B4B35OSY: Operační systémy

Lekce 2. Systémové volání

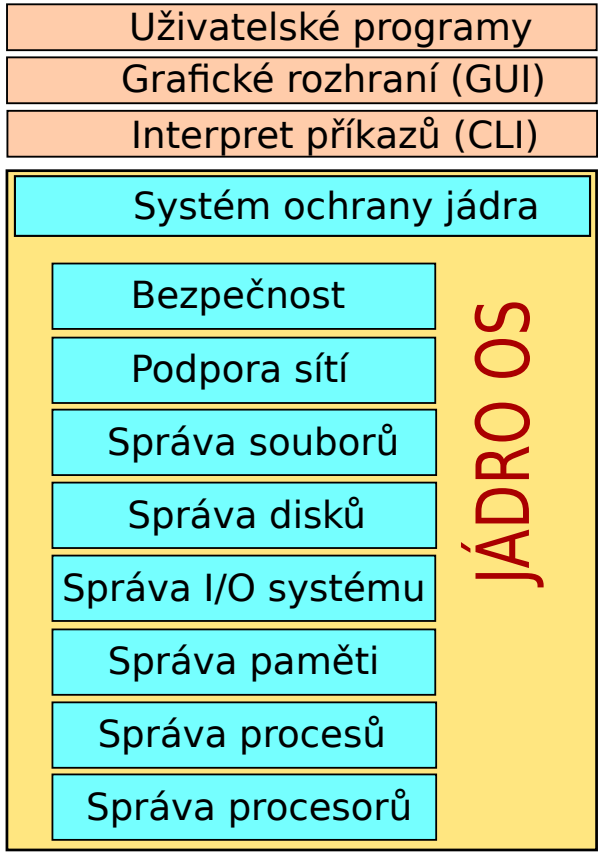
Petr Štěpán
stepan@fel.cvut.cz

September 10, 2020 (draft)

Outline

Složky OS

- Správa procesorů
- Správa procesů
- Správa (hlavní, vnitřní) paměti
- Správa I/O systému
- Správa disků – vnější (sekundární) paměti
- Správa souborů
- Podpora sítí
- Bezpečnost - security
- Systém ochrany jádra



Interpret příkazů

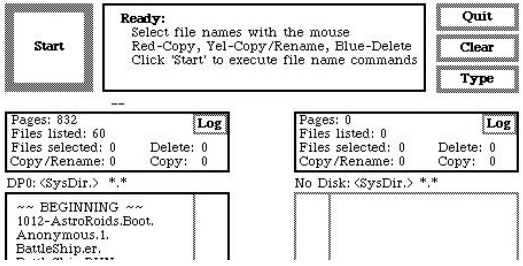
- Většina zadání uživatele je předávána operačnímu systému řídícími příkazy, které zadávají požadavky na
  - správu a vytváření procesů
  - ovládání I/O
  - správu sekundárních pamětí
  - správu hlavní paměti
  - zpřístupňování souborů
  - komunikaci mezi procesy
  - práci v síti, ...
- Program, který čte a interpretuje řídicí příkazy se označuje v různých OS různými názvy
  - Command-line interpreter (CLI), shell, cmd.exe, sh, bash, ...
  - Většinou rozumí jazyku pro programování dávek (tzv. skriptů)
  - Interpret příkazů není částí jádra OS
  - Interpret příkazů pracuje v uživatelském režimu, který je stejný jako pro Vaše programy

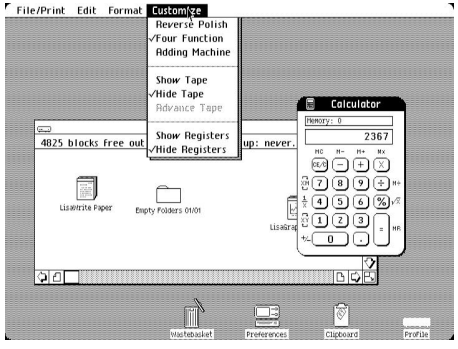
Systémové nástroje

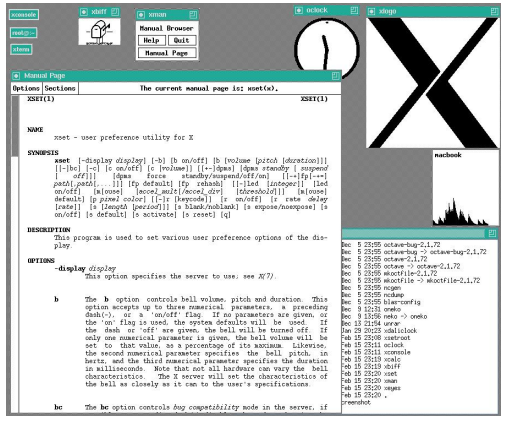
- Poskytují prostředí pro vývoj a provádění programů
- Typická skladba
  - Práce se soubory, editace, kopírování, katalogizace, ...
  - Získávání, definování a údržba systémových informací
  - Modifikace souborů
  - Podpora prostředí pro různé programovací jazyky
  - Sestavování programů
  - Komunikace
  - Anti-virové programy
  - Šifrování a bezpečnost
  - Aplikační programy z různých oblastí
- Systémové nástroje pracují v uživatelském režimu, který je stejný jako pro Vaše programy

GUI

- První Xerox Alto (1973)
- Apple Lisa (1983)
- X window (1984) – MIT, možnost vzdáleného terminálu přes síť
- Windows 1.0 pro DOS (1985)
- Windows 3.1 (1992) podpora 32-bitových procesorů s ochranou paměti, vylepšená grafika
- Windows NT (1993) – preemptivní multitasking, předchůdce Windows XP (2001)







