Operační systémy a databáze

Petr Štěpán, K13133
KN-E-129
stepan@fel.cvut.cz

Michal Sojka
sojkam1@fel.cvut.cz

Téma 2. Služby a architektury OS

Obsah

- Služby operačního systému
- Mechanismus volání služeb
- Monolitické operační systémy
- Operační systémy s mikrojádrem
- Virtuální počítač
- Cíle návrhu OS a složitost OS

Rekapitulace z minula

- K čemu je dobrý asembler?
- Co je to operační systém?
- Z čeho se skládá?
- Co je to jádro OS a jaké jsou jeho hlavní funkce?

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

- mechanismus pro kontrolu a řízení přístupu k systémovým a uživatelským zdrojům

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 (jaké?)

Přechod z uživatelského módu do systémového

- pouze programově vyvolaným přerušením
- speciální instrukce (trap, int, sysenter, swi, ...)
- nejde spustit cokoliv, spustí se pouze kód povolený operačním systémem
- Systémové volání – služby jádra (system calls)

Přechod ze systémového módu do uživatelského

- Speciální instrukce či nastavení odpovídajících bitů ve stavovém slově
- návrat z 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 == 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/

API

Standardy pro soustavy služeb OS (system calls)

- Rozhraní systémových služeb – API (**Application Programming Interface**)
- Definuje rozhraní 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 M$ Windows

Nesystémová API:

- Standard Template Library pro C++
- Java API

ABI

- Application Binary Interface
- 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, systenter, 32/64 bit

ABI Linuxu

32 bitový systém (i386):
instrukce int 0x80

Value	Storage
syscall nr	eax
arg 1	ebx
arg 2	ecx
arg 3	edx
arg 4	esi
arg 5	edi
arg 6	ebp

Návratová hodnota v eax

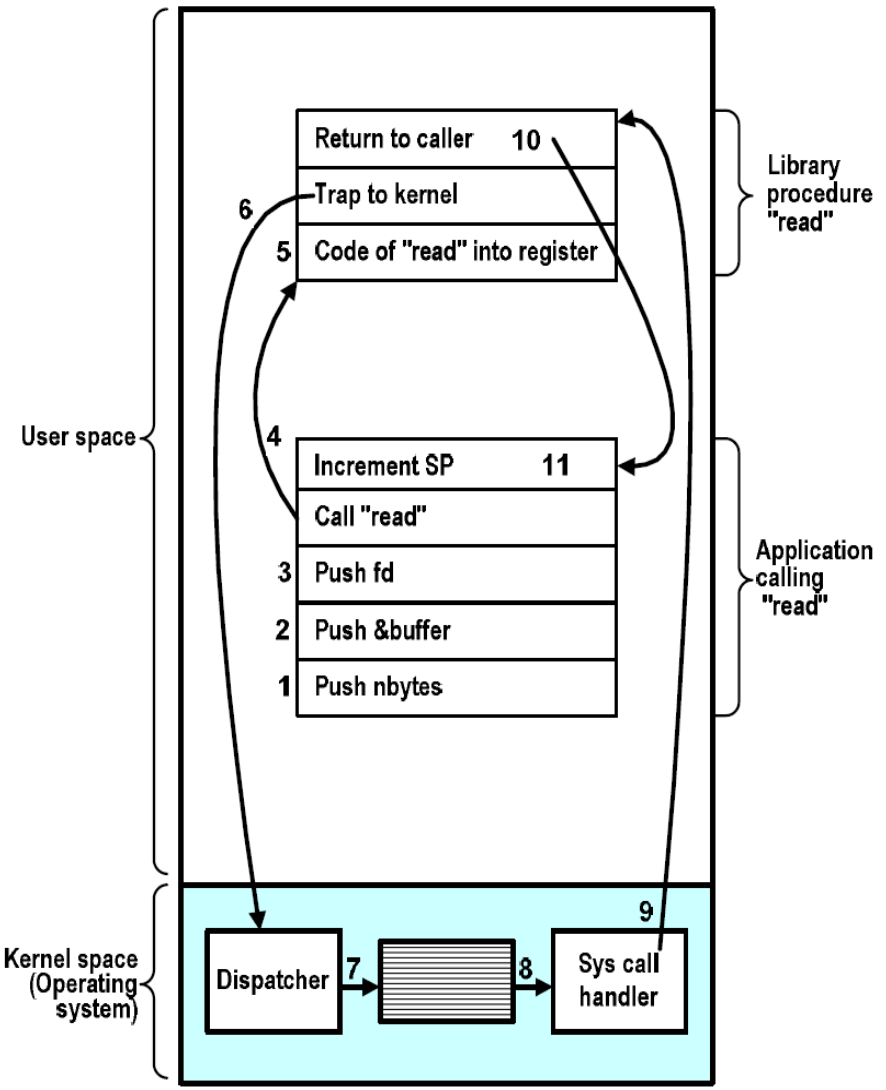
64 bitorý systém (amd64),
instrukce syscall

