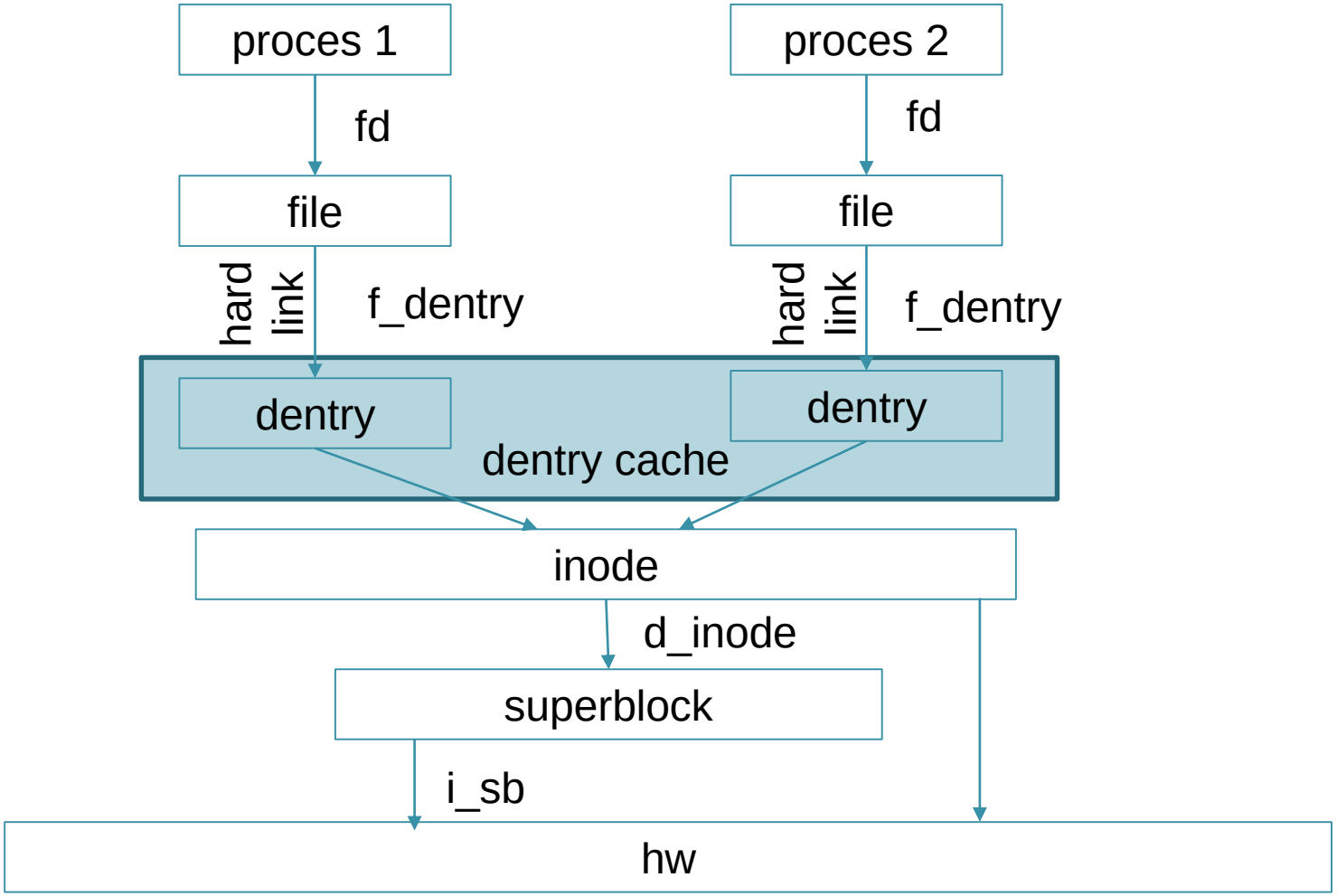
VFS koncept

- VFS je sice obecný model souborového systému, ale má design podle UNIXu
  - Souborové systémy, které byly navržené podle jiných pravidel, se mu musí přizpůsobit – např. FAT nemá koncept inode
- Hlavní komponenty VFS:
  - superblock – info o připojeném souborovém systému
  - inode – info o konkrétním souboru
  - file – info o konkrétním, otevřeném souboru
  - Dentry – info o adresářové položce

VFS koncept

- VFS je sice obecný model souborového systému, ale má design podle UNIXu
  - Souborové systémy, které byly navržené podle jiných pravidel, se mu musí přizpůsobit – např. FAT nemá koncept inode
- Hlavní komponenty VFS:
  - superblock – info o připojeném souborovém systému
  - inode – info o konkrétním souboru
  - file – info o konkrétním, otevřeném souboru
  - Dentry – info o adresářové položce

Virtual File System



VFS tabulka funkcí

- V OOP bychom nadefinovali abstraktní třídy jednotlivých objektů VFS, a konkrétní implementace souborových systémů by je implementovaly
- Linux má C rozhraní, a v C není OOP
- =>každý VFS má definovanou sadu funkcí, které nad ním lze provádět a instance každého objektu je reprezentována tabulkou ukazatelů na funkce
  - Obdoba VMT
  - Některé generické funkce může poskytnout už OS

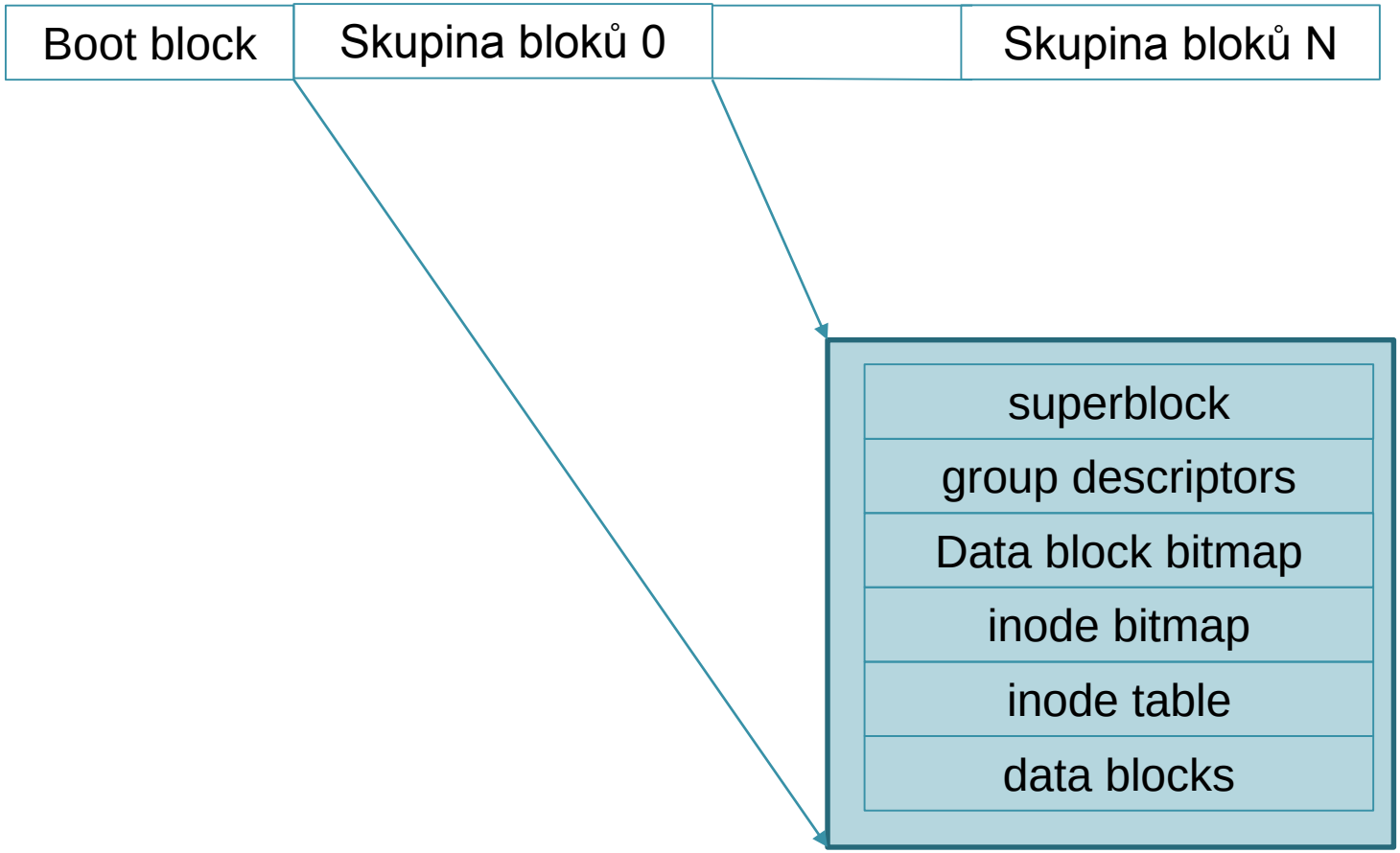
Proces a soubory

- Každý proces má svůj pracovní a kořenový adresář
  - Uloženo ve *fs_struct* ve *fs* položce PCB
- Otevřené soubory jsou uloženy ve *files_struct* v položce *files* PCB
  - Funkce otevření souboru open() vrací index do pole objektů *file* (položka *fd* ve *files*)
  - current->files->fd[0] je stdin, 1 stdout a 2 stderr
  - Po uzavření close() nějakého fd, open() vrátí první volný index do fd – tj. lze takto přesměrovat stdio

Extended File System 2

- ext1 … ext4, klíčový pro pochopení je ext2
- Návrhovým vzorem odpovídá VFS
- Skládá se z bloků, které seskupuje
  - Velikost bloku je od 1 do 4kB
  - Volitelný počet inode
  - Prealokuje bloky souborům před tím, než jsou použitý
  - První blok je vždy vyhrazen pro boot sector