Jádro OS

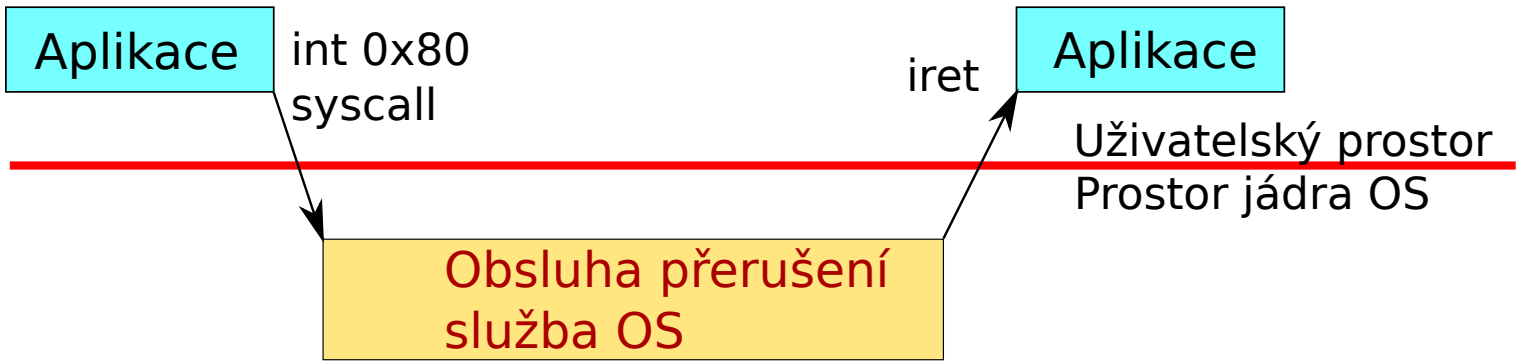
- Poskytuje ochranu/izolaci
  - Aplikačních programů mezi sebou
  - Hardwaru před škodlivými aplikacemi
  - Dat (souborů) před neoprávněnou manipulací
- Řídí přidělování zdrojů aplikacím
  - Paměť, procesorový čas, přístup k HW, síti, …
- Poskytuje aplikacím služby
  - Jaké?

Ochrana jádra OS

- Ochrana jádra
  - mechanismus pro kontrolu a řízení přístupu k systémovým a uživatelským zdrojům (paměť, HW zařízení, soubory, …)
- Systém ochran „prorůstá“ všechny vrstvy OS
- Systém ochran musí
  - rozlišovat mezi autorizovaným a neautorizovaným použitím
  - poskytnout prostředky pro prosazení legální práce
- Detekce chyb
  - Chyby interního a externího hardware
    - Chyby paměti, výpadek napájení
    - Chyby na vstupně/výstupních zařízeních či mediích („díra“ na disku)
  - Softwarové chyby
    - Aritmetické přetečení, dělení nulou
    - Pokus o přístup k „zakázaným“ paměťovým lokacím (ochrana paměti)
  - OS nemůže obsloužit žádost aplikačního programu o službu
    - Např. „k požadovanému souboru nemáš právo přistupovat“

