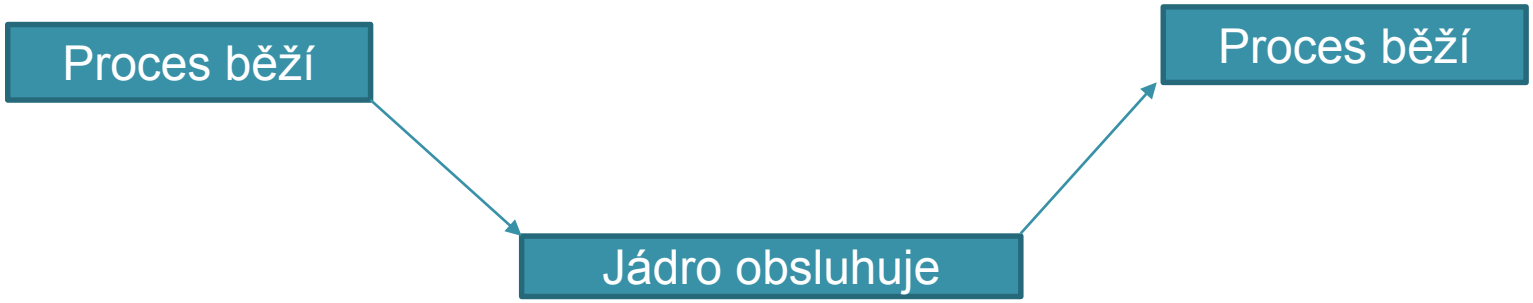
KIV Operační systémy

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

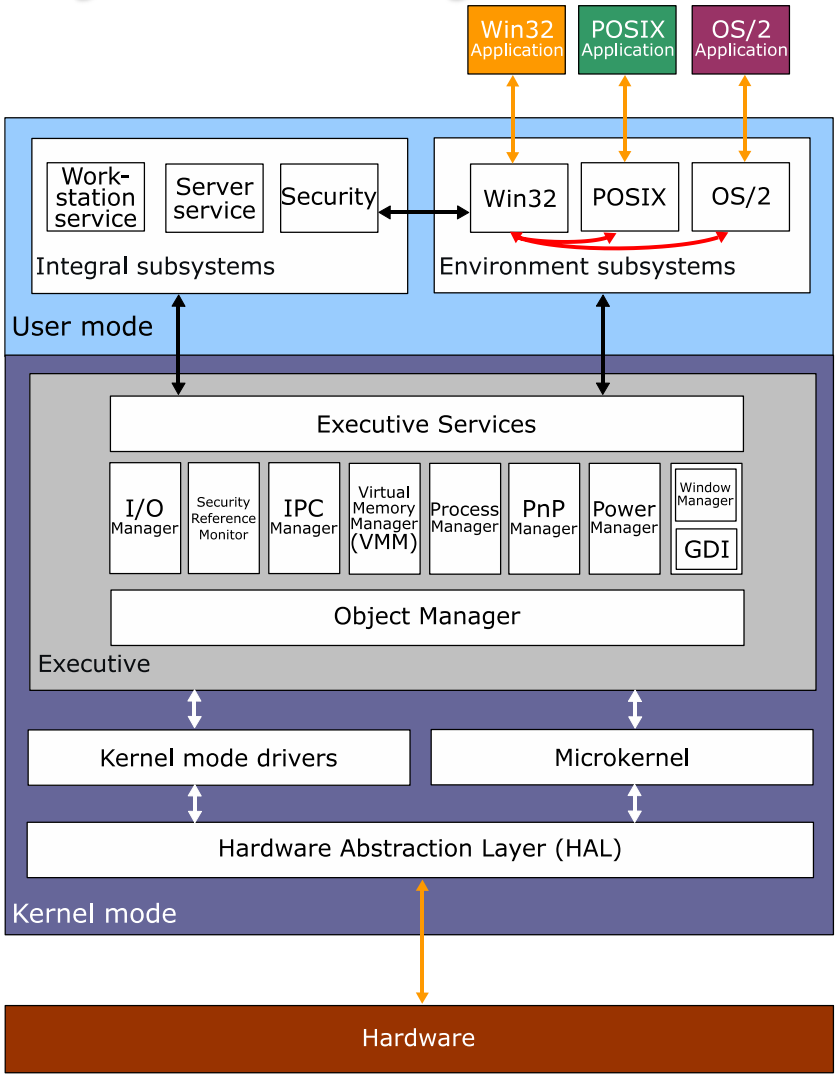
Mikrojádro – optimalizace

- 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

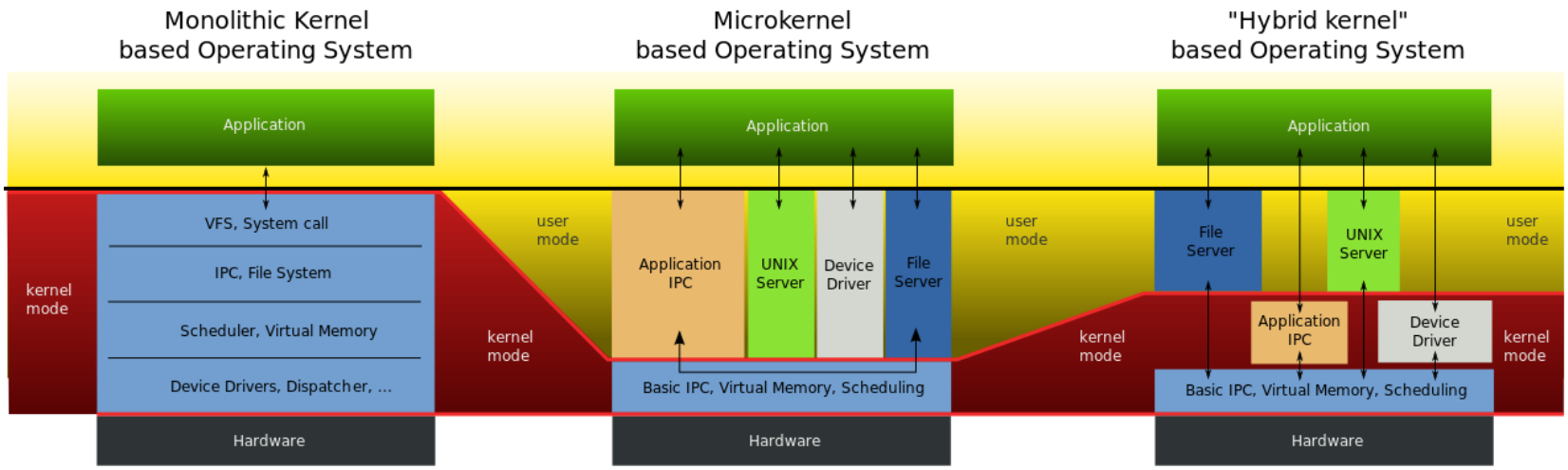
- Protože je mikrojádro pomalé, vývoj OS začne monolitickým jádrem
- Výkonnostně kritické části kódu pak zůstanou v jádře, zatímco ostatní se přesunou na úroveň s méně privilegovanou CPL
  - Např. ovladače třetích stran, které často padají a zákazníci si myslí, že je to špatným OS