Struktura ext2



- První skupina bloků, tj. 0, je vyhrazena pro jádro OS

superblock

- Obsahuje
  - Celkový počet inode uzlů
  - Velikost souborového systému v blocích
  - Počítadla volných bloků a inode uzlů
  - Velikost bloku
  - Počty bloků a inodů uzlů ve skupině
  - 128-bit id souborového systému
  - Počítadlo připojení
  - A další

group descriptor

- ext2_group_desc Obsahuje
  - Počty bloků bitmap bloků, inode uzlů a prvního inode v tabulce bloků
    - Pokud je n-tý bit bitmapy nastaven, daný blok/inode je použit
  - Počty volných bloků, inode uzlů a adresářů v bloku
  - A další

Tabulka inode uzlů

- Jsou to po sobě uložené bloky obsahující záznamy typu ext2_inode, tj. o stejné velikosti
- ext2_inode obsahuje
  - Typ souboru a přístupová práva
  - Identifikátory vlastníka a skupiny
  - Délku v bytech a blocích
  - Časová razítka
  - Pole ukazatelů na datové bloky
  - A další

Typy souborů

- inode.filetype
  - Běžný soubor – potřebuje datové bloky, kde ukládá data
  - Adresář – speciální typ souboru, jeho datové bloky ukládají jména souborů v adresáři společně s jejich čísly inode uzlů
  - Symbolický odkaz – do 60 bytů se odkaz ukládá v inode (tzv. fast symbolic link), jinak také potřebuje datové bloky

ext2 paměťové struktury

- Časté operace nad souborovým systémem vyžadují frekventovaný přístup k některým datovým strukturám
  - => lze je při připojení systému nahrát do paměti, aktualizovat je tam a průběžně a při odpojení je nahrát na disk
- OS sice drží část disku v diskové cache, ale proč tam zabírat místo datům něčím, co stejně potřebujeme v mít paměti?

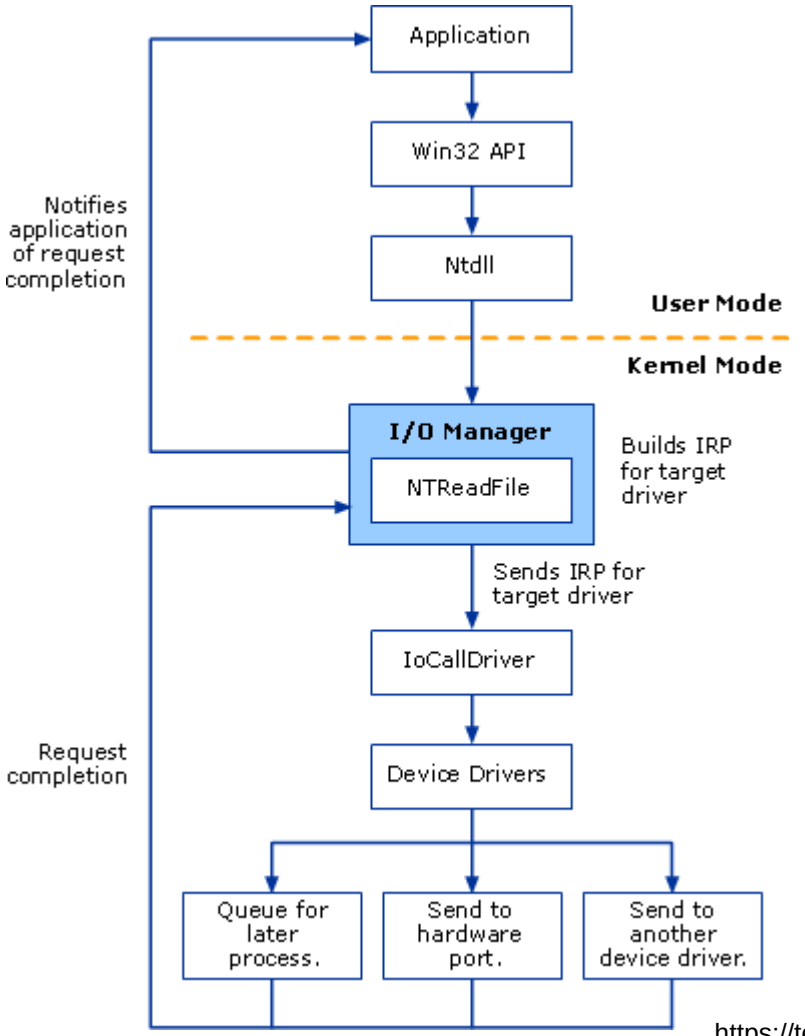
Installable File System

- VFS Windows
- Funguje ve třech režimech
  - nemusí být nutné implementovat všechny tři
  - file system – vytváří vlastní souborový systém na diskem
  - Mini Filter – rozhraní pro antivirové programy a indexovací služby
  - FS FilterDriver – používá se pro úpravu již existujících souborových systémů
    - Zachycuje požadavky a vrací modifikované odpovědi
- Používá IO Request Packet pro komunikaci

IO Request Packet

- Struktura používaná ke komunikaci mezi ovladačem zařízení a OS
- Popisuje požadavky, které se mají se provést
- Dávají se do fronty, kterou si OS může přeuspořádat
- Většinou je vytváří I/O manager podle volání souborových funkcí z uživatelského adresového prostoru
- Ale mohou je vytvářet i další části, např. systém úspory energie bude chtít při nečinnost vypnout disk

IO Request Packet

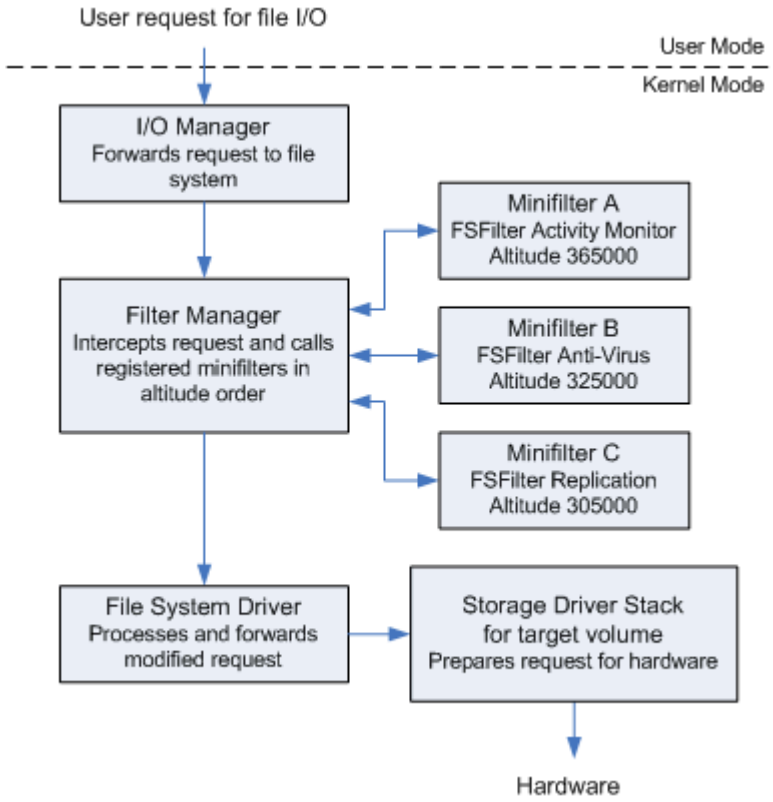


https://technet.microsoft.com/en-us/library/cc776371%28WS.10%29.aspx

Filter Manager

- Aktivuje se při načtení mini filtru
- Připojí se do file system stacku daného disku
  - Respektive zaregistruje se pro I/O operace, které bude chtít filtrovat
- Filtry jsou ve stacku podle priorit, např. antivir má větší prioritu než indexovací či replikovací služba

Filter Manager



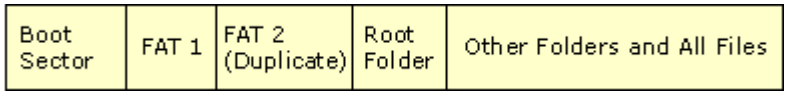
https://msdn.microsoft.com/en-us/library/windows/hardware/ff541610%28v=vs.85%29.aspx

Fast I/O

- IRP je výchozí mechanismus pro všechno
  - Synchronní i asynchronní přenosy, data v cache i mimo ni, výpadky stránek
- Ale pro synchronní operace nad daty v cache lze použít Fast I/O
- Data jsou pak přenesena přímo mezi buffery procesů přes systémovou cache
  - Zkratka, která obejde celý fs a hw stack – něco jako loopback na localhostu
- Fast I/O dělá v případě neúspěchu fallback na IRP

File Allocation Table

- První verze byla nasazena roku 1977 na 8 palcových disketách, v dalších verzích se používá dodnes
  - Digitální kamery, bootování UEFI, USB čitelné různými OS…
- Disk s FAT má bootovací sektor, alokační tabulku souborů, její kopii, kořenový adresář a zbývající adresáře a soubory



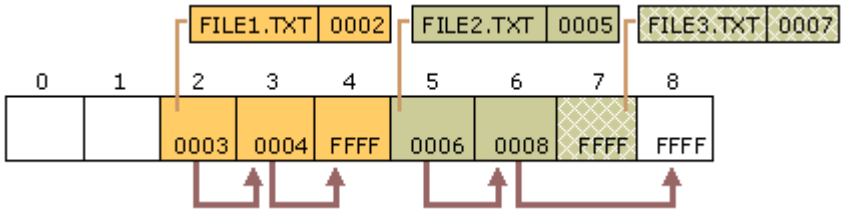
https://technet.microsoft.com/en-us/library/cc938438.aspx

FAT – položka adresáře

- Obsahuje
  - Jméno ve formátu 8.3
    - Dlouhá jména řeší VFAT a pozdější verze
  - Atributy
    - Disk, adresář, soubor
      - Skrytý, systémový, jen ke čtení
  - Časová razítka
  - Číslo prvního clusteru, kde začínají data položky adresáře
  - Velikost souboru
  - A další