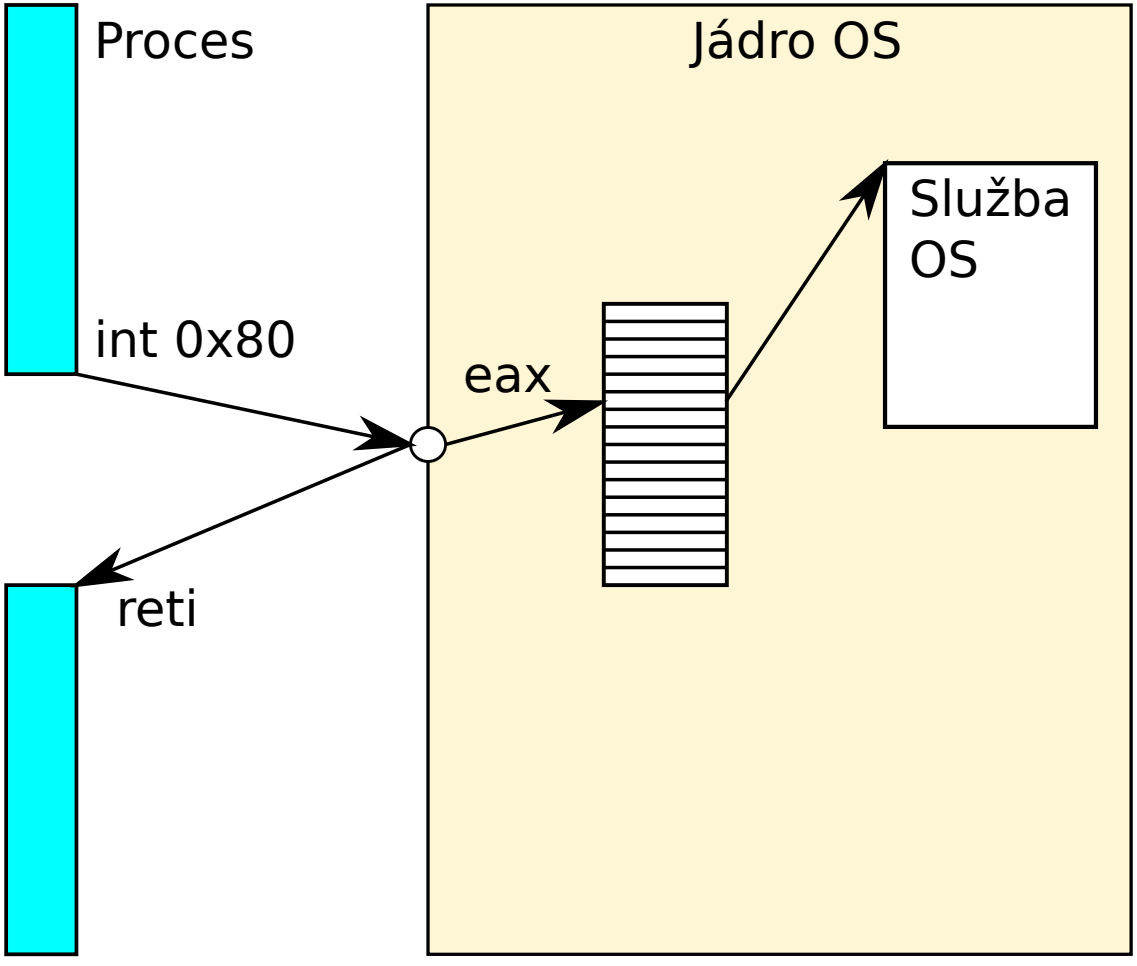
Ochrana jádra OS

- Základ ochrany OS, přechod do systémového módu
  - Intel x86 rozlišuje 4 úrovně ochrany (priviledge level): 0 – jádro OS, 3 – uživatelský mód
  - Jiné architektury mají většinou jen dva módy (jeden bit ve stavovém slově)
  - V uživatelském módu jsou některé instrukce zakázané (opakování – jaké?)
- Přechod z uživatelského módu do systémového
  - pouze programově vyvolaným přerušením
  - speciální instrukce (trap, int, sysenter, swi, ...)
  - nejde spustit cokoliv, spustí se pouze kód připravený operačním systémem
  - Systémová volání – služby jádra (system calls)
- Přechod ze systémového módu do uživatelského:
  - Speciální instrukce či nastavení odpovídajících bitů ve stavovém slově FLAGS
  - Návrat z přerušení



Ochrana jádra OS

- Uživatel má do jádra OS přístup pouze přes obsluhu přerušení



Služby jádra OS

x86 System Call Example – Hello World on Linux

```
        .section .rodata
greeting:
        .string "Hello World\n"

        .text
        .global _start
_start:
        mov $4,%eax             ; write is syscall no. 4
        mov $1,%ebx             ; file descriptor, 1 je stdout
        mov $greeting,%ecx      ; address of the data
        mov $12,%edx            ; length of the data
        int $0x80               ; call the system
```

Služby jádra OS

- Služby jádra jsou číslovány
  - Registr eax obsahuje číslo požadované služby
  - Ostatní registry obsahují parametry, nebo odkazy na parametry
  - Problém je přenos dat mezi pamětí jádra a uživatelským prostorem
    - malá data lze přenést v registrech – návratová hodnota funkce
    - velká data – uživatel musí připravit prostor, jádro z/do něj nakopíruje data, předává se pouze adresa (ukazatel)
- Volání služby jádra na strojové úrovni není komfortní
  - Je nutné použít assembler, musí být dodržena volací konvence
  - Zapouzdření pro programovací jazyky – API
  - Základem je běhová knihovna jazyka C (libc, C run-time library)
- Linux system call table

  `http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html`
- Windows system call table

  `http://j00ru.vexillium.org/ntapi/`

Application Binary Interface – ABI

- Definuje rozhraní na úrovni strojového kódu:
  - V jakých registrech se předávají parametry
  - V jakém stavu je zásobník
  - Zarovnání vícebytových hodnot v paměti
- ABI se liší nejen mezi OS, ale i mezi procesorovými architekturami stejného OS.
  - Např: Linux i386, amd64, arm, …
  - Možnost podpory více ABI: int 0x80, sysenter, 32/64 bit

ABI Linuxu

32 bitový systém (i386):
instrukce `int 0x80`
EIP a EFLAGS se ukládají na zásobník

Popis	Registr
číslo syscall	eax
první argument	ebx
druhý argument	ecx
třetí argument	edx
čtvrtý argument	esi
pátý argument	edi
šestý argument	ebp

64 bitový systém (amd64):
instrukce `syscall`
rychlejší přechod do jádra OS,
RIP a RFLAGS ukládá do registrů RCX a R11

Popis	Registr
číslo syscall	rax
první argument	rdi
druhý argument	rsi
třetí argument	rdx
čtvrtý argument	r10
pátý argument	r9
šestý argument	r8

Application Programming Interface – API

- Definice rozhraní pro služby OS (system calls) na úrovni zdrojového kódu
  - Jména funkcí, parametry, návratové hodnoty, datové typy
- POSIX (IEEE 1003.1, ISO/IEC 9945)
  - Specifikuje nejen system calls ale i rozhraní standardních knihovních podprogramů a dokonce i povinné systémové programy a jejich funkcionalitu (např. ls vypíše obsah adresáře)
  - `http://www.opengroup.org/onlinepubs/9699919799/nframe.html`
- Win API
  - Specifikace volání základních služeb systému v MS Windows
- Nesystémová API:
  - Standard Template Library pro C++
  - Java API
  - REST API webových služeb

Volání služeb jádra OS přes API

Aplikační program (proces) volá službu OS:

- Zavolá podprogram ze standardní systémové knihovny
- Ten transformuje volání na systémové ABI a vykoná instrukci pro systémové volání
- Ta přepne CPU do privilegovaného režimu a předá řízení do vstupního bodu jádra
- Podle kódu požadované služby jádro zavolá funkci implementující danou službu (tabulka ukazatelů)
- Po provedení služby se řízení vrací aplikačnímu programu s případnou indikací úspěšnosti

POSIX

- Portable Operating System Interface for Unix – IEEE standard pro systémová volání i systémové programy
- Standardizační proces začal 1985 – důležité pro přenos programů mezi systémy
- 1988 POSIX 1 Core services – služby jádra
- 1992 POSIX 2 Shell and utilities – systémové programy a nástroje
- 1993 POSIX 1b Real-time extension – rozšíření pro operace reálného času
- 1995 POSIX 1c Thread extension – rozšíření o vlákna
- Po roce 1997 se spojil s ISO a byl vytvořen standard POSIX:2001 a POSIX:2008

UNIX

- Operační systém vyvinutý v 70. letech v Bellových laboratořích
- Protiklad tehdejšího OS Multix
- Motto: **V jednoduchosti je krása**
- Ken Thompson, Dennis Ritchie
- Pro psaní OS si vyvinuli programovací jazyk C
- Jak UNIX tak C přežilo do dnešních let
- Linux, FreeBSD, *BSD, GNU Hurd, VxWorks...

