SSZ 2016 SWI

Systémové programování

SP

Část: Operační systémy

Seznam otázek

1. Zavedení a implementace jádra operačního systému a procesu uživatelského režimu.
2. Implementace virtuálního adresového prostoru, sdílené paměti, sdílených knihoven a copy-on-write.
3. Obsluha a implementace přerušení, systémových volání, vyjímek a V/V zařízení.
    a. přednáška 3 (možná i 4) + 9
4. Implementace vláken na symetrickém multiprocesoru a jejich synchronizace.
    a. přednáška 5 + 6
5. PCB, TCB, stavová fronta, plánovač.
6. Implementace meziprocesové synchronizace – semafor, mutex, roura, zprávy, signály.
    a. přednáška 7
7. Virtual File System, Installable File System, FAT, Ext2, NTFS.
    a. přednáška 8
8. Emulace, paravirtualizace, binární překlad, VT-x, VfA.
    a. přednáška 10

1. Zavedení a implementace jádra operačního systému a procesu uživatelského režimu.

Zavedení operačního systému

BIOS

- program přítomný ve vestavěné paměti HW (většinou na základní desce)
- provádí testy a nastavení HW
- vybere zaváděcí jednotku
- načte první sektor (MBR), kde je umístěn program zavaděče a provede skok na adresu jeho programu, čímžmu předá řízení

Zavaděč(bootstrap program)

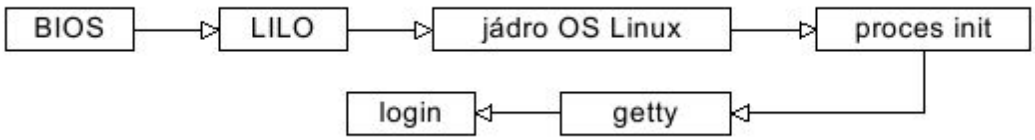
- pro Linux LILO (LInux LOader) je všeobecněpoužitelnýzavaděč(boot loader) proLinux) nebo GRUB
- dává možnost zvolit startující OS
- načte jádro operačního systému do paměti a spustí ho

Jádro OS

- detekuje hardware a odpovídajícím způsobem nastaví ovladače zařízení
- připojí kořenový svazek pro čtení a provede kontrolu souborového systému
- spustí na pozadí proces init

Proces init

- proces init se konfiguruje pomocí souboru /etc/inittab
- inicializuje operační systém
- spuštěn po celou dobu běhu operačního systému a ošetřuje některé události (úklid v adresáři /tmp)
- spustí služby -Démony
- nakonec spustí program getty pro terminály a virtuální konzoli a v ní program login
- nastavením parametru jádra tzn. runlevel lze upravit chování systému



Úrovně běhu systému -runlevel

- runlevel 0 –zastavenísystému -halt
- runlevel 1 –jednouživatelský režim Single user mode
- runlevel 2 –víceuživatelský režim bez podpory sítě Multiuser, without NFS
- runlevel 3 –víceuživatelský režim s podporou sítě Full multiuser mode
- runlevel 4 –nenípoužit unused
- runlevel 5 –víceuživatelský režim s podporou sítě a XFree X11
- runlevel 6 –restart systému reboot

Zavádění systému

- nejprve proběhne úspěšný test zavádění systému –POST
  - kontroluje HW v zařízení
  - série testůke zjištění, zda HW pracuje správně
  - zjištěné chyby jsou uloženy nebo oznámeny –blikáním LED/série pípnutí
  - Po dokončení je řízení předáno bootovací sekvence volající ovládací SW nebo zavaděč OS
- Bootování
  - Najde a zavede (vygeneruje se přerušení 19h) se tzv. Bootovací sector (boot sector)
  - Boot sector -oblast 512 bajtůna záznamovém médiu, které je jako první nastavené v paměti BIOSu
    - Bootsektor se nachází na prvním sektoru záznamového média (v případěpevných diskůje to válec 0 hlava 0 stopa 0 sektor 1)
  - BIOS se snaží najít na tomto sektoruMaster Boot Record(MBR) –hlavní spouštěcí záznam.
    - Ten nahraje do paměti na adresu 0000:7C00 a v případěúspěchu mu předá řízení.
    - V případěchybného MBR se bootovací proces přeruší pomocí softwarového přerušení 18h –vygeneruje chybové hlášení (např. NO ROM BASIC –SYSTEM HALTED)
    - Správnost MBR BIOS zjišťuje pomocí kontrolní hodnoty umístěné na posledních dvou bajtech sektoru -AA55h(zápis je uložen ve formátu little endian)
    - MBR –ze dvou částí –partition loadera partition tabulky
    - MBR uchovává záznamy o rozdělení disku (oddílech) a určuje, ze kterého z nich se má bootovat.
    - Pokud MBR OK –řízení se předá partition loaderu
    - Partition loaderv Partition tabulce vyhledá oddíl, který je označen jako aktivní a přejde na první sektor tohoto oddílu. MBR sám sebe překopíruje na jiné místo v paměti a nasvé původní místo zkopíruje tento první sektor a předá mu řízení

Spuštění počítače

- Po zapnutí PC jsou všechny procesory v reálném režimu
- Náhodněse vybere jedno jádro -> bootstrap processor (BSP)
  - ostatní CPU jsou nyní application processors AP -pozastavené, dokud je nezapne kernel
- nyní je BSP v tzv. real mode, vypnuté stránkování (BSP simuluje staré 8086 ze let okolo '78)
  - v tomto stavu je adresováno pouze 1MB paměti bez ochrany (lze v ní spustit cokoliv)
  - u Intel CPU je hack, kdy se nastaví bázová adresa (jako offset) na tzv. reset vector –0xFFFFFFF0 (konec 4GB paměti -16B)
  - na adrese reset vectoru je jump na adresu, kde je namapovaný BIOS entry point (zajišťuje základní deska)
  - tento skok vymaže Intelí hack bázovou adresu
  - oblasti v paměti jsou zaplněna správnými daty díky memory map v chipsetu
- nyní BSP spustí BIOS -> Power-On self test (POST) -> error = pípání PC speakeru nebo zombie PC
  - po POST bootování systému, umístění volitelné (disketa, DVD ROM, HDD, ...)
- BIOS nyní přečte první sektor umístění (HDD) o velikosti 512B (zero sector) = Master Boot Record (MBR)
- MBR obsahuje:
  - malý zavaděčna začátku MBR specifický pro OS
  - tabulka partition
- obsah MBR je načten do adresy 0x7c00 a skočí na začátek kódu v MBR (zavaděč)
- bootujeme

...nebo (přednáška d_multithreading.pdf)...