FAT alokace souborů

- Souborům se při zápisu přiděluje první volný cluster
  - FAT disk je rozdělen na bloky nazývané clustery
  - Clustery nemusí jít po sobě => problém fragmentace
- Každý cluster obsahuje číslo dalšího cluster, který obsahuje další data daného souboru, anebo značku konce souboru



https://technet.microsoft.com/en-us/library/cc938438.aspx

New Technology File System

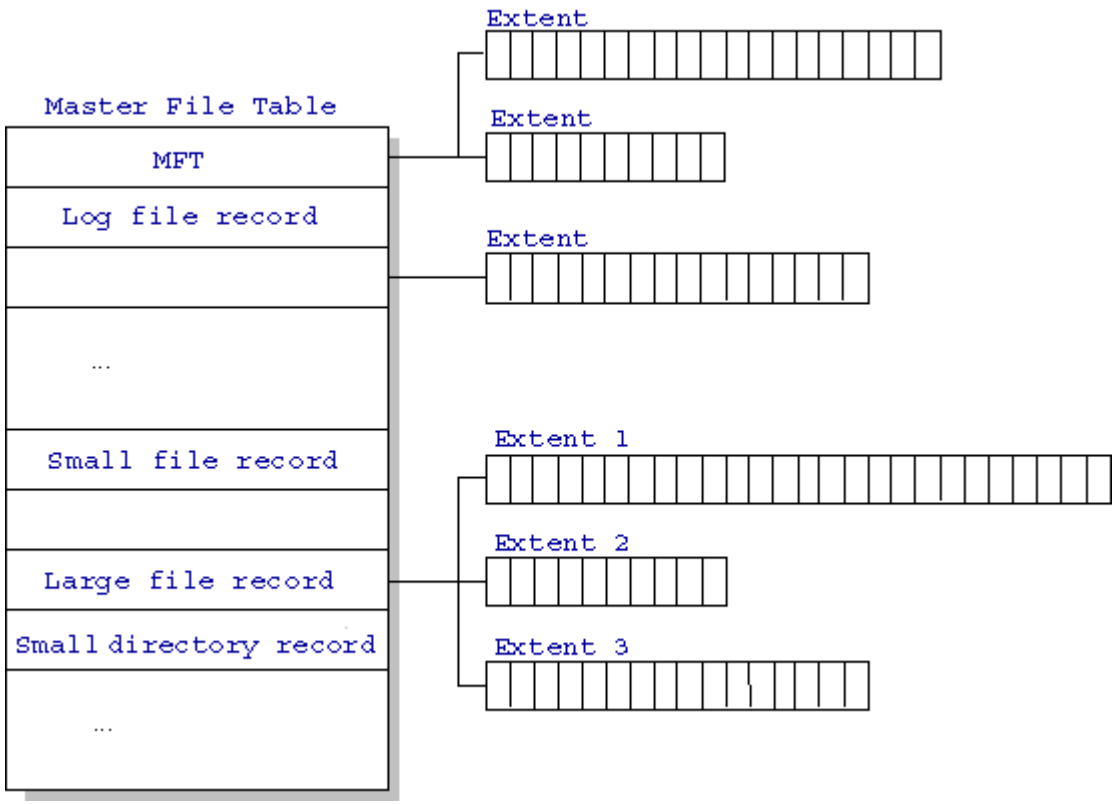
- Navržen pro Windows NT, aby na rozdíl od FAT a HPFS (fs vyvíjený s IBM pro OS/2) uměl
  - metadata, ACL, journaling, datové streamy, atd.
  - ale hlavně, aby se zvýšila rychlost, spolehlivost a zlepšilo se využití místa na disku
  - Např. od Vista umí i transakce
- Detailní srovnání např. NTFS, ext4, či btrfs je mimo rozsah a je ponecháno na laskavém čtenáři „as needed“

NTFS disk

- Disk naformátovaný s NTFS má následující strukturu:
  - Boot sektor
  - Master File Table
  - Master File Table zóna, do které může MFT růst
  - Systémové soubory
  - Uprostřed disku je kopie (části) MFT
  - Zbývající místo je určeno pro soubory

NTFS's MFT

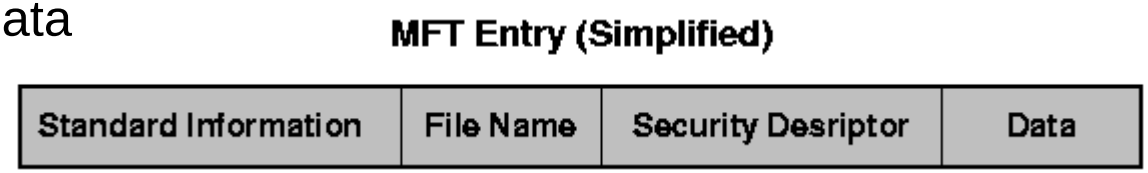
- Každá položka představuje jeden soubor



http://ntfs.com/ntfs-mft.htm

MFT položka

- Položka se sestává z dvojic <atribut, hodnota>, takže ji lze dynamicky rozšiřovat, a obsahuje
  - Standardní informace
    - Práva, časová razítka, hard-link count (kolik adresářů na soubor ukazuje)
  - Jméno souboru
  - Security descriptor
  - Data



http://pages.cs.wisc.edu/~bart/537/lecturenotes/s26.html

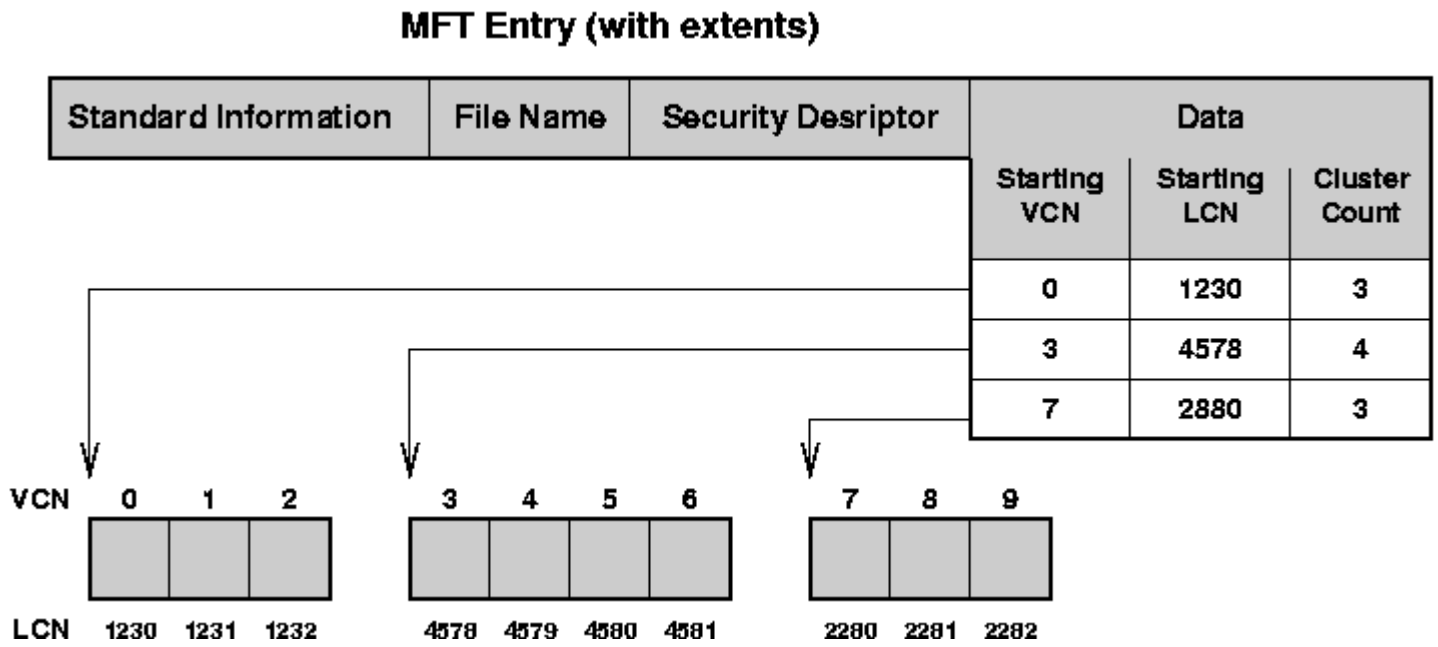
MFT položka souboru

- Každý soubor se skládá alespoň z jednoho data stream
  - type soubor.txt:druhytext
- Pokud jsou celá data souboru větší než místo v MFT, pak položka data odkazuje na další místo na disku, kde jsou data uložena
- Dostatečně malé souboru jsou uloženy přímo v MFT
  - Viz fast symbolic link u ext2

MFT data pointer

- Pointery na data jsou pointery na posloupnost logických clusterů na disku (extent)
- Každá posloupnost má
  - VCN – virtual cluster number, první cluster souboru
  - LCN – logical cluster number, první logický cluster jedné sekvence
  - Délka v počtu clusterů
- MFT obsahuje list položek popisujících extent
  - Unix-like používá strom