Value	Storage
syscall nr	rax
arg 1	rdi
arg 2	rsi
arg 3	rdx
arg 4	r10
arg 5	r9
arg 6	r8

Návratová hodnota v rax.

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

Co je to UNIX?

Portable **O**perating **S**ystem **I**nterface for Uni**x** – 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
- 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

UNIX/POSIX
Dokumentace systémových volání

- 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.

man 2 ioctl – 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 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)

Základní služby jádra OS – POSIX

Práce se soubory	
Služba	**Popis**
fd = open(filename, how, ...)	Otevře soubor pro čtení, zápis, modif. apod.
s = close(fd)	Zavře otevřený soubor (uvolní paměť)
n = read(fd, buff, nbytes)	Přečte data ze souboru do pole *buff*
n = write(fd, buff, nbytes)	Zapíše data z pole *buff* do souboru
pos = lseek(fd, offset, whence)	Posouvá *ukazatel aktuální pozice* souboru
s = stat(filename, &statbuffer)	Dodá stavové informace o souboru

Správa procesů	
Služba	**Popis**
pid = fork()	Vytvoří potomka identického s rodičem
pid = waitpid(pid, &stat, options)	Čeká až zadaný potomek skončí
s = execve(name, argv, environp)	Nahradí „obraz“ procesu jiným „obrazem“
exit(status)	Ukončí běh procesu a vrátí status

Co je to proces?

- Běžící aplikace
  - Obraz aplikace (spustitelný soubor, .exe)
  - Paměť používaná aplikací
  - Jedno nebo více „vláken“ (thread), která vykonávají kód
- Proces nemá přístup k pamětí jiného procesu (ani jádra)
- Jádro reprezentuje proces datovou strukturou (kontext procesu)
- Podobně je i každé vlákno reprezentováno datovou strukturou (kontext vlákna)

Základní správa procesů

- Primitivní shell:

```
while (TRUE) {                                  /* nekonečná smyčka */
   type_prompt( );                              /* zobraz výzvu (prompt) */
   read_command (command, parameters)       /* přečti příkaz z terminálu */
   if (fork() != 0) {                           /* vytvoř nový synovský proces */
      /* Kód rodičovského procesu */
      waitpid( -1, &status, 0);                 /* čekej na ukončení potomka */
   } else {
      /* Kód synovského procesu */
       /* Připrav prostředí (file descriptory, proměnné prostředí, atd.) */
      execve (command, parameters, 0); /* vykonej příkaz command  */
   }
}
```

Základní služby jádra OS – POSIX

Práce s adresáři souborů a správa souborů	
Služba	**Popis**
s = mkdir(name, mode)	Vytvoří nový adresář s danými právy
s = rmdir(name)	Odstraní adresář
s = link(name1, name2)	Vytvoří položku *name2* odkazující na *name1*
s = unlink(name)	Zruší adresářovou položku
s = mount(spec, name, opt)	„Namontuje“ souborový systém
s = umount(spec)	„Odmontuje“ souborový systém