- Po zapnutí jsou všechny procesory v reálném režimu

- BIOS vybere BSP a ostatní procesory zastaví
- Kód SMP jádra běžící na BSP prohledá paměťna_MP_
- Pokud nenašel, zavede se jednoprocesorové jádro
- Pokud našel, inicializuje APIC BSP
    - K tomu je nutné se přepnout do chráněného režimu
- Kód vykonávaný BSP postupněvzbudí AP pomocí Init-IPI (Inter-Processor Interrupt)
- AP se přepne do chráněného režimu a začne svojidalší činnost synchronizovat s kódem, který ho spustil a běží na BSP
- Jakmile jsou inicializovány všechny AP, BSPpřepne I/O APIC do symetrického IO režimu
    - Routovací tabulka, která přesměruje přerušení od sběrnic periferií na některý lokální APIC AP
- SMP jádro pokračuje dál s vlastní inicializací

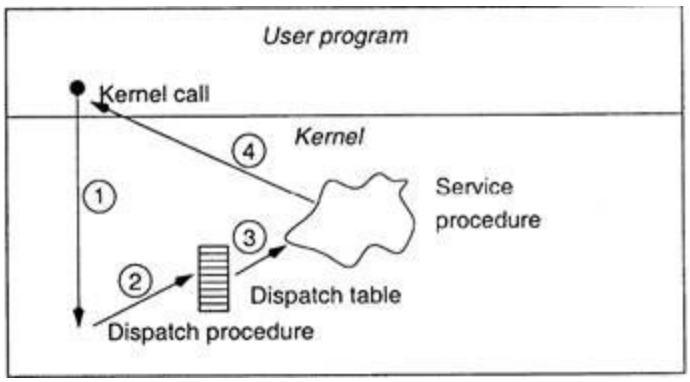
Struktura OS

Monolitické systémy

- pro jednotlivé funkce jsou definovány moduly
- modul může volat jakýkoli jiný modul
- všechny moduly jsou spojeny do vykonatelného souboru s operačním systémem

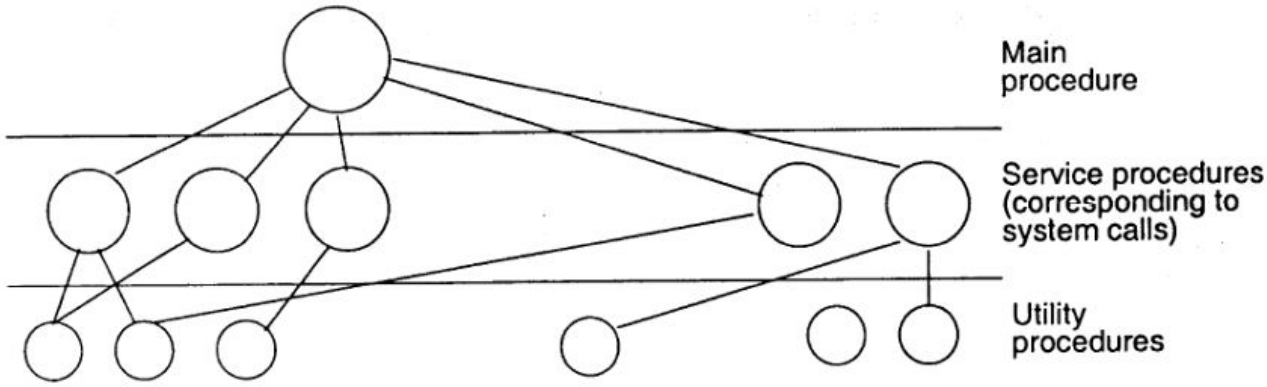
Systémové volání (služba jádra)

- volání vstupního bodu jádra OS spřepnutím do privilegovaného režimu
- zjištění čísla požadované služby
- volání obslužné procedury
- návrat spřepnutím do neprivilegovaného režimu



Model struktury monolitického systému

- hlavní program, který spouští obslužnou proceduru
- množina obslužných procedur pro systémová volání
- podpůrné procedury pro vykonání obslužných procedur
    - mají tendenci extrémněnarůstat
    - Monolit akumuluje moduly, které by potenciálněmohli být potřebné
    - Těžce se ladí



Systémy založené na mikrojádře

Vrstvené systémy

- Hierarchie vrstev poskytujících služby
- Programy vyšší vrstvy využívají služeb nižších vrstev
- Holý počítačje nejnižší vrstva
- Aplikační program je nejvyšší vrstva
- Princip vrstev umožňuje systematickou tvorbu programůa jejich testování
- Princip vrstev je možné použít pro monolitický model i pro systémy

Mikrojádro

- Vrstva nad holým strojem, která obsahuje minimální množinu abstrakcí, tak aby ostatní funkce OS mohly být implementovány nad ním
- Tyto funkce OS nemusí být vykonávány vprivilegovaném režimu
- Jenom mikrojádro musí být vykonáváno vprivilegovaném režimu
    - Přerušení
    - Vlákna
    - Správa paměti
    - Meziprocesová komunikace
    - Procesy
    - Typická množina abstrakcí implementována mikrojádrem:
- Ostatní funkce –soubory, adresáře, síťové služby –jsou programy vykonávané vuživatelském režimu

Základní rozdělení

- monolitcké systémy - hlavní program, obslužné procedury, podpůrné procedury
- vrstvené systémy - hierarchie vrstev, nejnižší je holý počítač, nejvyšší je aplikační program