MFT data pointer

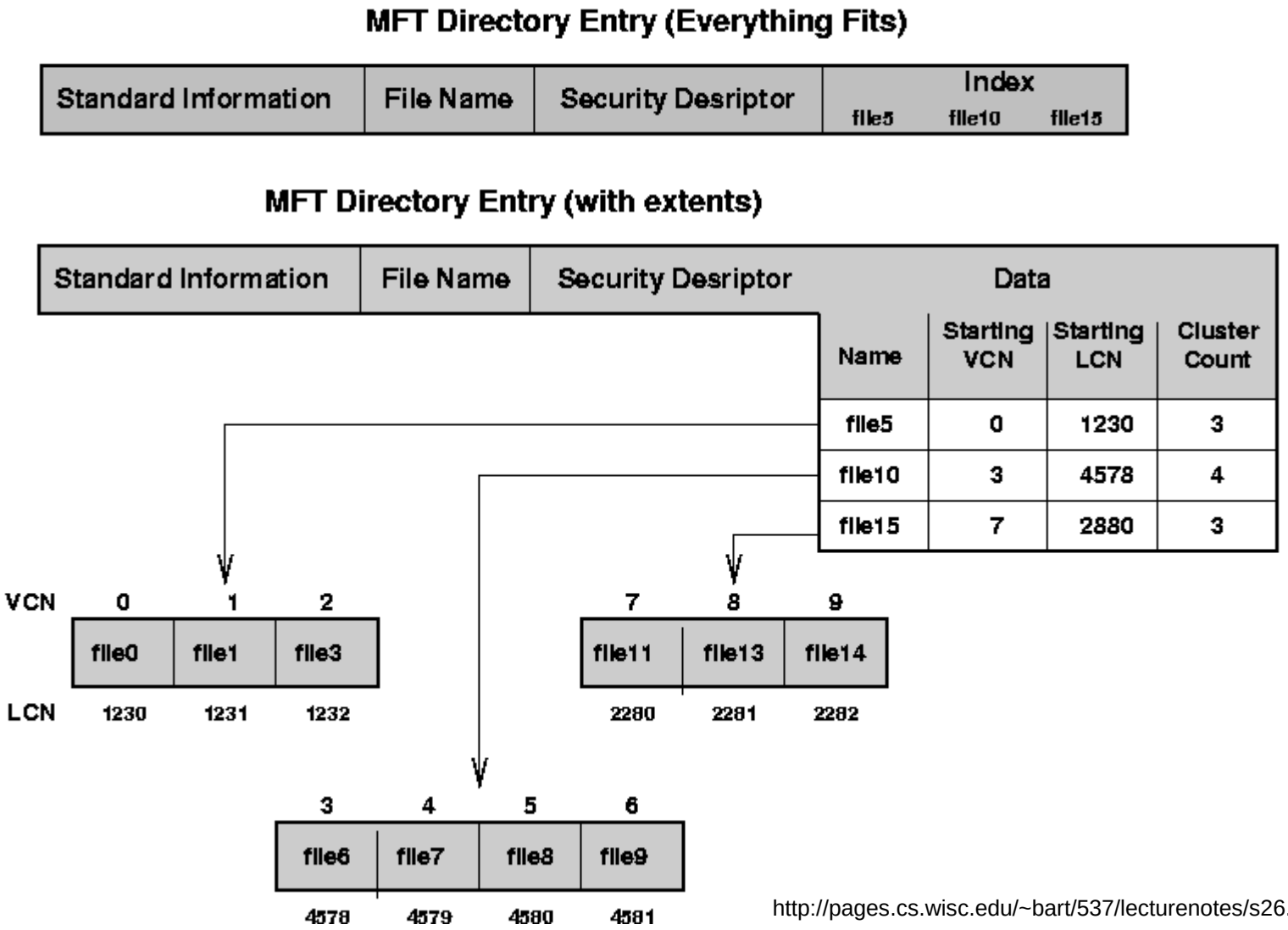


http://pages.cs.wisc.edu/~bart/537/lecturenotes/s26.html

MFT položka adresáře

- Adresář je speciální soubor obsahující seznam souborů
- Adresář má jméno a referenci
  - Reference je pár <číslo souboru, sekvenční číslo>
  - Číslo souboru je offset do MFT
    - Něco jako číslo inode uzlu ve VFS
- Položka adresáře obsahuje seznam souborů v B+ stromu
  - Jméno souboru je jak v položce adresáře, tak i v MFT
- Pokud je záznam adresáře dostatečně malý, je v MFT

MFT položka adresáře



http://pages.cs.wisc.edu/~bart/537/lecturenotes/s26.html

KIV Operační systémy

V/V zařízení

Verze 1.0

Konfigurace směrovače konzolí

- Např. viz cvičení KIV/PSI, směrovač lze konfigurovat po připojení konzolového portu směrovače do sériového portu počítače
  - BIOS sice poskytuje funkce pro práci se sériovými porty, ale jsou to rutiny pro real-mode, ve kterém není izolace procesů
  - Komunikace se sériovým portem se tak odehrává pomocí portů, na x86 instrukcemi in a out, což jsou privilegované operace
  - S periferií tak může ve skutečnosti komunikovat pouze jádro OS a procesům periferii jenom virtualizuje

Tiskárna

- Pokud bychom např. místo směrovače uvažovali lokální tiskárnu na (dnes již de-facto u PC nepoužívaném LPT), pak by současný přístup několika procesů k fyzicky jedné tiskárně vedl k promíchání příkazů a nesmyslnému výstupu z tiskárny
- Jádro OS tedy vedle virtualizace hw zajišťuje i celistvost prováděných (tiskových) úkolů
  - U tiskárny serializuje tiskové úlohy
  - U sériového portu povolí komunikaci jenom jednomu procesu

Disk

- Co když bude chtít několik procesů číst z disku?
- Stejné jako s tiskárnou a sériovým portem
- OS zde ale má možnost optimalizace/prioritizace
  - Např. budeme mít magnetický disk s plotnami a záznamovou hlavou
  - OS pak může přeuspořádat požadavky na disk tak, aby se s hlavou hýbalo co nejméně => tišší a rychlejší chod
    - Dělal např. Novell Netware
    - Dnes na to mají disky Native Command Queuing

VGA – provedení akce

- VGA je hw, který lze ovládat jak pomocí přerušení, tak pomocí zápisů na porty
- Např. změna grafického režimu

```
mov ax, 13h          ;320x200px a 256 barev v paletě
int 10h
```

- Např. změna barvy v paletě

```
outportb(0x3c8, 1);          //číslo barvy v paletě
outportb(0x3c9, 255);        //červená složka barvy
outportb(0x3c9, 0);          //zelená složka barvy
outportb(0x3c9, 0);          //modrá složka barvy
```

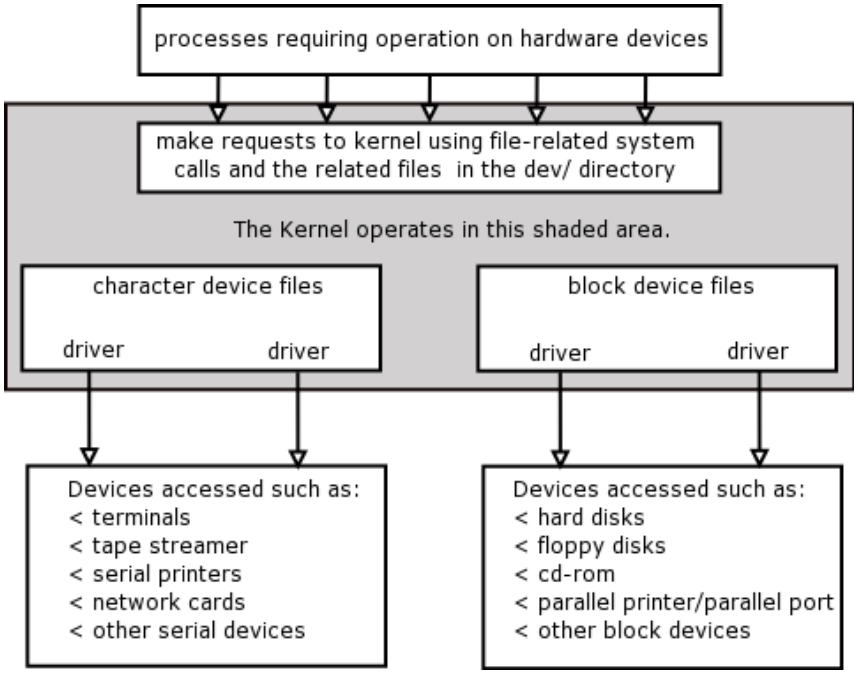
VGA – počkání na dokončení

- V době elektronových obrazovek, bylo dobré měnit paletu v době, kdy se svazek elektronů přesouval z konce obrazu na jeho začátek
  - I z dnešního pohledu jde stále o to, že se synchronizuje FPS s obnovovací frekvencí monitoru
- VGA to ovšem neoznámí přes IRQ, takže je nutné dělat polling stavové informace

```
do { //čeká, dokud se svazek nevrátí zpět
} while (inportb(0x3DA) & 8); //nekreslí se obrazovka


do { //čeká, dokud se svazek nezačne opět vracet
 } while (!(inportb(0x3DA) & 8)); //kreslí se obrazovka
```

Bloková a znaková zařízení

- Bloková zařízení – ovladač zařízení komunikuje s hw pomocí bloků dat velkých několik bytů
  - Např. u disků se čtou a zapisují celé sektory
  - Pro urychlení práce se zařízením, je-li to možné, se cachují právě tyto bloky dat
- Znaková zařízení
  - Komunikuje se sekvenčně po jednom bytu, nemá cache



- Každý hw má své id; id může mít strukturu
  - Např. typ zařízení a pořadí ve skupině daného typu

http://www.learnlinux.org.za/courses/build/internals/ch06s05.html

I/O API

- Programátor nepracuje přímo s konkrétním hw, ale volá funkce operačního systému
  - Např. může volat funkce pro čtení a zápis do souboru, přičemž je konkrétní, pro uživatelský proces virtualizovaný hardware identifikovaná pomocí souborového deskriptoru
  - Tj. dochází k využití infrastruktury, kterou jsme si již ukázali u souborových systémů
  - Dále si tedy ukážeme, co se děje na úrovni kernelu, který pracuje se skutečným, nevirtualizovaným hw

I/O subsystém

1. Po zavolání příslušného API, OS vytvoří I/O požadavek (request), který předá dál (block) I/O subsystému.
2. Volající thread se pozastaví, než bude I/O požadavek dokončen.
3. I/O požadavek se zařadí do I/O fronty. Až přijde na řadu, ovladač provede požadovanou akci s hw.
4. Po dokončené se příslušný thread převede do stavu runnable.
    - U síťové karty je operace dokončena ihned po předání příkazů hw. Ale např. u disku může hw provádět akci dále a její dokončení teprve oznámí – např. pomocí IRQ.