Další služby	
Služba	**Popis**
s = chdir(dirname)	Změní „pracovní adresář“
s = chmod(fname, mode)	Změní „ochranné příznaky“ souboru
s = kill(pid, signal)	Zašle *signál* danému procesu
	a mnoho dalších služeb

Windows API

Nebylo plně popsáno, skrytá volání využívaná pouze spřátelenými stranami

MS developers mají privilegovaný přístup k popisu systémových volání

Win16 – 16 bitová verze rozhraní pro Windows 3.1

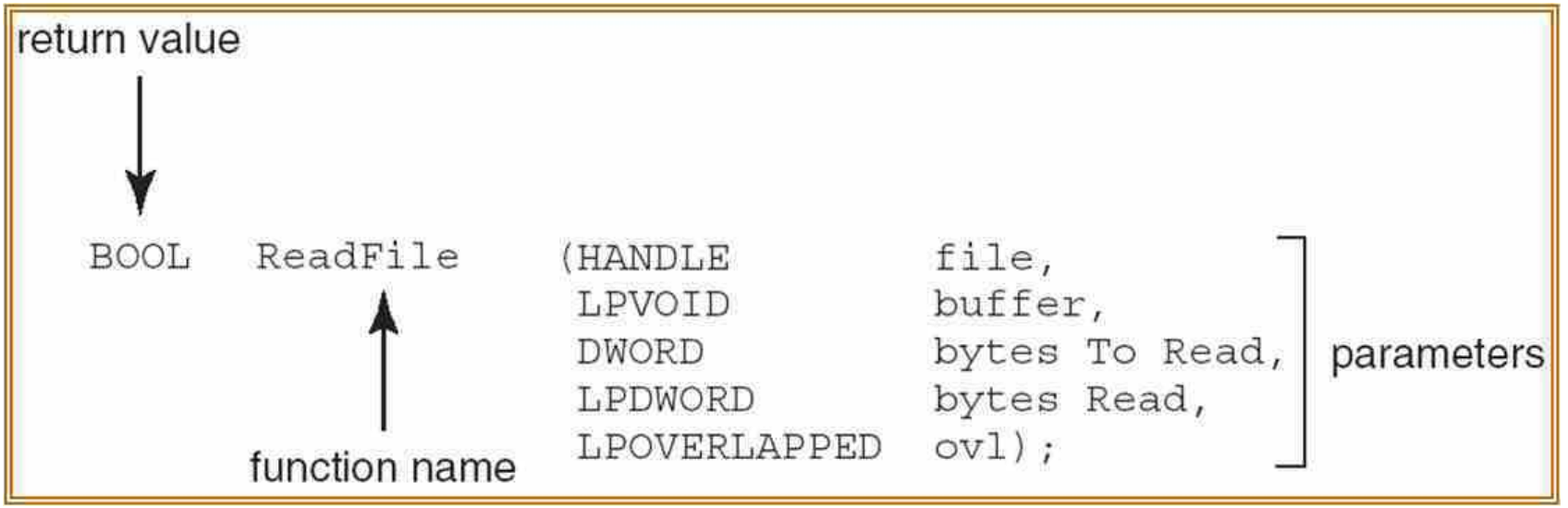
Win32 – 32 bitová verze od Windows NT

Win32 for 64-bit Windows – 64 bitová verze rozhraní Win32

Nová window mohou zavést nová volání, případně přečíslování starých služeb.