Funkční hierarchie

Někdy je problém rozdělit do vrstev podle úrovněabstrakce, proto dělení do vrstev podle funkčnosti:

- klient-server
    - obsahuje mikrojádro, které poskytuje pouze základní funkce
    - většinu práce dělají servery, které jsou oddělené od jádra
- objektově orientovaná struktura
    - jádro spravuje řadu objektů(zastupují soubory, HW zařízení, ...)
    - mezi objekty jsou tzv. capability = odkaz na objekt + množina práv definujících operace

2. Implementace virtuálního adresového prostoru, sdílené paměti, sdílených knihoven a copy-on-write.

Status: potřebuje revizi, mírně se liší od původní otázky (část otázky je nová)

Virtuální adresný prostor

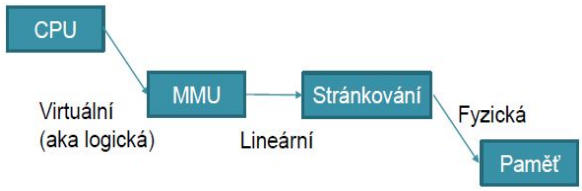
Chceme, aby…

…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 MemoryManagement Unit (MMU)
  - Detaily převodu zadá MMU přímo OS

- 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



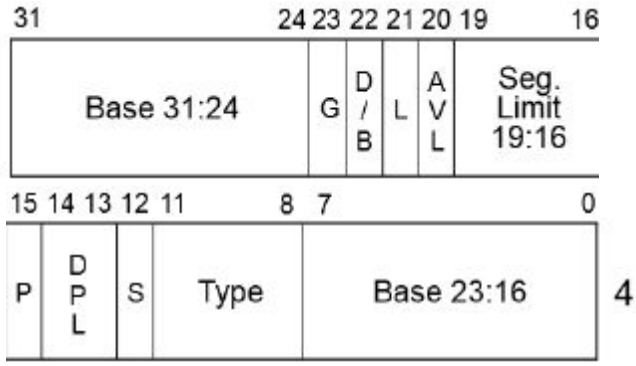
Protected mode

- 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

Segment Descriptor

Segment má:

- Základní (base) adresu (lineární adresa)
- Velikost (limit)
- Typ (kód, data, atd.)
- Přístupová práva + další vlajky



Izolace jádra

- Pro izolaci je důležitá hodnota bitů **CPL/DPL**
  - **00** – většinou jádro
  - **01** – může být jádro, když na 00 poběží hypervizorvirtualizace
  - **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 => izolacejádraod uživ. procesů

Izolace procesů

- Před vlastní inicializací protectedmode musí jádro OS vytvořit tabulku deskriptorů segmentů
- **Globální tabulka** - používaná jádrem
  - uložena v registru gdtrprivilegovanou instrukcí lgdt
- **Lokální tabulka** - používaná konkrétním jedním procesem
  - Uložena v registru ldtrprivilegovanou instrukcí lldt
  - Tj. při přepnutí kontextu může lldtvykonat pouze jádro s CPL=0 a tudíž si uživatelský proces nemůže měnit jemu přidělenou paměť

Výhody segmentace

- Izoluje procesy a jádro
- Lze sdílet segmenty –např. read-onlyprogramový 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 => jakje poskládatdo 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 protectedmode
  - 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)
- 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), Dirtya Accessed
- Běžící proces často potřebuje jenom omezené množství stránek
  - Jak tedy zrychlit převod virtuální adresy na fyzickou?
  - Pomocí asociativní pamětí Translationlook-asidebuffer(TLB)
    - Je rychlá, ale také je malá
      - Větší hit rate když je větší, ale také je pak pomalejší
    - Má zásadní vliv na výkon

Sdílená paměť a kód (knihovny)

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 pageentry, která ukazuje na ten samý rámec (fyzická stránka) jako pageentriesv tabulkách stránek ostatních procesů, které tímto „trikem“ sdílejí tu samou paměť

Sdílený kód

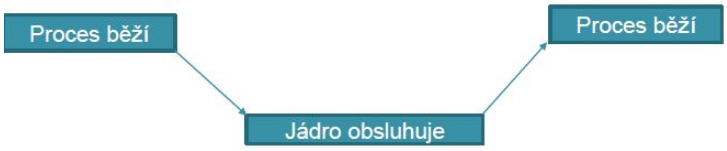
- Pokud několik procesů používá sdílený programový kód, proč ho do paměti nahrávat vícekrát?
- Příslušné stránky s kódem se označí jako read-onlya namapujíse do paměťových prostorů příslušných procesů ⇒ tj. **stále jde o sdílenou paměť**
- Příkladem jsou dynamické knihovny
    - dll, so, …

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-onlystránky, procesor vygeneruje vyjímku, kterou zachytí OS
3. OS pak alokuje novou stránku, zkopíruje do ní původní read-onlystránku, a aktualizuje tabulku stránek procesu
4. Po návratu z obsluhy proces normálně zapíše data už do nové stránky, aniž by o něčem vůbec věděl

3. Obsluha a implementace přerušení, systémových volání, vyjímek a V/V zařízení.



Přerušení pro systémová volání

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
  - 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. axna určení konkrétní služby
  - Služba MS-DOS se zavolá pomocí přerušení 21h
    - adr. hlavní rutiny služeb OS je zapsána na 0x21. pozici tabulky vektorů přerušení
  - **Obsluha**
    1. Program nastaví příslušné registry vykoná int21h
    2. Do zásobníku, na jehož vrchol ukazuje SS:SP, se uloží registry CS:IP (ukazující ve volajícím programu na další instrukci po int21h) a registr Flags
       - Provede procesor v rámci zpracování instrukce int21h
    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
- Při použití intse 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 realmode, ale její pozici udává registr IDTR
- Procesor nastaví CS:EIP na získaný vektor přerušení
  - V zásobníku jsou uloženy registry přerušeného vlákna flags, csa eip; cs:eipukazují 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 -main