Ne2000 (NIC)

- Jednoduchý, referenční design chipsetu síťové karty od Novellu, který se stal populárním

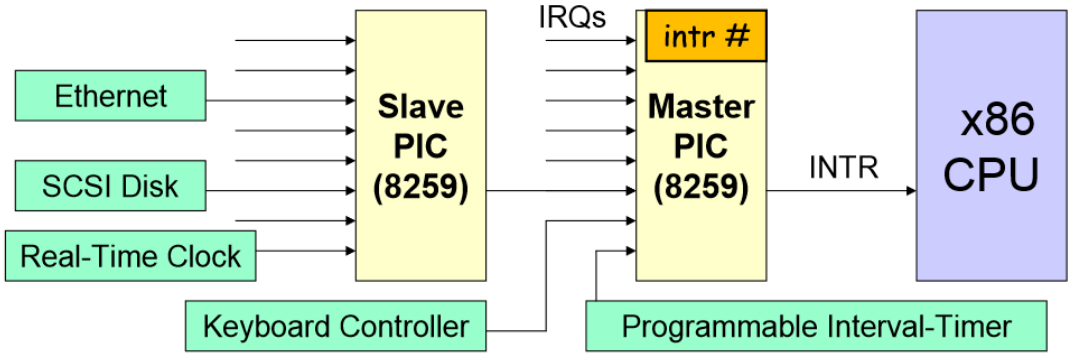
- Odesílání dat
  - Kombinace odesílání příkazů na konkrétní porty a čekání, až budou konkrétní stavové bity shozeny či nastaveny
- Přijímání dat
  - Nejprve hw vygeneruje IRQ, čímž oznámí přijetí dat
  - A teprve poté se vykoná obsluha, která data přečte

Obsluha IRQ

- Jakmile dojde u konkrétního hw k nějaké události, kterou je třeba obsloužit pomocí sw, tento hw vygeneruje tzv. interrupt request – IRQ
  - Stisk klávesy, data přijatá síťovou kartou, hodiny, dělení nulou, atd.
- IRQ je tedy hw signál, který je zaslán CPU
- CPU následně zastaví aktuálně vykonávaný program a vykoná obsluhu přerušení, která přísluší danému IRQ
  - K čemuž použije tabulku vektorů přerušení, protože na tyto čísla jsou namapované čísla jednotlivých IRQ

PIC

- Programmable Interrupt Controller
- Dokáže obsloužit 8 IRQ, používaly se proto 2
  - Master PIC
    - IRQ0 (hodiny) až IRQ 7 (LPT1); IRQ2 je kaskádově Slave PIC
  - Slave PIC
    - IRQ8 (RTC) až IRQ15 (sekundární ATA kanál)



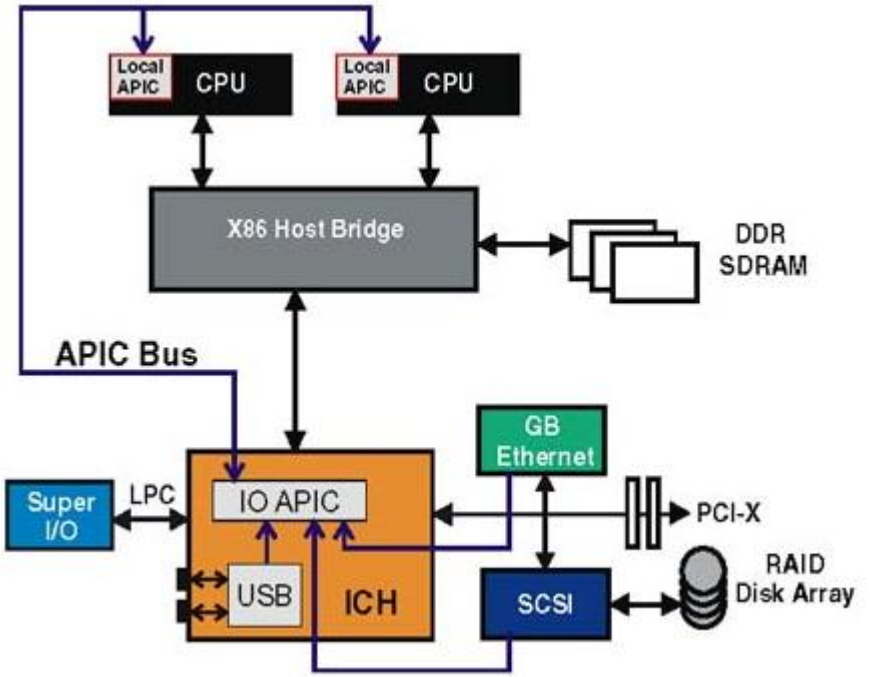
http://www.cs.columbia.edu/~junfeng/10sp-w4118/lectures/l04-syscall-intr.pdf

IRQ konflikt/sdílení

- Co se stane, když budou chtít dvě periferie sdílet to samé IRQ?
  - Např. zvuková karta a tiskárna na LPT1 budou sdílet IRQ5
  - Pokud se nebudou zařízení používat najednou, může to projít, pokud bude obsluha přerušení správně reagovat (známé jako IRQ sdílení)
  - Anebo také může dojít k tomu, že se celý počítač „zamrzne“
- Systémově se situace vyřešila pomocí
  - Message-Signaled Interrupts – zařízení už nepředpokládá, že má dedikovanou linku do PIC, ale že je tato linka možná sdílená, a proto po ní vysílá své ID
  - Advanced PIC

Advanced PIC

- Umožňuje směrovat mnohem více IRQ než PIC
- Umožňuje konstrukci SMP
  - Každý CPU má svůj Local APIC
    - 224 IRQ
    - prvních 0-31 IRQ je vyhrazeno
      - Vyžaduje Message-Signaled Interrupts
      - Někdy se jim říká Virtual IRQs
  - V systému je ještě I/O APIC
- (Naprogramováním) Lze směrovat jednotlivé IRQ na jednotlivé CPU



http://flylib.com/books/en/3.356.1.153/1/

Top Half

- Přerušení musí být obslouženo co nejrychleji, jinak nám může „počítač zmrznout“ stejně jako u IRQ konfliktu
- Když po IRQ CPU spustí příslušnou obsluhu přerušení, tato obsluha vykoná jenom to nejnutnější a zbytek činnosti se odloží na pozdější dobu
  - Tj. nevykonaná práce přidá se někam do fronty
  - Ale hw se řekne, že jeho IRQ bylo úspěšně obslouženo!
- Obsluha přerušení vykonává jenom tzv. „Top Half“

Bottom Half

- V systému běží několik „démonů“, kteří periodicky vykonávají odložené činnosti, které vznikly v důsledku Top Half obsluhy přerušení
  - Odložená činnost (deferred work) se jmenuje Bottom Half
  - Např. UDP datagram nedostanete jak přijde, ale až se vykoná příslušná Bottom Half!
- Ne každá Bottom Half má stejnou důležitost => tj. prioritu
- Obsluhu Bottom Half můžeme rozdělit do dvou skupin:
  - Kritická – v Linuxu se jí říká SoftIRQ
  - Plánovatelná – v Linuxu se jí říká Tasklet a vykonává ji SoftIRQ

Linux: SoftIRQ vs Tasklet

- SoftIRQ
  - Staticky alokovaný a přiřazený konkrétnímu CPU
  - Obsluhuje časově kritické události, časovače (a v Linuxu síť a SCSI)
  - SoftIRQ stejného typu může zároveň běžet na několika CPU v SMP
  - Jeden ze SoftIRQ vykonává tasklety
- Tasklet
  - Lze je deklarovat staticky i dynamicky
  - Tasklety mohou běžet zároveň, ale musí být různého typu
    - Tj. může být spuštěn maximálně jeden Tasklet konkrétního typu
  - V Linuxu se dále dělí na dvě priority HI_SOFTIRQ a TASKLET_SOFTIRQ
  - Musí dokončit svou činnost atomicky, nelze ho uspat
  - Poběží na tom samém CPU, které ho naplánovalo

Linux: Workqueue

- (deferred work) SoftIRQ a Tasklet „co nejdříve“ vykonává jádro při různých příležitostech
  - Např. při dokončení poslední obsluhy přerušení (ISR), nicméně konkrétní „kdy“ závisí na konkrétní architektuře a jádru
- (delayed work) Workqueue je také odložená práce, ale už nemusí být vykonána „co nejdříve“
  - Je to mechanismus, kterým si jádro řekne, že daná práce se udělá někdy v budoucnu
- Každá Workqueue má v jádře svůj vlastní thread
  - Respektive, může mít jeden thread na každý jeden CPU v SMP
- Protože každý ovladač nepotřebuje svou vlastní Workqueue, v systému existuje tzv. Shared WorkQueue

IRQ Polling

- Např. IRQ hodin, dělení nulou nebo výpadek stránky budou vždy obslouženy pomocí přerušení
- Ale jiná IRQ mohou být obsloužena pomocí tzv. pollingu
  - Např. viz příklad s diskem u I/O subsystému, driver nemusí čekat až disk oznámí dokončení akce pomocí IRQ, ale může periodicky číst (tzv. poll) jeho stavové bity
  - Nelze sice automaticky povědět, že IRQ je špatný a okamžitě zavržení hodný
  - Ale, máme-li obsluhovat obrovské množství I/O požadavků, polling řeší několik aspektů, nad kterými je třeba se zamyslet

Interrupt vs. Polling

- Při IRQ pollingu ovladač čeká ve smyčce, dokud není práce vyřízena – respektive, dokud nemá prázdnou frontu I/O požadavků
- Když se generuje velké množství IRQ CPU musí držet krok co do počtu změn kontextu tam a zpět
  - Při stovkách tisíc IRQ za sekundu ho ale hw nakonec „uštve“
  - U tzv. short-wait by se zase I/O thread uspal, plánovač pak spustí jiné vlákno a to začne přepisovat cache CPU, což zpomalí I/O thread, až bude čekat na opětovné nahrání svých dat do cache
  - Interrupt coalescing – hw generuje IRQ až od nějakého počtu událostí nebo dojde k timeoutu, po který se na ně čeká