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

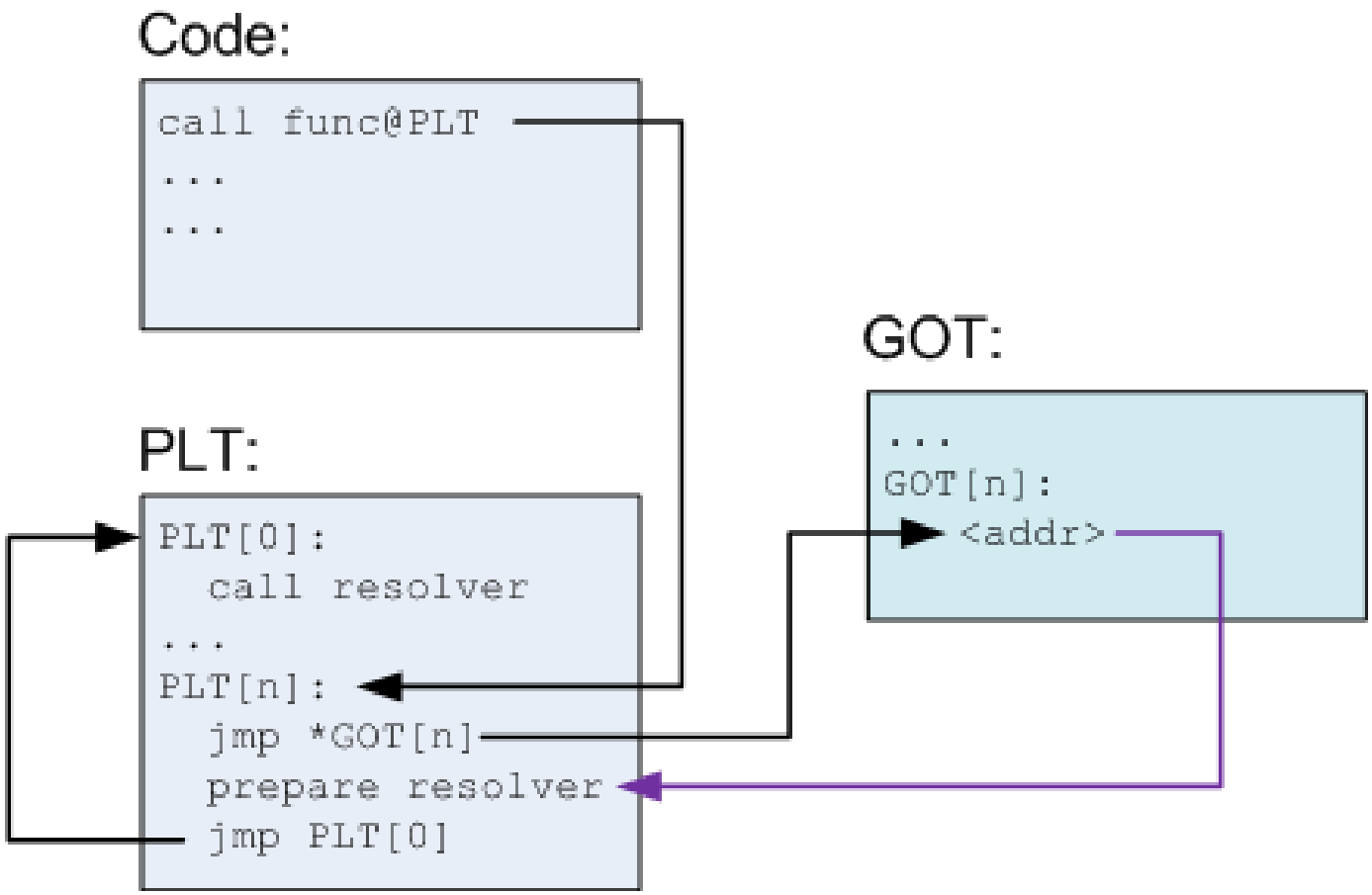
- Relokace vyžaduje přepsání kódu knihovny jejím zavaděčem
- Lepší je vytvořit tabulku, Global Offset Table (GOT), ve které budou adresy symbolů knihovny, a které (adresy) tam zapíše zavaděč knihovny
  - Kód tedy bude symbol dereferencovat 2x
  - GOT bude privátní pro každý proces
    - A nezměněný kód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