- Obsluha přerušení vykoná pouze nezbytnou část požadované činnosti
  - Lze-li něco odložit na později, odloží se to –viz BottomHalfdále

    - 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 flagstohoto vlákna
    - Pokud jde o jiné vlákno, musí se obnovit i zbývající registry
- Obsluha přerušení udělá instrukci iret, která nastaví registry CS:EIP a flagsz aktuálního zásobníku SS:ESP
    - A z CS:EIP procesor určí svůj režim -CPL

Odložení kontextu

- **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 syscalljako rychlejší varianty int
- Deprecatedna x86-64

Syscall

- intgenerovaný programem je synchronní událost vůči běhu programu ⇒ ze toho využít
- ***syscall/sysret*** jsou speciální instr. → 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
- Kdy lze použít int, sysenter/sysexitči syscall/sysretzávisí na dostupném hw a jádru OS
    - V každém případě se ale jedná o netriviální činnost, které by mělo být volající vlákno ušetřeno
- OS namapujedo 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
- ***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 dwordptr[adresa]
    - Nebo se použijí registry ***fs*** (Win) či ***gs*** (Linux), přičemž segmentový registr odkazuje na ThreadControlBlock (TBC)
        - call fs:[offset od začátku TCB]

Přerušení pro vyjímky

- Vyjímkaje **přerušení generované procesorem**, když dojde k
    - **Trap** – např. breakpoint
    - **Fault** – opravitelná chyba, program může pokračovat
        - Např. PageFault
    - **Abort** – neopravitelná chyba
        - Např. Double Fault – dojde k PageFault, 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, ipa flags

Exceptionhandler

- 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*
- Pokud by OS neměl handlervyjímek, došlo by k restartu počítače
  - OS ho má –tj. dojde-li k vyjímcev uživatelském procesu, OS vyjímkuzachytí
- 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 ***ThreadControlBlock***

Přerušení pro V/V (I/O) zařízení

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

Master PIC

- **IRQ 0** — systémový časovač.
- **IRQ 1** — přerušení klávesnice.
- IRQ 2 — předává signály z IRQs 8–15
  - *každé zařízení konfigurované tak, aby používalo **IRQ 2 ve skutečnosti** používá přeruš. **IRQ 9***
- IRQ 3 — sériový port controller pro COM2.
- IRQ 4 — řadič sériového portu COM1.
- IRQ 5 — LPT port 2 nebo zvuková karta
- IRQ 6 — řadič disketové mechaniky (viz disketa)
- IRQ 7 — LPT port 1 nebo zvuková karta.

Slave PIC

- **IRQ 8** — hodiny reálného času
- IRQ 9 — open interrupt / available nebo SCSI host adapter
- IRQ 10 — open interrupt / available nebo SCSI nebo NIC
- IRQ 11 — open interrupt / available nebo SCSI nebo NIC
- **IRQ 12** — myš na PS/2
- IRQ 13 — matematický koprocessor nebo integrovaná floating point unit nebo meziprocesorové přerušení.
- IRQ 14 — primární ATA kanál
- IRQ 15 — sekundární ATA kanál
  - ATA rozhraní obvykle obsluhuje hard disky a CD-ROM mechaniky

Časovač

- Např. tik hodin je IRQ0 (nelze změnit ani maskovat) a vyvolá obsluhu přerušení int08h
  - Proběhne každých 55ms
- Na tomto přerušení závisí mnoho důležitých činností a je proto nutné, aby
  - Bylo co nejrychlejší
  - Zavolalo původní obsluhu přerušení
- Pomocí časovače se implementuje preemptivní multithreading(a následně multitasking)
- Nejprve se do proměnné ***oldVec8*** uloží adresa původní obsluhy přerušení, takže obsluha může vypadat následovně:

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

I/O Porty

- Input/Ouputbase 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 int08h je nutné poslat řadiči přerušení informaci, že přerušení již skončilo

```
moval, 20h          ;signál Konec přerušení
out20h, al          ;port řadiče přerušení 8259
```

4. Implementace vláken na symetrickém multiprocesoru a jejich synchronizace.

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 běžína dalších procesorech v systému
- Vlákno je sekvence instrukcí, která může být spravována plánovačem OS
  - Unit ofCPU scheduling
- Z pohledu programátora je to často funkce, která se vykonává asynchroněk funkci main
  - Přičemž funkce mainběží 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 → ***jak velké bude časové kvantum?***
- **Kooperativní multitasking**: 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 (TCB)

- Jádro spravuje threadpodle jeho TCB *(Win/Linux přístupný přes fs/gsregistry na x86, x86-64)*
- TCB obsahuje:
  - Uložené hodnoty registrů procesoru
  - Priorita
  - Stavovou proměnnou, čas dosavadního běhu vlákna
  - Ukazatel na ProcessControlBlock
  - A další specifické info, jako jsou např. seznam obsluh vyjímek, skok na stránku s fcí 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.

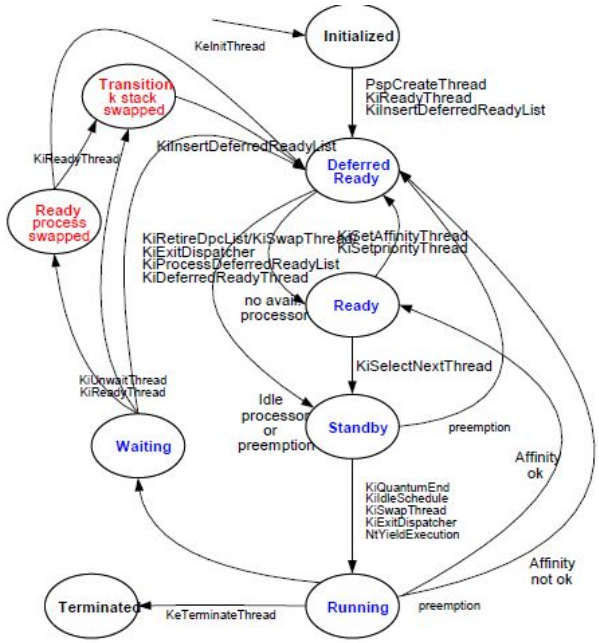
Stav vlákna

Zjednodušeně

- **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**

Skutečně

- Ve skutečnosti je to složitější, protože OS potřebuje vědět, proč vlákno čeká
- Určitě je rozdíl např. mezi ***PageFault*** a kritickou sekcí