DMA - motivace

- Direct Memory Access
- Čtení dat ze zařízení pomocí portů a přerušení může fungovat rozumně pouze tehdy, pokud se jedná o malý počet dat
  - Např. modem s rychlostí 9600 baudů vyvolá přerušení cca každé 2ms při přenosu cca 1 znaku každou milisekundu
  - I disketová mechanika přenáší příliš velký objem dat
  - A co teprve moderní disk nebo síťová karta?
  - => potřebujeme mechanismus, který přenese data z paměti systému do paměti zařízení – DMA

DMA - použití

- V systému je omezený počet DMA kanálů a nelze je sdílet
  - Ovladač musí vždy počkat, až je některý kanál volný
  - A ne každé zařízení může použít libovolný DMA kanál
  - => jádro si udržuje seznam DMA kanálů a mj. ke každému zná jeho číslo a stav, zda je volný
- CPU zahájí přenos dat a dále se pak může věnovat jiné činnosti – zařízení oznámí konec přenosu přes IRQ
  - Legacy hardware (disketa, paralelní port, IrDA,…) používají 16-bitovou adresu – DMA je pak omezen na prvních 16MB fyzické paměti
  - DMA s PCI sběrnicí ve výchozím režimu používá 32-bitovou adresu, ale v double-address cycle mapping umí 64-bitovou adresu

KIV Operační systémy

Virtualizace

Verze 1.0

Motivace

- Mějme konkrétní operační systém a konkrétní procesor
  - Programy zkompilované pro daný procesor a režim, ve kterém tento procesor běží, běží za takových podmínek nativně
  - Ale co když potřebujeme spustit program, který byl napsaný pro jiný režim procesoru, nebo jiný procesor, nebo dokonce pro jinou procesorovou architekturu?
  - Pak potřebujeme buď emulaci nebo virtualizaci

Emulace

- Softwarovým řešením vytváříme iluzi skutečného hardware
- Můžeme pak např. na ARMu spustit DOSBox – tj. staré programy pro x86
  - Nebo když potřebujeme spustit něco, co běží na hardware, který (už) nemáme k dispozici
- Jedná se sice o univerzální, ale výpočetně náročné řešení
  - V emulovaném prostředí lze také provádět virtualizaci

Virtualizace

- Virtualizace hw neemuluje, ale využívá hw, na kterém sama běží
  - Je proto výkonější než emulace
  - Ale také je limitovaná na programy, které byly zkompilované pro daný hw

Hypervizor

- Též známý jako Virtual Machine Monitor (VMM)
- (Hostitel) Vytváří a spouští virtuální stroje (hosty)
- Typ 1 – běží přímo na hw
  - Xen, VMWare ESX, Hyper-V
- Typ 2 – sám je hostován v OS
  - Virtual Box, WMVare Player

Mainframe

- Před érou PC dominovaly mainframy, které měly (na tehdejší dobu 50tých až 70tých let) velký výpočetní výkon, redundantní hw, I/O pro datově náročné aplikace..
- Jenomže nebyly vzájemně kompatibilní – jak tedy spustit software pro jeden mainframe na jiném?
  - IBM System/360 oddělila (virtualizovatelnou) architekturu od implementace
  - A tak bylo možné na jednom mainfraimu provozovat několik OS zároveň… až dnešních IBM zSeries, které se stále používají

PC

- PC je sice mnohem méně výkonné a spolihlivé než mainframe, ale zato je také daleko levnější
- Poměr cena/výkon dostala PC na „every desktop“
- Z PCs se dá navíc postavit cluster, tj. distribuovaný systém
  - V porovnání s mainframem stále levně
  - Software umí zajistit spolehlivost systému
  - A pomocí virtualizace pak lze efektivně vytvářet a spouštět virtuální stroje v takovém clusteru (dnes se říká cloudu)

V86 – motivace (1)

- Aneb jako to začalo na x86… viz první přednáška o MS-DOSu
- Začal se využívat protected-mode kvůli většímu adresnímu rozsahu a izolaci procesů, jenomže…
  - Některé programy nebyly přepsány do protected-mode, ale bylo potřeba je i nadále spouštět
  - Také bylo nutné ovládat zařízení,počítač nemá jenom jeden BIOS
    - Deska má svůj, grafická karta také, i síťová, SCSI, atd…
    - Protože BIOS inicializuje zařízení a x86 startuje v reálném režimu, BIOS obsahoval programy zkompilované jen pro reálný režim

V86 – motivace (2)

- BIOS neobsahuje jenom rutiny pro inicializaci hw, ale i rutiny pro jeho ovládání
  - Např. přepnutí video stránky SVGA podle VESA – viz první přednáška
  - A právě toho hojně využívaly ovladače – bylo pohodlnější zavolat již implementovanou funkci, než si ji napsat
- Protože se ale protected-mode od real-mode zásadně liší adresováním, nelze spustit v protected-mode program pro real-mode
  - =>x86 se musí přepnout do virtuálního režimu V86

V86 – popis

- Vznikl s 80386 po zkušenostech s implementací protected-mode u 80286
- Je to hw virtualizace 8086
  - Používá segmentaci jako real-mode, tj. 20-bitové adresy, ale ty už podléhají mechanismu stránkování v protected-mode
  - Pentium ještě přidalo pár vylepšení, Virtual 8086 Mode Enhancements – redukce režie s obsluhou přerušení
  - Na 64-bitových procesorech je dostupný už jenom v legacy-mode
  - Long-mode dokáže spustit program pro 8086 pomocí VT-x

V86 – princip

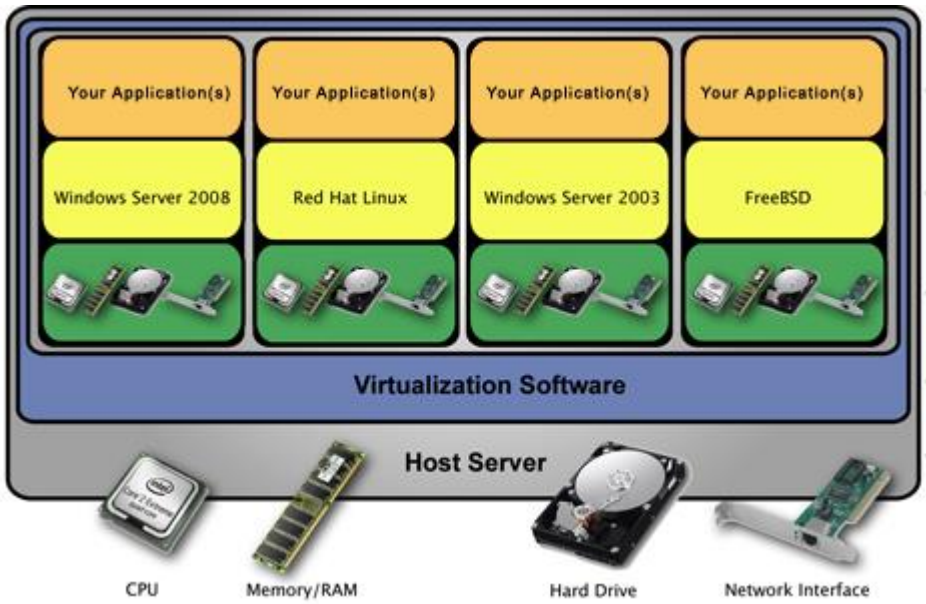
- Pokud bychom přepnuli procesor z protected-mode do real-mode, přijdeme o paměť a celý běžící OS by spadnul
- Proto se
  - Vytvoří 1MB velký paměťový prostor pro real-mode program, který poběží s CPL=3
  - Vytvoří se monitor, protected-mode task, s CPL=0
  - V okamžiku, kdy se real-mode program pokusí o privilegovanou operaci, řízení dostane monitor – CPU generuje vyjímku, ISR patří jádru
  - V okamžiku, kdy real-mode program volá službu OS, jádro převezme řízení – real-mode program generuje instrukci int, ISR patří jádru
  - Tj. v každém případě má jádro OS možnost vykonat privilegovanou operaci jak potřebuje, a real-mode program nic nepozná

V86 – důsledky

- Program běží rychle, protože jeho instrukce vykonává přímo CPU – tj. není emulován
- Není ale zcela pravda, že by real-mode program nemusel poznat, že je virtualizován
  - Některé instrukce to mohou prozradit, není-li použito VT-x nebo speciální technika, která takové instrukce z programu vyřadí před tím, než je spuštěn
  - Starý program to ale vědět nebude, protože v době jeho vzniku tyto instrukce neexistovaly

Problém dvou OS

- Jak zajistit, že jeden hostovaný OS nepoškodí jiný host. OS?
- Goldberg, „Formal Requirements for Virtualizable Third Generation Architectures“, 1974
  - Privilegované instrukce – CPU generuje vyjímku, jakmile se pokusí o vykonat instrukci, která neodpovídá CPL hostovaného OS
  - Sensitive instr. – mění hw konfiguraci a jejich výsledek závisí na aktuální hw konfiguraci



http://www.datahive.ca/data_centre_virtualization.html

Privilegované vs. sensitive

- Efektivně a bezpečně lze virtualizovat pouze tehdy, jsou-li sensitive instrukce podmnožinou privilegovaných instrukcí
  - Do příchodu Intel VT-x a AMD-V toto nebylo na x86 splněno
    - Např. SMSW byla sensitivní, ale ne privilegovaná
    - Ideově viz IBM VM/370 OS pro mainframe
  - Non-sensitive instrukce jsou vykonávány přímo CPU – jejich virtualizace má zanedbatelnou režii
  - Sensitive-instrukce – pokus o jejich vykonání generuje vyjímku, která se musí obsloužit – tj. zde dochází k emulaci v rámci virtualizace, a to je pomalé

Paravirtualizace

- Nemáme-li k dispozici obdobu VT-x, jedním z možných řešení je modifikovat hostovaný OS tak, aby nepoužíval instrukce které jsou sensitive, ale ne privilegované
  - Hostovaný OS si je vědom, že mezi ním a hw běží ještě tzv. hypervizor
  - Dostaneme výkonnostní potenciál virtualizace, ale…
  - Co se stane, když se nám do OS dostane a spustí program, který bude tyto zakázané instrukce obsahovat?
    - Bezpečnostní problém?