Unix v kostce

- Všechno je soubor[1]
- Systémová volání pro práci se soubory:
  - open(pathname, flags) → file descriptor (celé číslo)
  - read(fd, data, délka)
  - write(fd, data, délka)
  - ioctl(fd, request, data) – vše ostatní co není read/write
  - close(fd)
- Souborový systém:
  - /bin – aplikace
  - /etc – konfigurace
  - /dev – přístup k hardwaru
  - /lib – knihovny

[1] až na síťová rozhraní, která v době vzniku UNIXu neexistovala

POSIX dokumentace

- Druhá kapitola manuálových stránek
- Příkaz (např. v Linuxu): man 2 ioctl

```
ioctl(2) -- Linux man page

Name

  ioctl -- control device

Synopsis

  #include <sys/ioctl.h>
  int ioctl(int d, int request, ...);

Description

  The ioctl() function manipulates the underlying device parameters of
  special files. In particular, many operating characteristics of
  character special files (e.g., terminals) may be controlled with
  ioctl() requests. The argument d must be an open file descriptor.

  The second argument is a device-dependent request code. The third
  argument is an untyped pointer to memory. It's traditionally char
  *argp (from the days before void * was valid C), and will be so named
  for this discussion.
```

POSIX dokumentace

Pokračování

```
  An ioctl() request has encoded in it whether the argument is an in
  parameter or out parameter, and the size of the argument argp in
  bytes. Macros and defines used in specifying an ioctl() request are
  located in the file <sys/ioctl.h>.

Return Value

  Usually, on success zero is returned. A few ioctl() requests use the
  return value as an output parameter and return a nonnegative value on
  success. On error, -1 is returned, and errno is set appropriately.

Errors

  EBADF    d is not a valid descriptor.
  EFAULT   argp references an inaccessible memory area.
  EINVAL   Request or argp is not valid.
  ENOTTY   d is not associated with a character special device.
  ENOTTY   The specified request does not apply to the kind of object
  ^^I   that the descriptor d references.

Notes

  In order to use this call, one needs an open file descriptor. Often
  the open(2) call has unwanted side effects, that can be avoided under
  Linux by giving it the O_NONBLOCK flag.

See Also

  execve(2), fcntl(2), ioctl_list(2), open(2), sd(4), tty(4)
```

Přehled služeb jádra

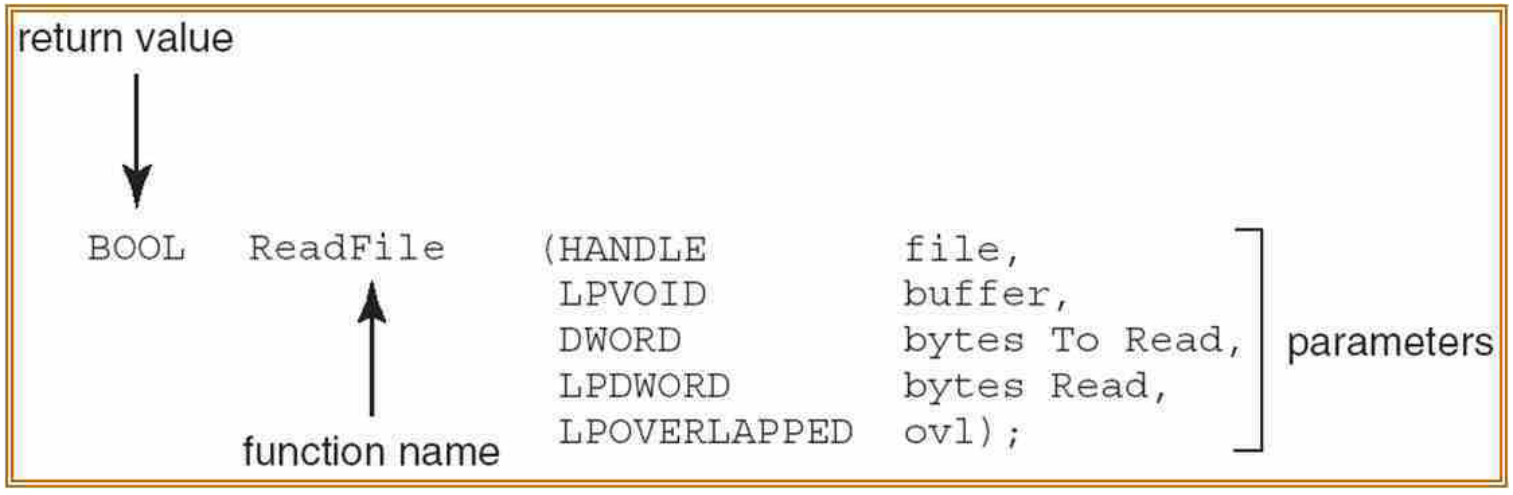
- Práce se soubory
  - open, close, read, write, lseek
- Správa souborů a adresářů
  - mkdir, rmdir, link, unlink, mount, umount, chdir, chmod, stat
- Správa procesů
  - fork, waitpid, execve, exit, kill, signal

Windows system call API

- Nebylo plně popsáno, skrytá volání využívaná pouze spřátelenými stranami
- Garantováno pouze API poskytované DLL knihovnami (kernel32.dll, user32.dll, …)
- Win16 – 16 bitová verze rozhraní pro Windows 3.1
- Win32 – 32 bitová verze od Windows NT
- Win32 for 64-bit Windows – 64 bitová verze rozhraní Win32
- Nová window mohou zavést nová volání, případně přečíslovat staré služby.

Windows API příklad

- Funkce ReadFile() z Win32 API – funkce, která čte z otevřeného souboru



- Parametry předávané funkci ReadFile()
  - HANDLE file – odkaz na soubor, ze kterého se čte
  - LPVOID buffer – odkaz na buffer pro zapsání dat ze souboru
  - DWORD bytesToRead – kolik bajtů se má přečíst
  - LPDWORD bytesRead – kolik bajtů se přečetlo
  - LPOVERLAPPED ovl – zda jde o blokující či asynchronní čtení