Příklad Win API

- 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 blokované čtení

Porovnání služeb POSIX a Win32

POSIX	Win32	Popis
fork	CreateProcess	Vytvoř nový proces
waitpid	WaitForSingleObject	Může čekat na dokončení procesu
execve	--	CreateProcess = fork + execve
exit	ExitProcess	Ukončí proces
open	CreateFile	Vytvoří nový soubor nebo otevře existující
close	CloseHandle	Zavře soubor
read	ReadFile	Čte data ze souboru
write	WriteFile	Zapisuje data do souboru
lseek	SetFilePointer	Posouvá ukazatel v souboru
stat	GetFileAttributesExt	Vrací různé informace o souboru
mkdir	CreateDirectory	Vytvoří nový adresář souborů (složku)
rmdir	RemoveDirectory	Smaže adresář souborů
link	--	Win32 nepodporuje „spojky“ v soub. systému
unlink	DeleteFile	Zruší existující soubor
chdir	SetCurrentDirectory	Změní pracovní adresář

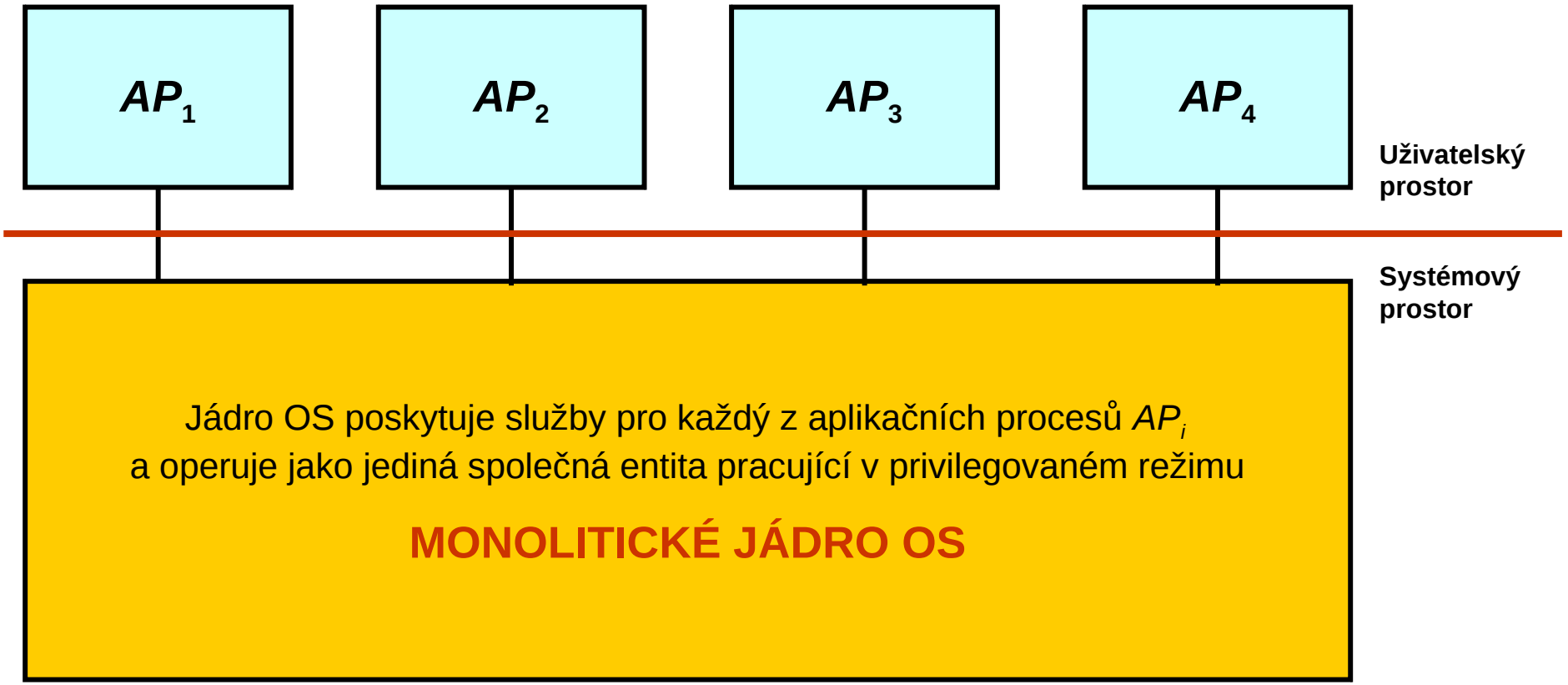
POSIX služby mount, umount, kill, chmod a další nemají ve Win32 přímou obdobu a analogická funkcionalita je řešena jiným způsobem