Binární překlad

- Aneb na čem byl založený business-plan VMware, který Intel VT-X a AMD-V zničily
- Než je hostovaný program spuštěný, je analyzovaný a všechny sensitivní, ale neprivilegované instrukce se nahradí sekvencemi instrukcí, které dělají to samé, ale bez nežádoucích vedlejších efektů
  - Dále je možné nahradit i ty sekvence instrukcí, které jinak vedou k emulaci – tj. když privilegovaná instrukce generuje vyjímku
  - Je to netriviální záležitost, protože nahrazovaná a nahrazující sekvence instrukcí nemusí mít stejnou velikost a v nahrazované sekvenci může být i cíl skoku

Binární překlad

- Aneb na čem byl založený business-plan VMware, který Intel VT-X a AMD-V zničily
- Než je hostovaný program spuštěný, je analyzovaný a všechny sensitivní, ale neprivilegované instrukce se nahradí sekvencemi instrukcí, které dělají to samé, ale bez nežádoucích vedlejších efektů
  - Dále je možné nahradit i ty sekvence instrukcí, které jinak vedou k emulaci – tj. když privilegovaná instrukce generuje vyjímku
  - Je to netriviální záležitost, protože nahrazovaná a nahrazující sekvence instrukcí nemusí mít stejnou velikost a v nahrazované sekvenci může být i cíl skoku

Binární překlad – int 10h

- Mějme program pro real-mode, který se snaží změnit mód obrazovky do textového režimu CGA 80x25x16/8

  mov ax, 3

  int 10h

- Tento kód vyžaduje emulaci grafické karty, takže bychom ho mohli rovnou nahradit sekvencí, která volá rovnou náš emulátor gr. karty, aniž bychom museli nejdříve přepínat kontext

  mov ax, 3

  pushf

  call EmulatedISR10h

Binární překlad – skok

- Mějme následující smyčku s privilegovanou instrukcí HLT, která nechá jádro CPU zastavené, dokud není signalizováno jinak

```
db flag
  ….
@skok:
    HLT                                   ;opcode HLT je 0xF4 – tj. 1B
    test byte ptr [flag], 0
    jz @skok:
```

- HLT vyžaduje CPL=0, jenomže, když se jí pokusíme nahradit pomocí call, máme problém – instrukce call má sice opcode také veliký 1B, jenomže následuje alespoň jeden další byte cíle skoku
  - Nahrazení HLT s call vyžaduje úpravu parametrů instrukcí, které adresují paměť – jmp, jnz, call, test, mov, inc, dec…. Toto není triviální

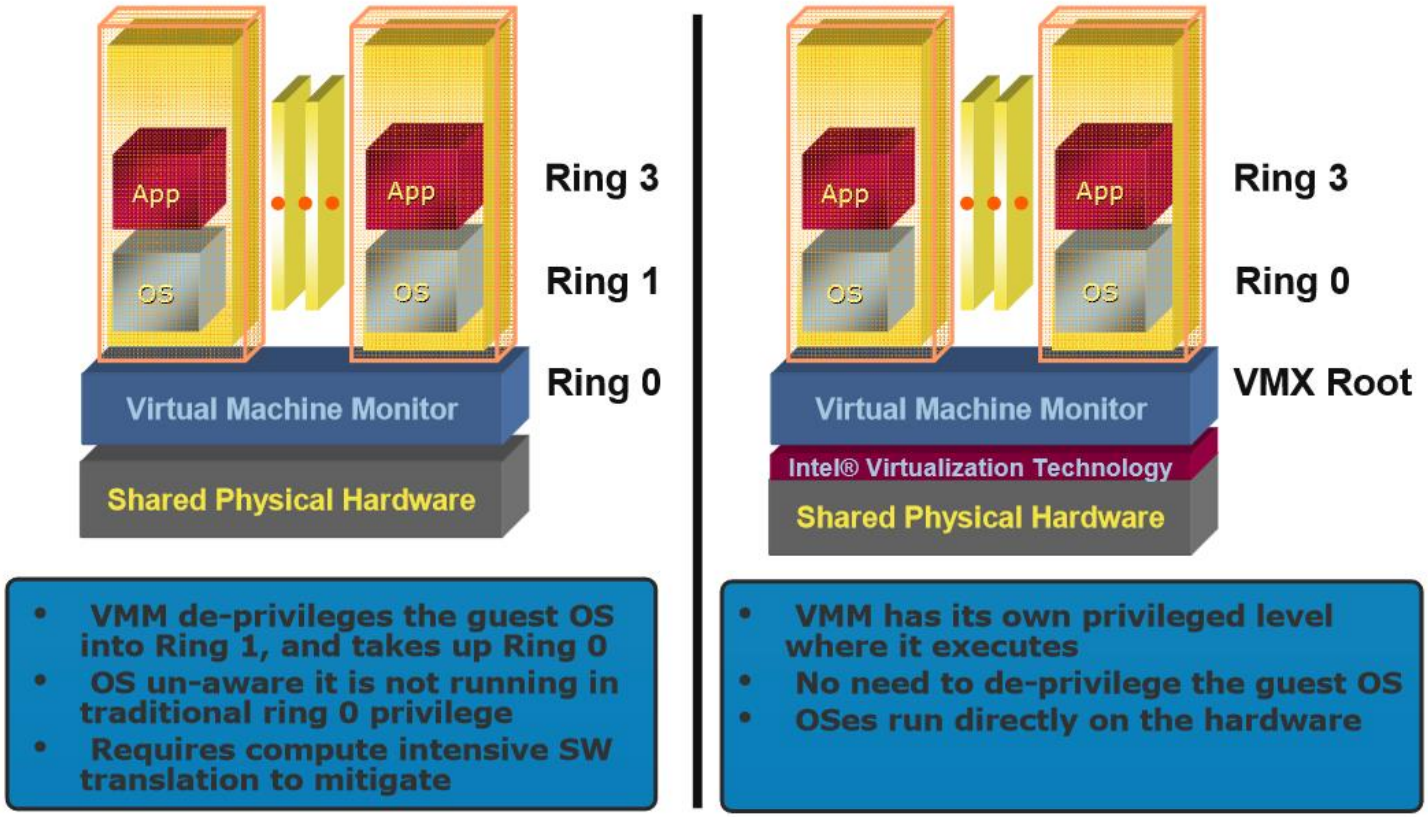
Binární překlad – pasti

- Mějme program, který obsahuje ochranu proti zpětnému inženýrství, pro jednoduchost uvažujme následující 3 byty
  - db 0ebh, 0ffh, 0c0h
  - Jedná se o instrukce jmp -1 a inc eax, které sdílejí byte 0ffh
    - 0ffh je totiž -1 jako relativní adresa skoku a zároveň je to opcode instrukce inc
    - Trik je v tom, že se disassembler po jmp -1 nevrátí o 1 byte zpět, a proto pro něj bude mít další instrukce opcode 0c0h a diassemblovaný kód pak po těchto třech bytech nebude dávat smysl
    - Anebo by se disassembler mohl vrátit o 1B zpět, ale to už vyžaduje heuristiku simulující chování procesoru

VT-x

- Cílem je eliminovat potřebu paravirtualizace a binárního překladu
- Procesor běží ve dvou režimech, takže odpadá potřeba měnit CPL – nicméně je třeba minimalizovat přechody mezi nimi
  - VMX root; přechod VM Entry
  - VMX non-rooot – hostovaný OS; přechod VM Exit
- Hostovaný OS bez jakékoliv úpravy běží ve VMX non-root režimu. a ani z žádného stavového bitu to nepozná. Jakmile se pokusí o operaci, kterou nemá dovolenu provést, dojde k tzv. VMX-transition. Řízení přebere hypervizor, který provede, co je třeba, ve VMX-root režimu procesoru.

Pre & Post Intel VT-x



- VMM de-privileges the guest OS into Ring 1, and takes up Ring 0
- OS un-aware it is not running in traditional ring 0 privilege
- Requires compute intensive SW translation to mitigate

- VMM has its own privileged level where it executes
- No need to de-privilege the guest OS
- OSes run directly on the hardware

https://software.intel.com/sites/default/files/m/0/2/1/b/b/1024-Virtualization.pdf