Lazy Binding

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

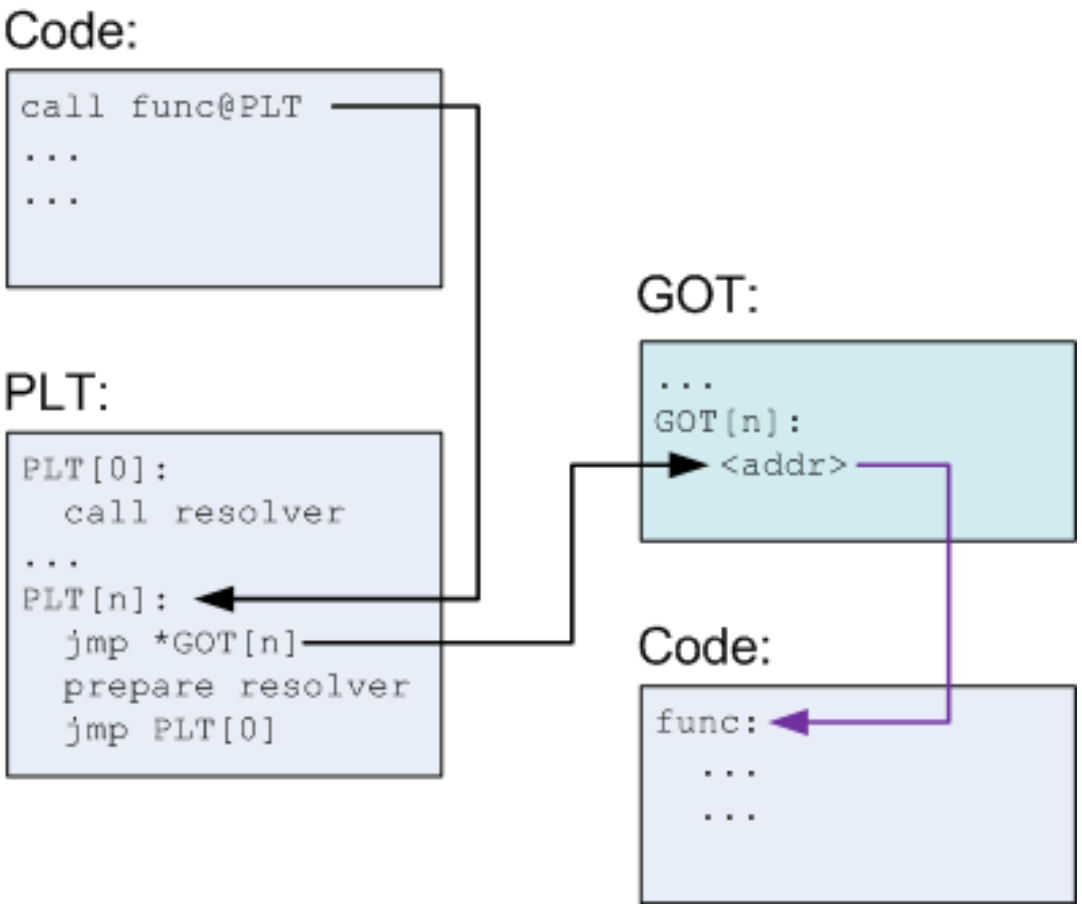
1. Proces volá funkci knihovny
   - Každá knihovnou exportovaná funkce má své ordinální číslo n – tj. volá se adresa PLT[n]
2. PLT[n] ale zatím ukazuje na rutinu zavaděče, která se zavolá jako první a resolvuje adresu rutiny do GOT než zavolá vlastní funkci
   - Než je funkce zavolána, GOT se upraví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