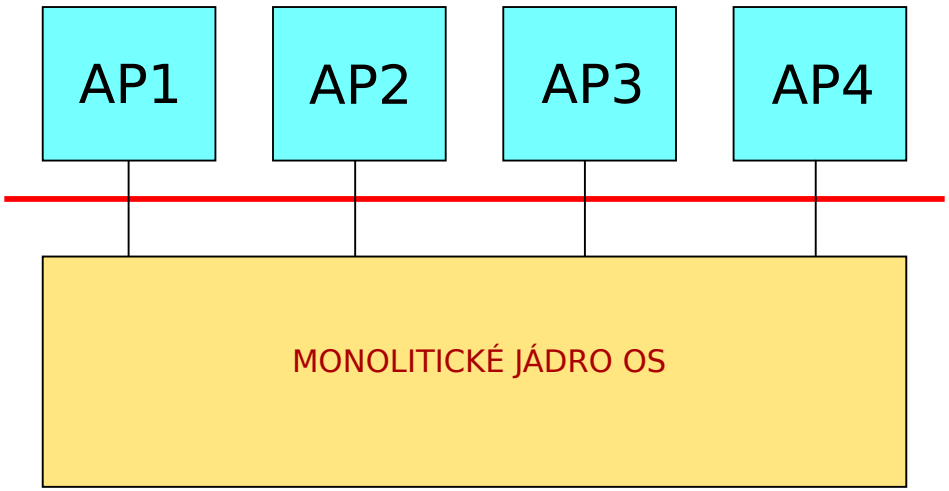
Porovnání POSIX a Win32

POSIX	Win32	Popis
fork	CreateProcess	Vytvoř nový proces
execve	–	CreateProcess = fork + execve
waitpid	WaitForSingleObject	Čeká na dokončení procesu
exit	ExitProcess	Ukončí proces
open	CreateFile	Vytvoří nový soubor nebo otevře existující
close	CloseHandler	Zavře soubor
read	ReadFile	Čte data ze souboru
write	WriteFile	Zapisuje data do souboru
seek	SetFilePointer	Posouvá ukazatel v souboru
stat	GetFileAttributesExt	Vrací informace o souboru
mkdir	CreateDirectory	Vytvoří nový adresář
rmdir	RemoveDirectory	Smaže adresář souborů
link	–	Win32 nepodporuje symbolické odkazy
unlink	DeleteFile	Zruší existující soubor
chdir	SetCurrentDirectory	Změní pracovní adresář

POSIX služby mount, umount, kill, chmod a další nemají ve Win32 přímou obdobu a analogická funkcionalita je řešena jiným způsobem.

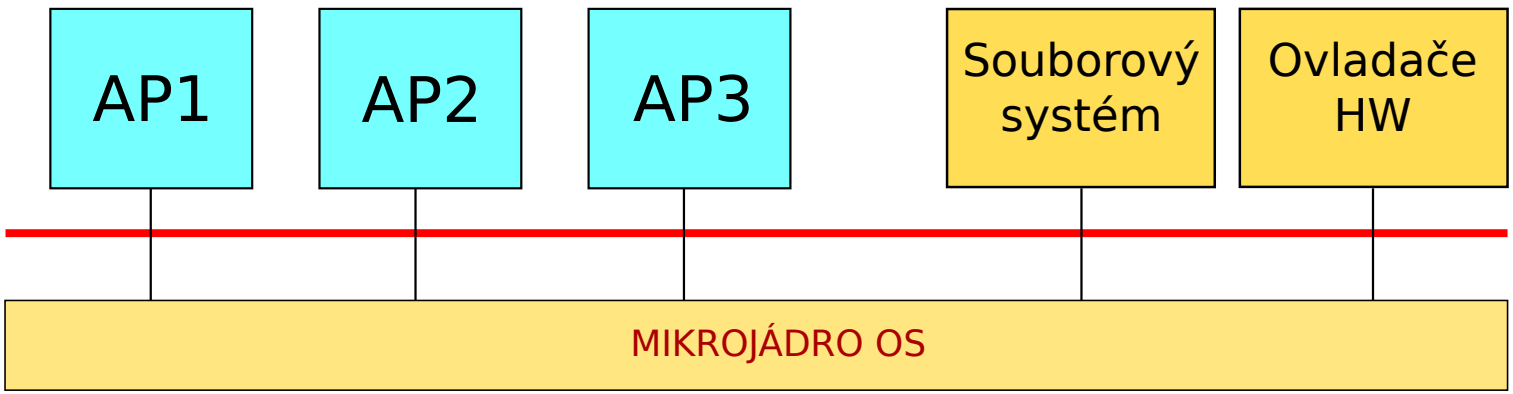
Vykonání služeb jádra OS

- Klasický monolitický OS
  - Non-process Kernel OS
  - Procesy – jen uživatelské a systémové programy
  - Jádro OS je prováděno jako monolitický (byť velmi složitý) program v privilegovaném režimu
    - „USB MIDI má přístup ke klíči k šifrování disku :-)“ CVE-2016-2384
- Služba jádra OS je typicky implementována jako kód v jádře, běžící jako přerušení využívající paměťový prostor volajícího programu