TLB

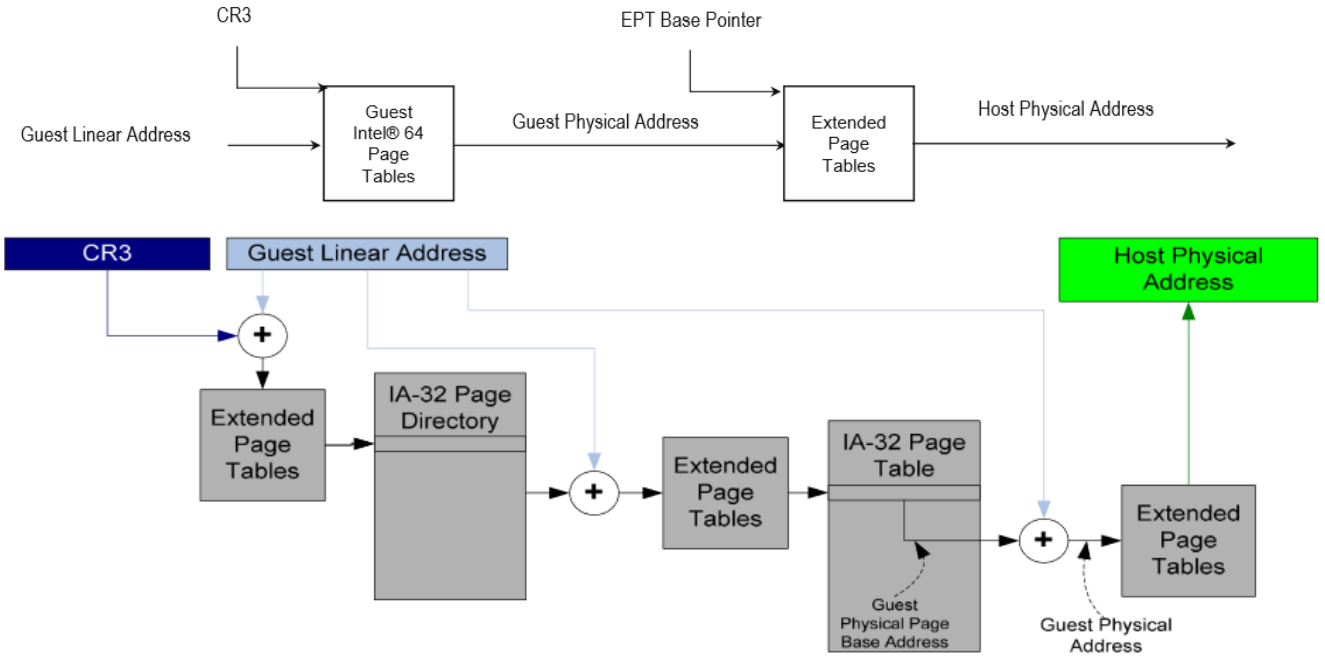
- Pokud by se při každém VMX transition měla v rámci bezpečnosti vyprázdnit TLB, mělo by to vliv na výkonost
  - Translation look-aside buffer – viz stránkování, druhá přednáška
  - A první generace VT-x to i dělala
  - V rámci vylepšení má každý VMX non-root host VPID – Virtual Processor ID
  - Položky v TLB mají VPID, takž se ví, komu patří a není nutné TLB vyprázdnit

Stránkování

- Shadow page-table jsou tabulky stránek hostujícího OS
- Tabulka stránek hostujícího OS je bez VT-x read-only, což umožní zachytit pokus o její modifikaci a následně synchronizovat shadow verzi
  - Jenomže pokus o zápis by generoval vyjímku, a to je pomalé
- VT-x má proto koncept zanořených/rozšířených tabulek stránek, který toto eliminuje

Stránkování

- Host používá tabulku stránek, jak byl zvyklý, ale fyzická adresa hosta se ještě přes rozšířenou tabulku, tj. zanoření, převede na fyzickou adresu hostitele

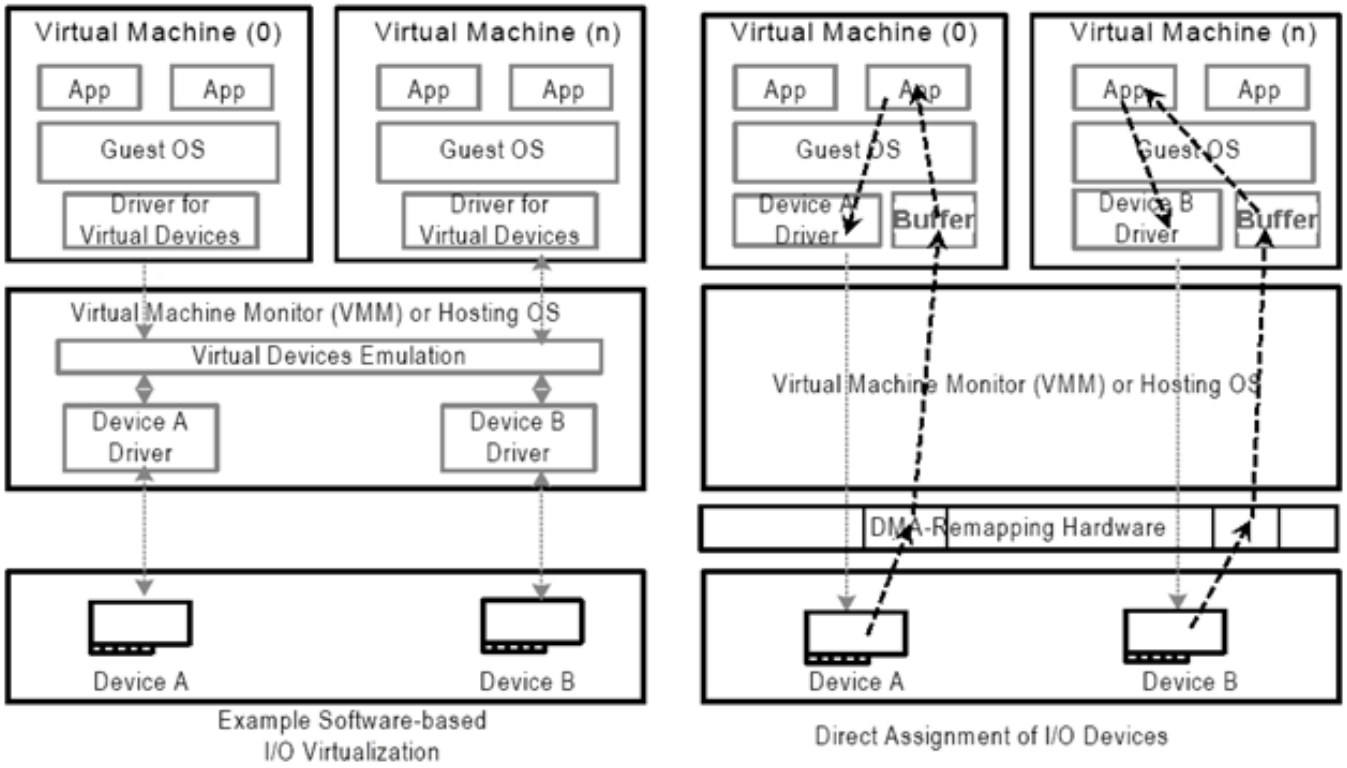


https://software.intel.com/sites/default/files/m/0/2/1/b/b/1024-Virtualization.pdf

VT-d/VT for Directed I/O

- K výkonnostním penaltám dochází ještě při přístupu k I/O zařízení – opět díky VMX-Transition do VMX-root
- VT-d umožňuje přemapování IRQ a definuje další verzi architektury DMA
  - I/O zařízení lze přiřadit přímo konkrétnímu hostu
  - Vyžaduje extended xAPIC

VT-d/VT for Directed I/O



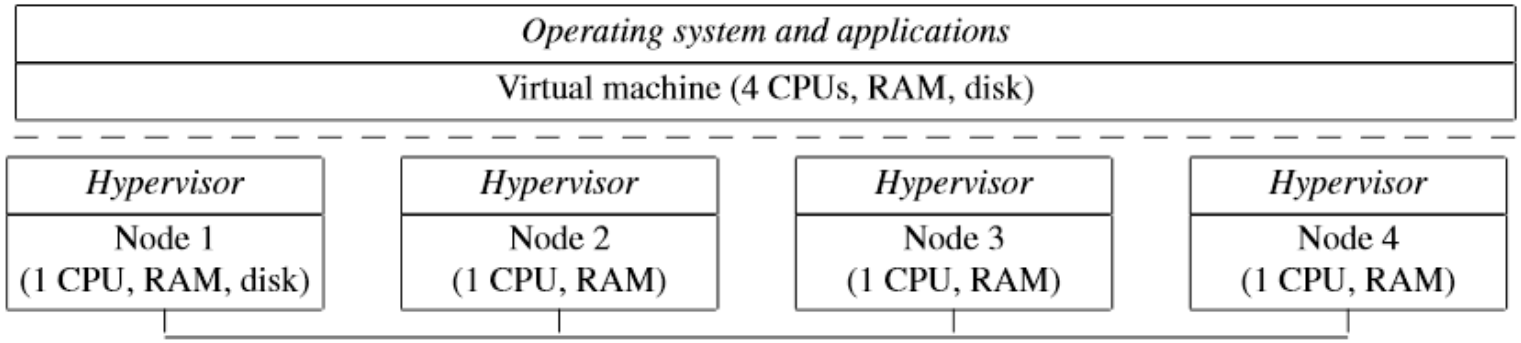
https://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices

Virtualization for Aggregation

- Nesnažíme se virtualizovat jeden počítač pro několik hostů, ale naopak se snažíme virtualizovat několik počítačů pro jednoho hosta
- Levně můžeme postavit počítač s enormním množstvím RAM a procesorových jader z běžně dostupných komponent
- Hostovaný OS uvidí jenom (virtuální) SMP
- S vSMP padají náklady na údržbu clusterů, na portování a vývoj programů pro distribuované prostředí

VfA – jak?

- Na každém počítači VfA vSMP běží VMM, který používá např. VT-x a komunikuje s ostatními VMM
- Když zachytí přístup k něčemu, co se nachází na počítači s jiným VMM, zasíláním zpráv „přesměruje“ zachycený požadavek jinému VMM, který ho vyřídí na svém HW



https://www.usenix.org/legacy/event/usenix09/tech/full_papers/chapman/chapman.pdf

Rootkit

- Máme-li k dispozici tak perfektní virtualizaci, jak složité by s ní bylo vytvořit rootkit?

1. Inicializace VT-x
2. Vytvoření VM a VMM
3. Zkopírování hostitelského OS do VM
4. Předání řízení do VM
5. Ukončení činnosti v hostitelském OS, nyní již běžícím jako hostu
6. Při VMX Transition do VMX root se aktivuje rootkit a ví vše, co se děje v hostu

Detekce rootkitu

- Sice není k dispozici oficiálně dokumentovaný stavový bit, který by host mohl použít, ale…
- Např. CPUID vždy způsobí VM Exit
  - Tj. má podstatně větší latenci, když je VT-x aktivní!