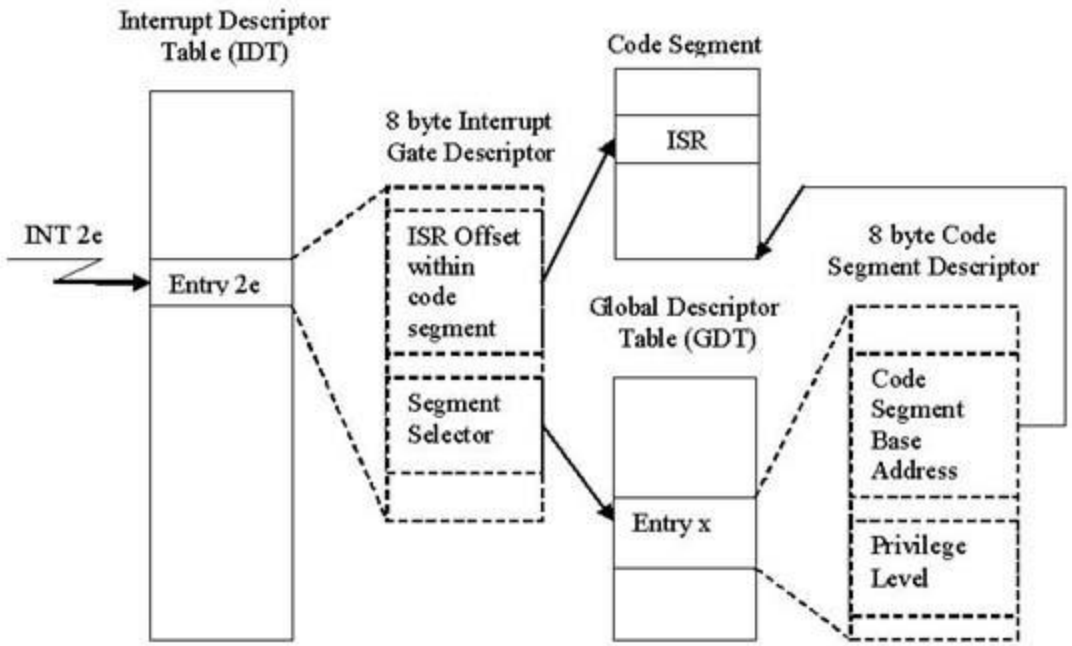
Předání parametrů jádru

- 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

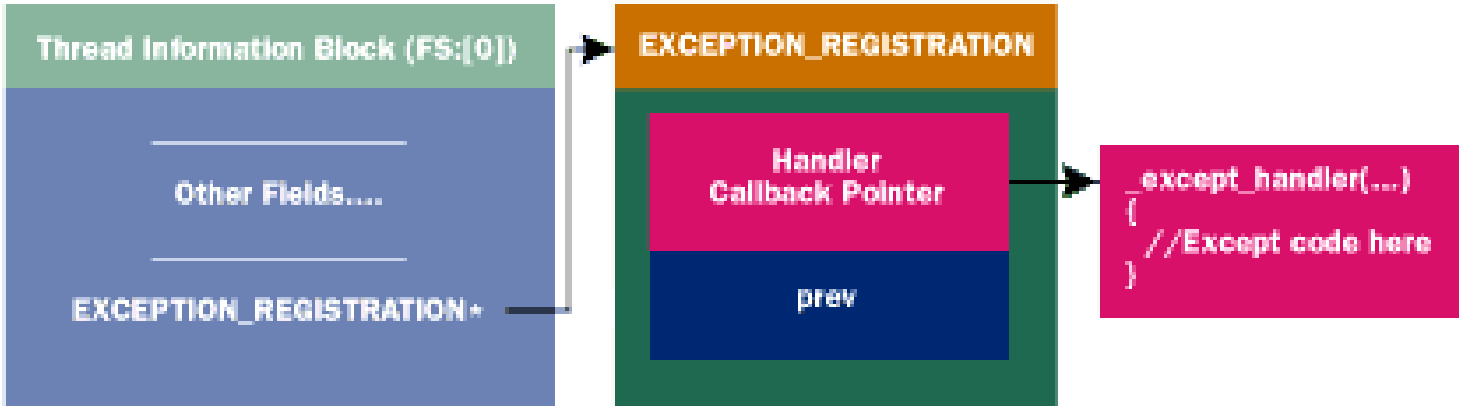
```
exc_0d_handler:
    push gs
    mov gs,ZEROBASED_DATA_SELECTOR
    mov word [gs:0xb8000],'D '
    ;; D in the top-left corner means we're handling
    ;;  a GPF exception right ATM.

    ;; 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