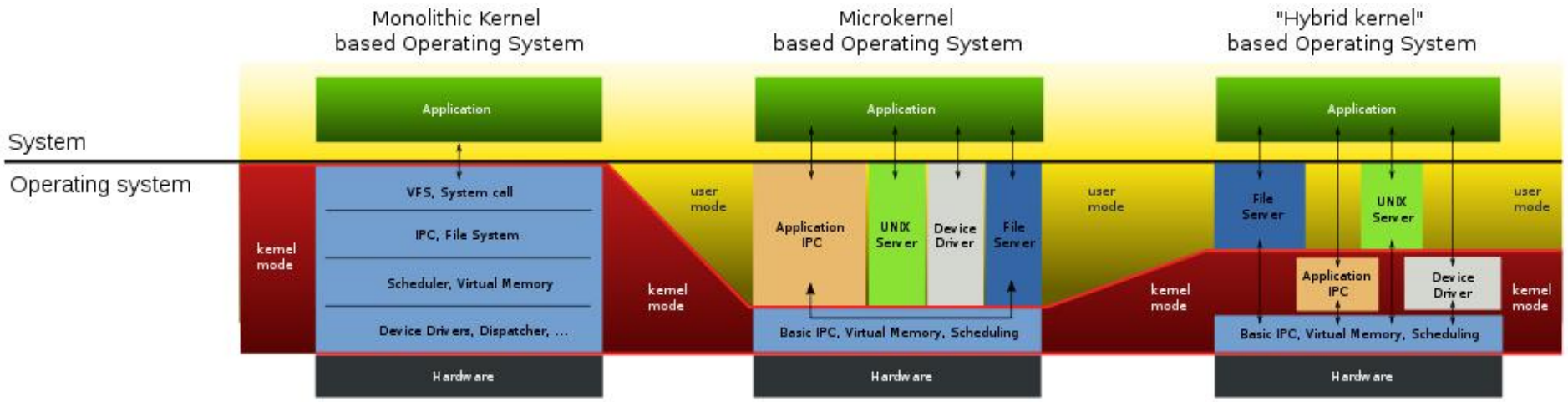


Procesově orientované jádro OS

- OS je soustavou systémových procesů
- Funkcí jádra je tyto procesy separovat ale umožnit přitom jejich kooperaci
  - Minimum funkcí je potřeba dělat v privilegovaném režimu
  - Jádro pouze ústředna pro přepojování zpráv
  - Řešení snadno implementovatelné i na multiprocesorech
- Malé jádro $\Rightarrow$ mikrojádro (μ-jádro) – (microkernel)



Porovnání JOS

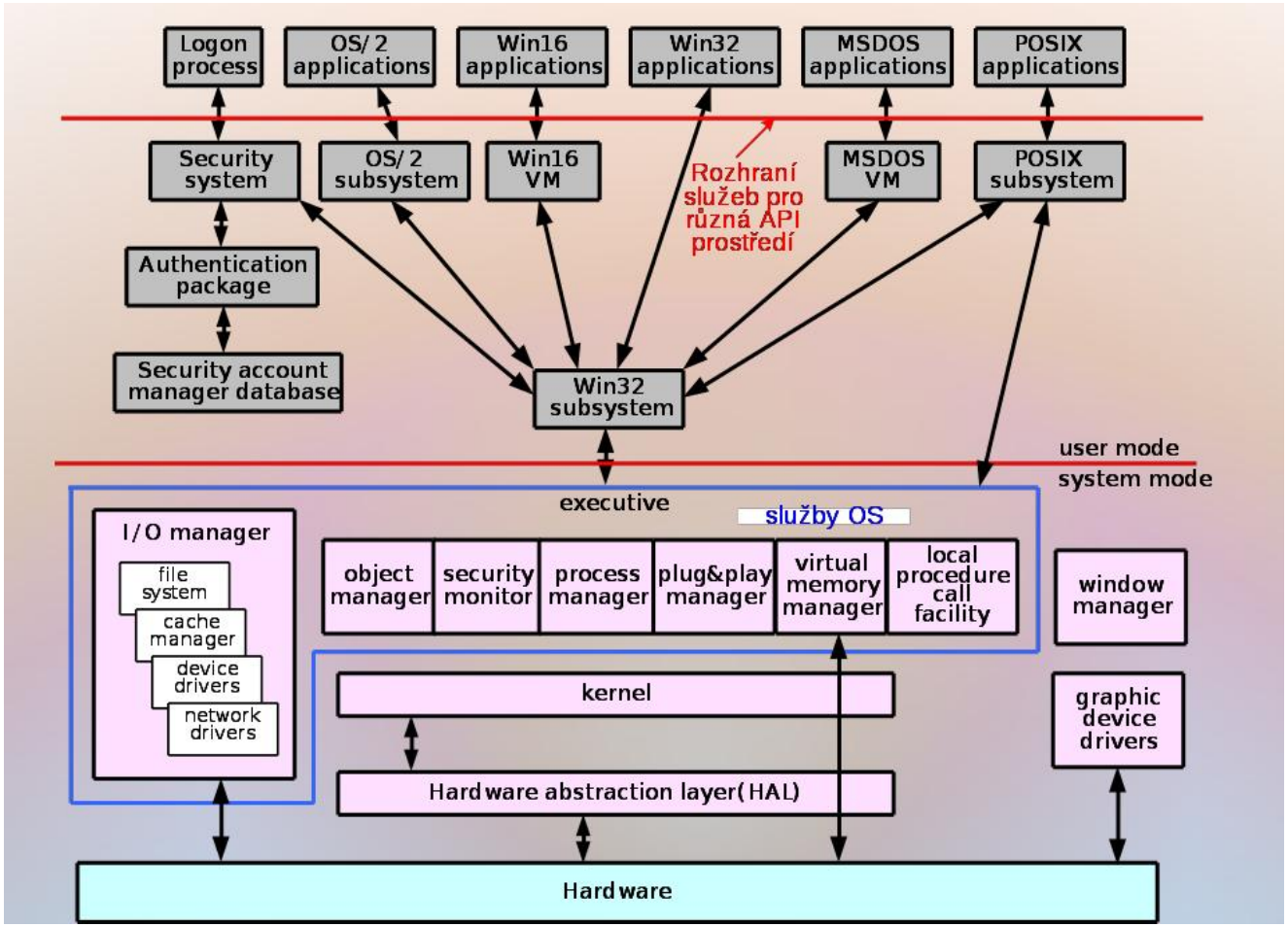


Mikrojádro – vlastnosti

- OS se snáze přenáší na nové hardwarové architektury,
  - µ-jádro je malé
- Vyšší spolehlivost – modulární řešení
  - moduly jsou snáze testovatelné
- Vyšší bezpečnost
  - méně kódu se běží v privilegovaném režimu
- Pružnější, snáze rozšiřitelné řešení
  - snadné doplňování nových služeb a rušení nepotřebných
- Služby jsou poskytovány unifikovaně
  - výměnou zpráv
- Přenositelné řešení
  - při implementaci na novou hardwarovou platformu stačí změnit µ-jádro
- Podpora distribuovanosti
  - výměna zpráv je implementována v síti i uvnitř systému
- Podpora objektově-orientovaného přístupu
  - snáze definovatelná rozhraní mezi aplikacemi a µ-jádrem
- To vše za cenu
  - zvýšené režie, volání služeb je nahrazeno výměnou zpráv mezi aplikačními a systémovými procesy