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 responsivnostisystému

Round Robin

- Žádná priorita, jenom seznam vláken
  - Vláknům se přiděluje pouze časové kvantum 20 –50 ms
  - Když se jedno vlákno přeruší, vezme se další runnablena 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-timeprocesy
     - 100 –139 jsou normální procesy, nastavuje se pomocí nice()
     - 5 integerůtvoří bitmapu jejíž bity říkají, která fronta má spusitelnýproces
   - Process větší statickou prioritou (tj. menším číslem) dostává větší časová kvanta
   - Dynamická priorita: vypočítává se ze statické priority a doby, po kterou proces neběžel
2. Vybere se v ní první spustitelný proces a vypočítá se jeho časové kvantum
3. Spustí se a až vyčerpá své kvantum, dá se expirefronty
4. Zpět do prvního bodu

Simetrický multiprocesor (SMP)

SMP - 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
- Cachepodstatným 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. sizeofeaxči 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
- zamčení:

```
movedx, DWORD(-1)   //-1 zamčeno

//otestujeme stav zámku, 0 = odemčeno
spin: moveax, [lockState]
       test eax, eax
       jnz spin

//zkusíme ho zamknout s -1
lockcmpxchg[lockState], edx

//nepředběhl nás jiný procesor?
//původní lockStateje v eax
       test eax, eax
jnzspin
```

- odemčení:

```
movDWORD PTR [lockState], 0
```

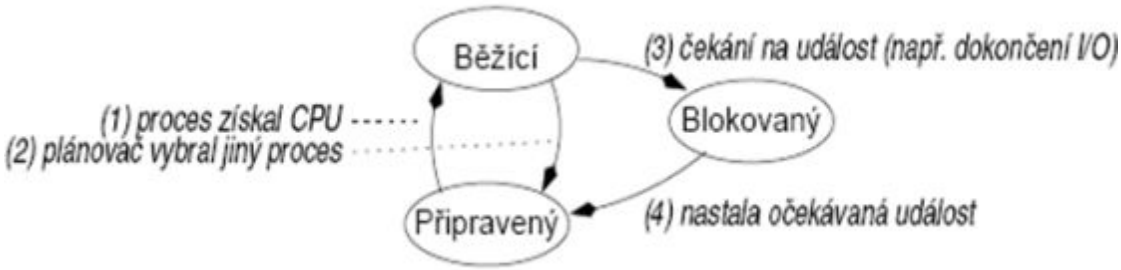
5. PCB, TCB, stavová fronta, plánovač.

PCB - Process Control Block

- OS udržuje tabulku nazývanou tabulka procesů
- Každý proces v ní má položku zvanou PCB (Process Control Block)
- PCB obsahuje všechny informace potřebné pro opětovné spuštění přerušeného procesu
  - Procesy se o CPU střídají, tj. jeho běh je přerušovaný
- Konkrétní obsah PCB – různý dle OS
- Pole správy procesů, správy paměti, správy souborů (!!)

Správa procesů

- Identifikátory (číselné)
  - Identifikátor procesu - PID
  - Identifikátor uživatele - UID
- Stavová informace procesoru
  - Univerzální registry,
  - Ukazatel na další instrukci - PC
  - Ukazatel zásobníku SP
  - Stav CPU – PSW (Program Status Word)
- Stav procesu (běžící, připraven, blokován)
- Plánovací parametry procesu (algoritmus, priorita)



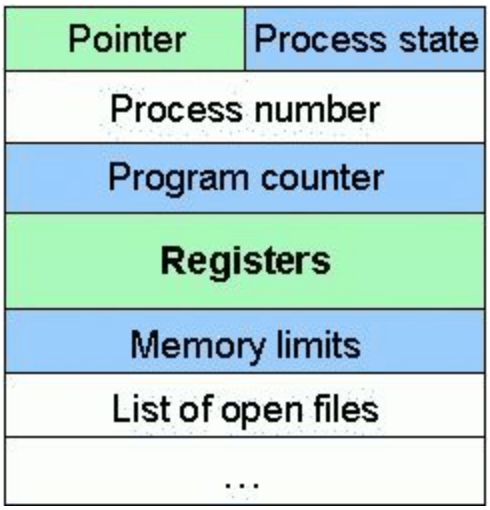
Správa paměti

- Popis paměti
  - Ukazatel, velikost, přístupová práva
- Úsek paměti s kódem programu
- Data – hromada
  - Pascal – new release
  - C – malloc, free
- Zásobník
  - Návratové adresy, parametry funkcí a procedur, lokální proměnné Pol

Správa souborů

- Nastavení prostředí
  - Aktuální pracovní adresář
- Otevřené soubory
  - Způsob otevření – čtení / zápis
  - Pozice v otevřeném souboru PC

PCB - obrázko



TCB

- info specifické pro vlákno
- stack pointer, pc - registry
- stav vlákna
- odkaz do PCB
- další info - seznam obsluhy výjimek, skok na stránku pro syscall

Stavová fronta

- kolekce seznamů, které drží stav všech procesů
- každý stav svůj seznam, včetně podstavů (např. čekající kvůli page fault)

Plánovač

Tři zásadní údaje, které charakterizují plánovač:

- rozhodovací mód
  - okamžik, kdy jsou vyhodnoceny priority procesu a vybrán proces pro běh
  - Nepremtivní
    - Proces využívá CPU, dokud se jej sám nevzdá
  - Preemtivní
- prioritní funkce
  - určí prioritu procesu v systému
  - Většinou dvě složky statická a dynamická priorita
    - Statická – přiřazena při startu procesu
    - Dynamická – dle chování procesu (dlouho čekal, aj.)
  - priorita = statická + dynamická
- rozhodovací pravidla
  - jak rozhodnout při stejné prioritě
  - malá pravděpodobnost stejné priority
    - náhodný výběr velká
  - pravděpodobnost stejné priority
    - cyklické přidělování kvanta
    - chronologický výběr (FIFO)

Obecně (od Hadyho)

- plánoví procesů/vláken, které se budou spouštět, spravedlivost, pravidla, efektivita, nízká režie
- Prioritní plánování
- loterie - priorita více lístků
- v dávkových systémech
    - **FCFS** (First Come First Served) - FIFO
    - **SJF** (Shortest Job First) - nejkratší úloha jako první
    - **SRT** (Shortest Remaining Time) - úloha, jejíž zbývající doba běhu je nejkratší

6. Implementace meziprocesové synchronizace – semafor, mutex, roura, zprávy, signály.

Status: potřebuje revizi, mírně se liší od původní otázky

Meziprocesová synchronizace

- 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(passeringv originále, „Pusť 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

Uspání threadu

- Není-li možné thread vpustit dále za semafor, OS ho musí uspat
    - Stav threadu se změní na blokovaný
    - Threadse 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 threadu spí
    - 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(vrijgavev originále, „pusť mě Ven“)
- 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

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 Acquires n>1?
    - Pak je třeba vybrat thread, který byl uspán s n menším nebo rovným počítadlu semaforu
  - Co když je thread uspán ještě z jiného důvodu?
    - Např. je-li uspán z debuggeru
    - Threadne bude 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

Mutex

- Sice má na venek 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

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

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
- Jeden thread doručí zprávu druhému threadu
  - Může i nemusí čekat, až ji příjemce zpracuje
- OS spravuje frontu příchozích zpráv per thread

Hlavní smyčka - sračka se učit, ale pro představu

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

- Každé window má svou WindowProcedure, která zprávy přijímá

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
  - Speciální zpráva, umožňující předat velké množství dat

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

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
- OS každému procesu při jeho vytvoření poskytne tzv. default handler pro každý signál
  - Dokud proces nenastaví svou obsluhu, vykonává se obsluha OS
  - Např. default handlerSIGINT ukončí proces
- 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

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

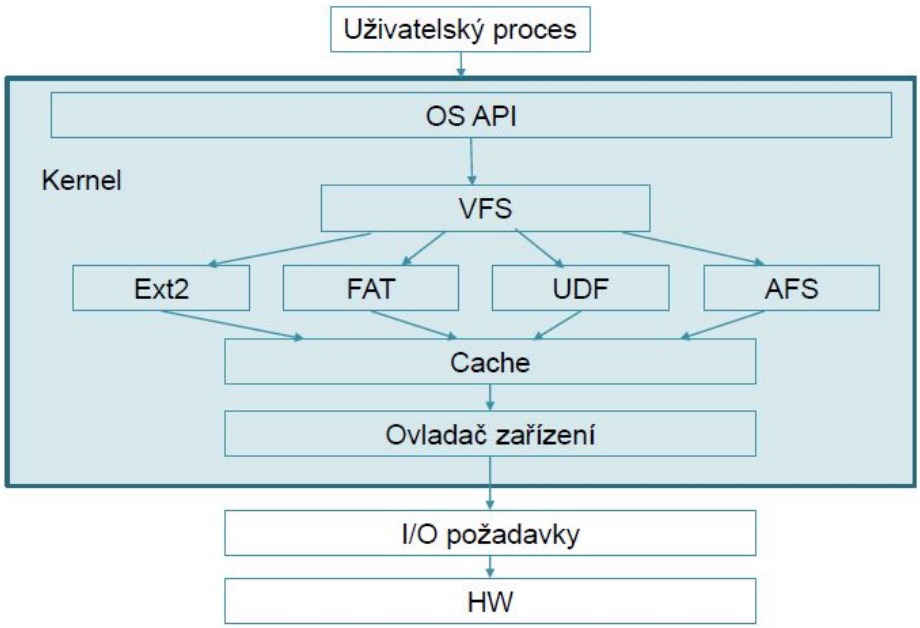
Signály vs. zprávy

- 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

7. Virtual File System, Installable File System, FAT, Ext2, NTFS.

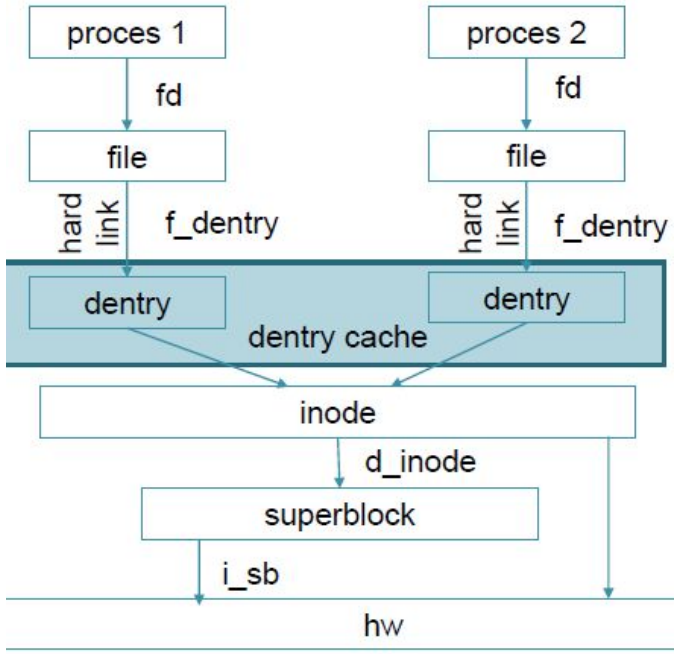
Virtual File System (VFS)

- Threadvolá 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 –VirtualFileSystem