Vykonávání služeb v klasickém 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 OS je typicky implementována jako kód v jádře, běžící „v kontextu“ daného procesu/vlákna
- Některé služby jsou částečně (nebo i úplně) implementovány v uživatelském prostoru systémovou knihovnou.
- Služba/systémové volání
  - Procesor se přepne do systémového režimu, nepřepíná se však kontext volajícího procesu/vlákna
  - K přepnutí kontextu (přechodu od jednoho procesu k jinému: $proces_1$ – OS – $proces_2$) dochází jen, je-li to nutné z hlediska plánování procesů po dokončení služby (např. volání „sleep“).

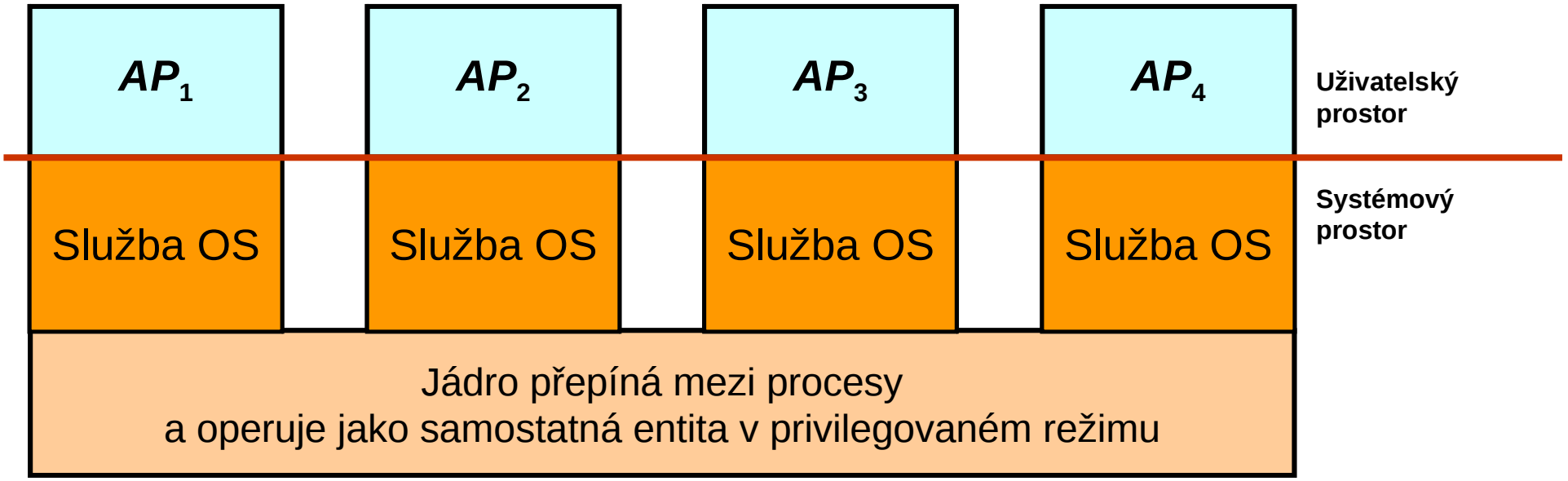