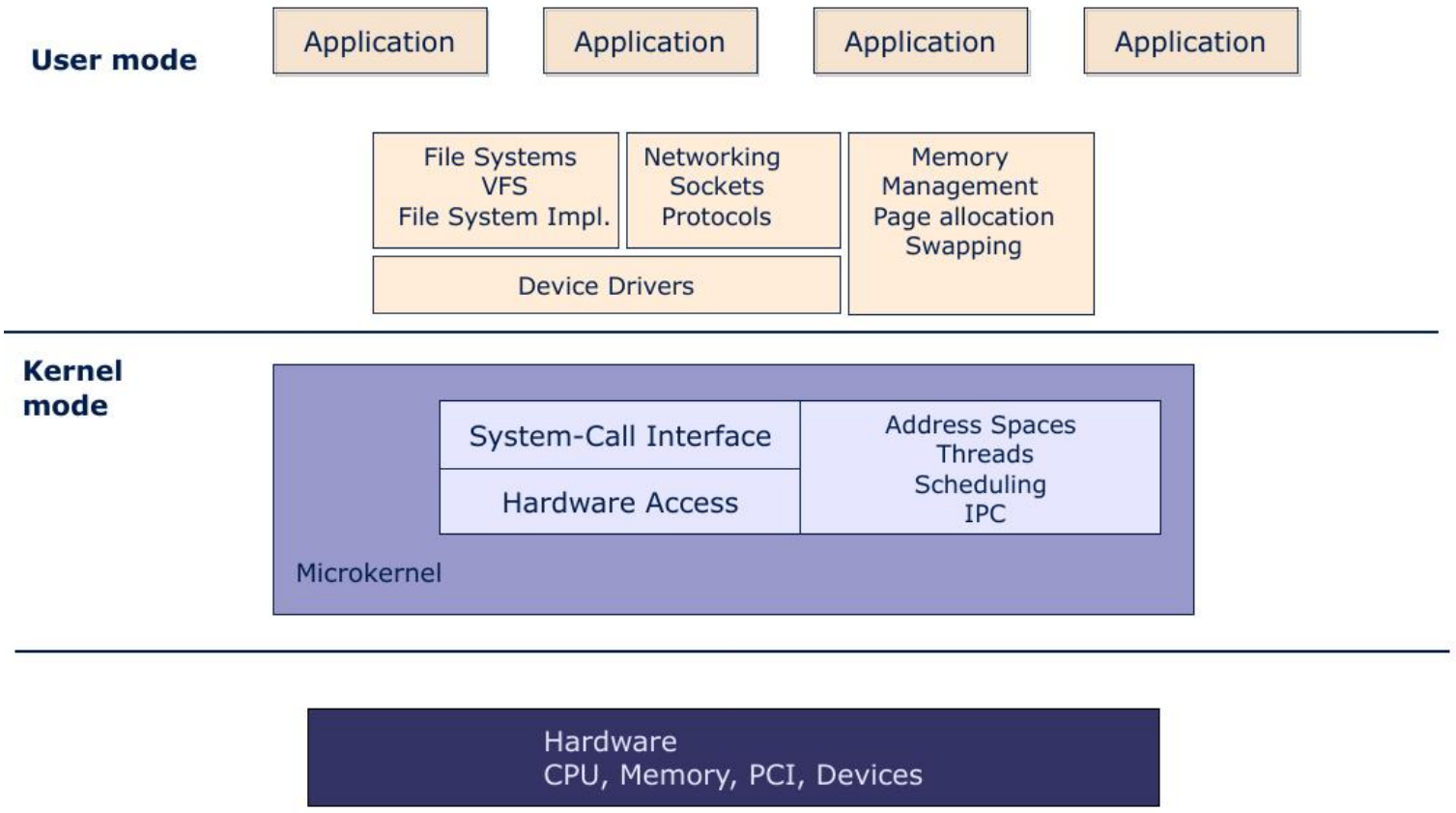
Windows (XP)

JOS Windows má architekturu μ-jádra, ale vše běží v jednom adresním prostoru, takže se jedná o monolitické jádro[2].



[2] https://techcommunity.microsoft.com/t5/Windows-Kernel-Internals/One-Windows-Kernel/ba-p/267142