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

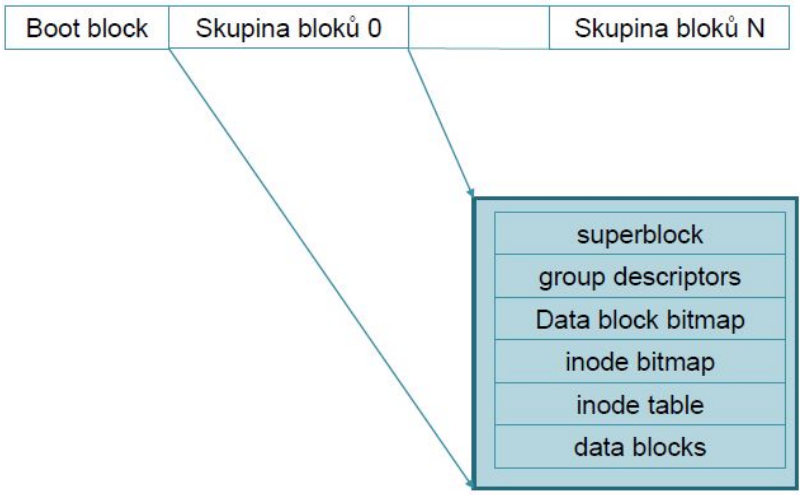


VFS tabulka funkcí

- 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

!Extended File System2

- 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
    - Prealokujebloky souborům před tím, než jsou použitý
    - První blok je vždy vyhrazen pro **bootsector**



Struktura ext2: první skupina bloků, tj. 0, je vyhrazena pro jádro OS

superblock obsahuje

- Celkový počet inodeuzlů
- Velikost souborového systému v blocích
- Počítadla volných bloků a inodeuzlů
- Velikost bloku
- Počty bloků a inodůuzlů ve skupině
- 128-bit id souborového systému
- Počítadlo připojení
- A další…

groupdescriptor (ext2_group_desc) obsahuje

- Počty bloků bitmap bloků, inodeuzlů a prvního inodev tabulce bloků
- Pokud je n-týbit bitmapy nastaven, daný blok/inodeje použit
- Počty volných bloků, inodeuzlů 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 inodeuzlů
- Symbolický odkaz – do 60 bytů se odkaz ukládá v inode(tzv. fast symboliclink), jinak také potřebuje datové bloky

Installable File System

- **VFS Windows**
- Funguje ve třech režimech
  - ◦nemusí být nutné implementovat všechny tři
  - ◦filesystem–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 RequestPacketpro komunikaci

IO Request Packet (IRP)

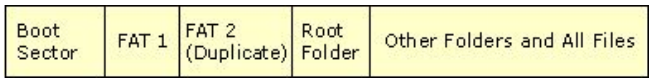
- Struktura používaná ke komunikaci mezi ovladačem zařízení a OS
- Popisuje požadavky, které se mají se provést
- Dávají se do fronty, kterou si OS může přeuspořádat
- Většinou je vytváří I/O managerpodle volání souborových funkcí z uživatelského adresového prostoru
- Ale mohou je vytvářet i další části,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

Fast I/O

- IRP je výchozí mechanismus pro všechno
  - Synchronní i asynchronní přenosy, data v cachei mimo ni, výpadky stránek
- Ale pro synchronní operace nad daty v cachelze použít Fast I/O
- Data jsou pak přenesena přímo mezi bufferyprocesů přes systémovou cache
  - Zkratka, která obejde celý fsa hw stack–něco jako loopbackna localhostu
- Fast I/O dělá v případě neúspěchu fallbackna IRP

File Allocation Table (FAT)

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

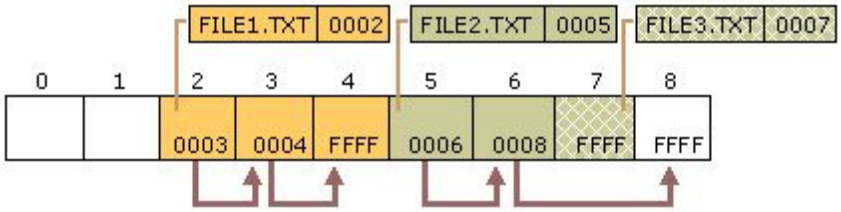


FAT - položka adresáře obsahuje

- 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

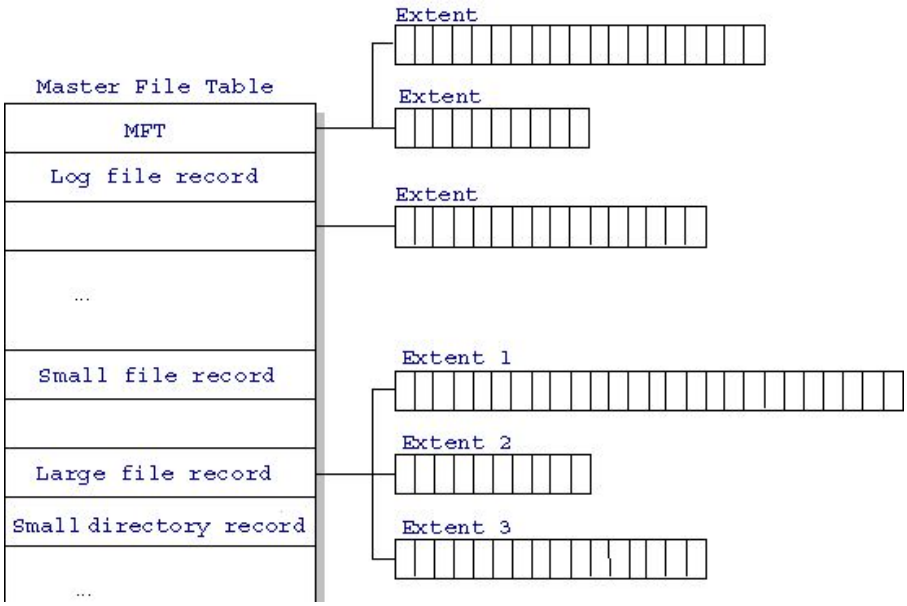


New Technology File System (NTFS)