L4Re – OS se skutečným μ-jádrem

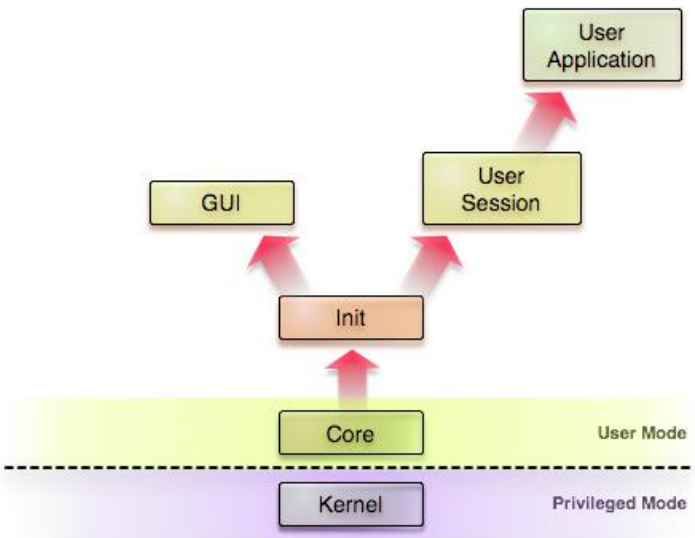


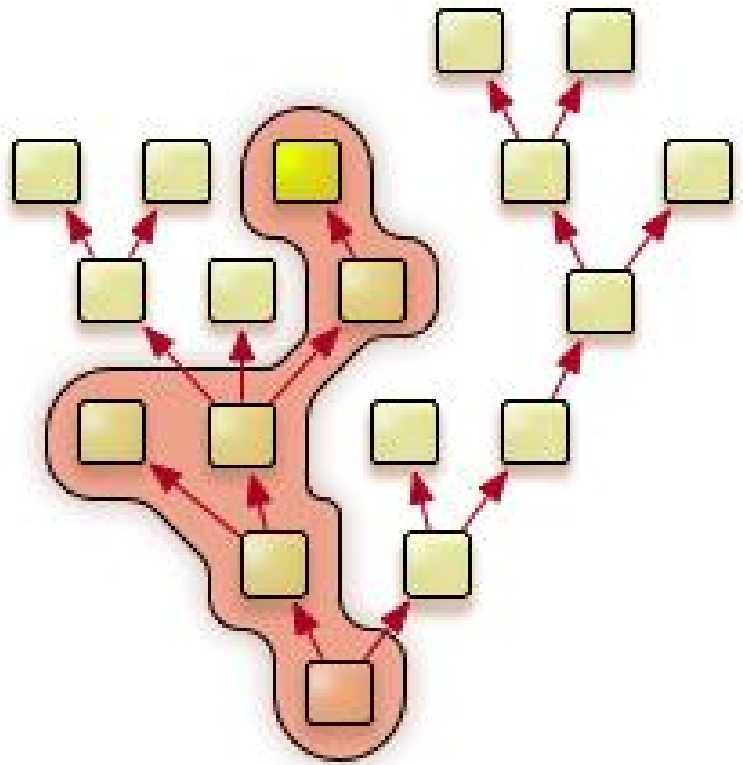
`http://os.inf.tu-dresden.de` `http://www.kernkonzept.com/`

Genode – OS se skutečným μ-jádrem

Jeden z cílů: Omezit velikost “Trustued computing base”

`http://genode.org/`





NOVA – μ-jádro

Systémová volání OS NOVA:

- call
- reply
- create_pd
- create_ec
- create_sc
- create_pt
- create_sm
- revoke
- lookup
- ec_ctrl
- sc_ctrl
- pt_ctrl
- sm_ctrl
- assign_pci
- assign_gsi

Výukový OS – bude používán na cvičení

- Víc systémových volání opravdu nemá
- PD – protection domain – proces
- EC – execution context
- SC – scheduling context
- PT – portal
- SM – semafor

Závěr - struktura OS

OS mohou (ale nemusí) být funkčně velmi složité

OS	Rok	# služeb
Unix	1971	33
Unix	1979	47
Sun OS4.1	1989	171
4.3 BSD	1991	136
Sun OS4.5	1992	219
Sun OS5.6 (Solaris)	1997	190
WinNT 4.0	1997	3443
Linux 2.0	1998	229
Linux 4.4	2016	332
NOVA	2014	15

Počty cyklů CPU spotřebovaných ve WinXP při

- Zaslání zprávy mezi procesy: 6K–120K (dle použité metody)
- Vytvoření procesu: 3M
- Vytvoření vlákna: 100K
- Vytvoření souboru: 60K
- Vytvoření semaforu: 10K–30K
- Nahrání DLL knihovny” 3M
- Obsluha přerušení/výjimky: 100K–2M
- Přístup do systémové databáze (Registry) : 20K

Počty cyklů CPU spotřebovaných v OS NOVA při

- Zaslání zprávy mezi procesy: 300–600 (dle použité metody)

Závěr - struktura OS

OS jsou velmi rozsáhlé
Údaje jsou jen orientační, Microsoft data nezveřejňuje
SLoC (Source Lines of Code) je velmi nepřesný údaj: Tentýž programový příkaz lze napsat na jediný nebo celou řadu řádků.

OS	**Rok**	**SLoC**
Windows 3.1	1992	3mil.
Windows NT 3.5	1993	4mil.
Windows 95	1995	15mil.
Windows NT 4.0	1997	16mil.
Windows 98 SR-2	1999	18mil.
Windows 2000 SP5	2002	30mil.
Windows XP SP2	2005	48mil.
Windows 7	2010	není známo
Linux 4.13 (jen JOS)	2017	16.8mil.
NOVA	2014	10tis.

Služby OS - procesy

POSIX	Popis
fork	Vytvoří nový proces jako kopii rodičovského
execve	Nahradí běžící process jiným programem - zavede ho do paměti a spustí
waitpid	Čeká na dokončení procesu potomka, přijme výsledek jeho běhu
exit	Ukončí proces, sdělí rodiči výsledek běhu (úspěch/číslo chyby)

Služby OS - fork, exit

Služba `pid_t fork(void)` vytvoří kopii procesu, která:

- má odlišný PID a rodičovský PID
- má návratovou hodnotu ze systémového volání 0 (rodičovský proces má návratovou hodnotu pid potomka)
- má kopii data a zásobníku

Služba `void exit(int status)`

- ukončí vykonávání procesu
- předá rodiči hodnotu status
- dokud rodič hodnotu nepřečte, tak nelze proces úplně odstranit z paměti

Služby OS - wait

Služba `pid_t wait(int *status)`:

- čeká na ukončení libovolného potomka
- `pid_t wait_pid(pid_t pid, int *status, int opt)` čeká na konkrétního potomka
- přijme jeho návratovou hodnotu
- `WEXITSTATUS(status)` dekóduje 8-bitů od končícího potomka
- `WIFEXITED(status)` dekóduje, zda potomek skončil normálně - tedy volání služby exit
- `WIFSIGNALED(status)` dekóduje, zda potomek skončil přijetím signálu
- další makra na detailní zjištění ukončení ptotmka

Zombie

Pokud potomek skončí a rodičovský proces ještě neskončil a nezavolal systémové volání wait, tak potomek nemůže být odstraněn z tabulky procesů.
Důvod:

- potomek musí předat rodiči výsledek svého běhu
- toto číslo musí být někde uloženo - ve struktuře, která popisuje proces potomka
- potomek nemůže běžet, ale ještě nemůže být úplně ukončen - stav zombie
- viz praktický příklad k přednášce

Fork bomb

Jednoduchý proces, který sám sebe spustí alespoň dvakrát.
Proces se začne nekontrolovaně množit a hrozí zahlcení systému.

- BASH `:() :|:& ;:`
  - definice funkce se jménem `:`
  - funkce `:` spustí funkci `:` dvakrát spojenou rourou
  - spustí se první provedení funkce `:`
- Windows – fork.bat: `%0 | %0`
  - `%0` - obdobně jako v bashi jméno spuštěného programu
  - spusť se dvakrát propojený rourou
- Perl – `perl -e "fork while fork"&`
- `https://en.wikipedia.org/wiki/Fork_bomb`