- Navržen pro Windows NT, aby na rozdíl od FAT a HPFS (fsvyvíjený 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
- Disk naformátovaný s **NTFS** má následující **strukturu**:
  - Boots ektor
  - Master File Table (MTF)
  - Master File Table zóna, do které může MFT růst
  - Systémové soubory
  - Uprostřed disku je kopie (části) MFT
  - Zbývající místo je určeno pro soubory

MFT položka

- Každá položka představuje jeden soubor



- 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

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 –virtualcluster number, první cluster souboru
  - LCN –logicalcluster number, první logický cluster jedné sekvence
  - Délka v počtu clusterů
- MFT obsahuje list položek popisujících extent
  - Unix-likepoužívá strom

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 inodeuzlu 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

8. Emulace, paravirtualizace, binární překlad, VT-x, VfA.

Přednáška 10

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 výkonnější než emulace
  - Ale také je limitovaná na programy, které byly zkompilované pro daný hw

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?

Hypervizor

- Též známý jako VirtualMachineMonitor (VMM)
- (Hostitel) Vytváří a spouští virtuální stroje (hosty)
- Typ 1 –běží přímo na hw
  - Xen, VMWareESX, Hyper-V
- Typ 2 –sám je hostován v OS
  - VirtualBox, WMVarePlayer

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
  - 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

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 ne privilegované 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

int10h

- Mějme program pro real-mode, který se snaží změnit mód obrazovky do textového režimu CGA 80x25x16/8
  movax, 3
  int10h
- 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
  movax, 3
  pushf
  call EmulatedISR10h

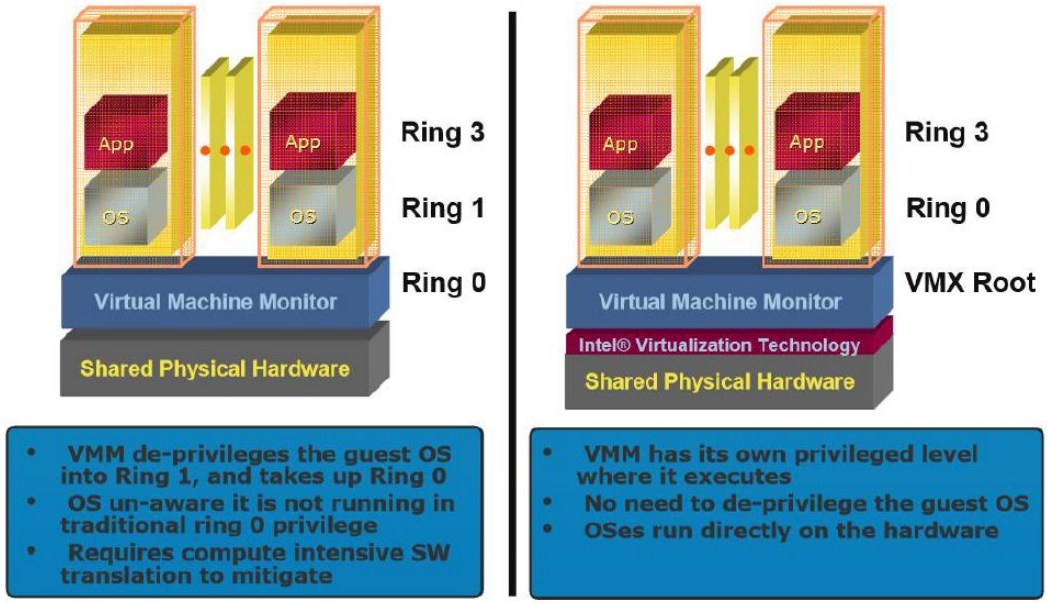
Pasti

- Mějme program, který obsahuje ochranu proti zpětnému inženýrství, pro jednoduchost uvažujme následující 3 byty
  - *dboebh, offh, ocoh*
  - Jedná se o instrukce *jmp-1* a *inceax*, které sdílejí byte *offh*
  - offh 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



TLB

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

Stránkování

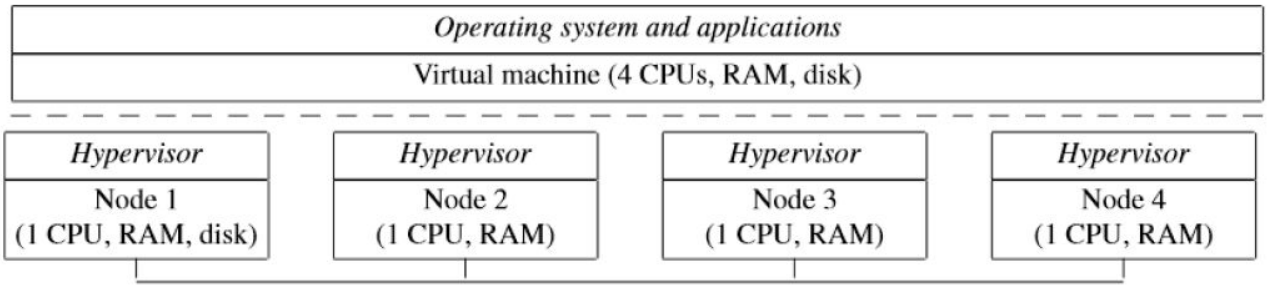
- Shadow page-table jsou tabulky stránek hostujícího OS
- Tabulka stránek hostujícího OS je bez VT-x read-only, což umožní zachytit pokus o její modifikaci a následně synchronizovat shadow verzi
  - Jenomže pokus o zápis by generoval vyjímku, a to je pomalé
- VT-x má proto koncept zanořených/rozšířených tabulek stránek, který toto eliminuje
- Host používá tabulku stránek, jak byl zvyklý, ale fyzická adresa hosta se ještě přes rozšířenou tabulku, tj. zanoření, převede na fyzickou adresu hostitele

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

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



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!