Služba OS plně jako součást JOS

- Tradiční řešení



Služba OS jako součást procesu

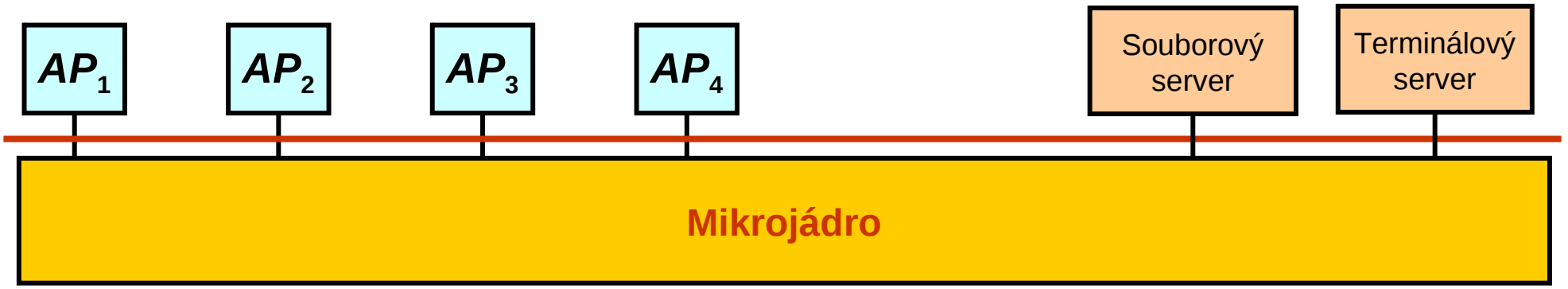
- Alternativní řešení



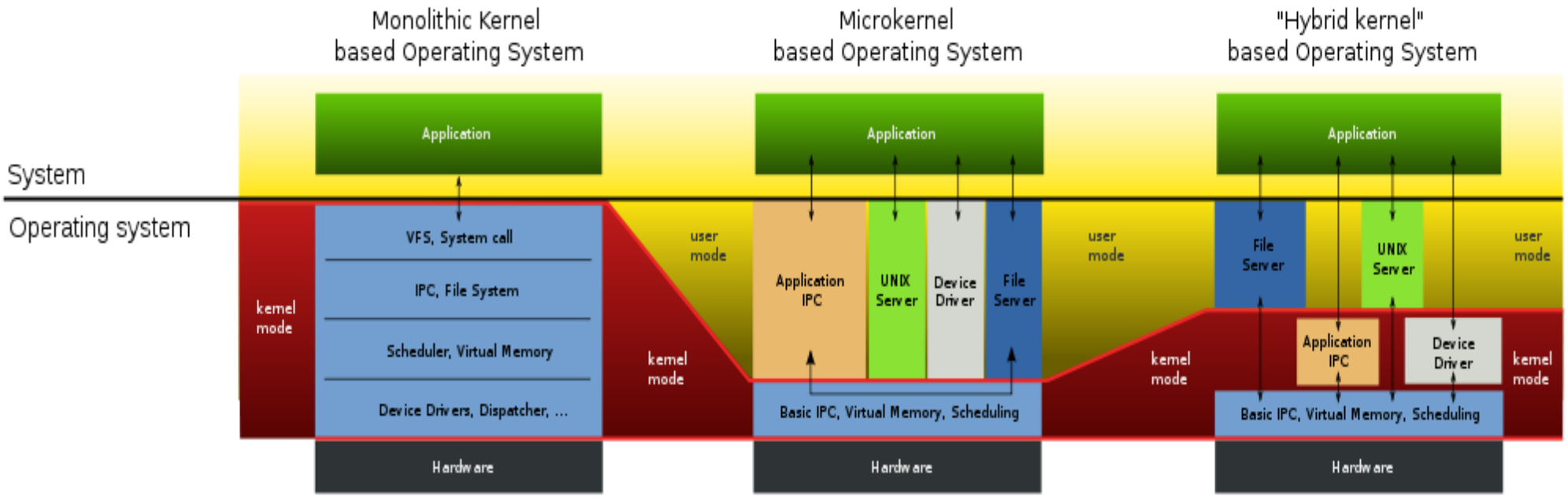
  - Synchronní přerušení se obsluhuje v režii procesu → minimalizace přepínání mezi procesy.
    - Používáno např. v UNIX SVR4
  - Uvnitř JOS používá každý proces samostatný zásobník
  - Kód a data JOS jsou ve sdíleném adresovém prostoru a jsou sdílena všemi procesy
  - Jakou má to řešení nevýhodu?

Procesově orientované JOS, mikrojádro

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



Porovnání mikrojádra a monolitického jádra

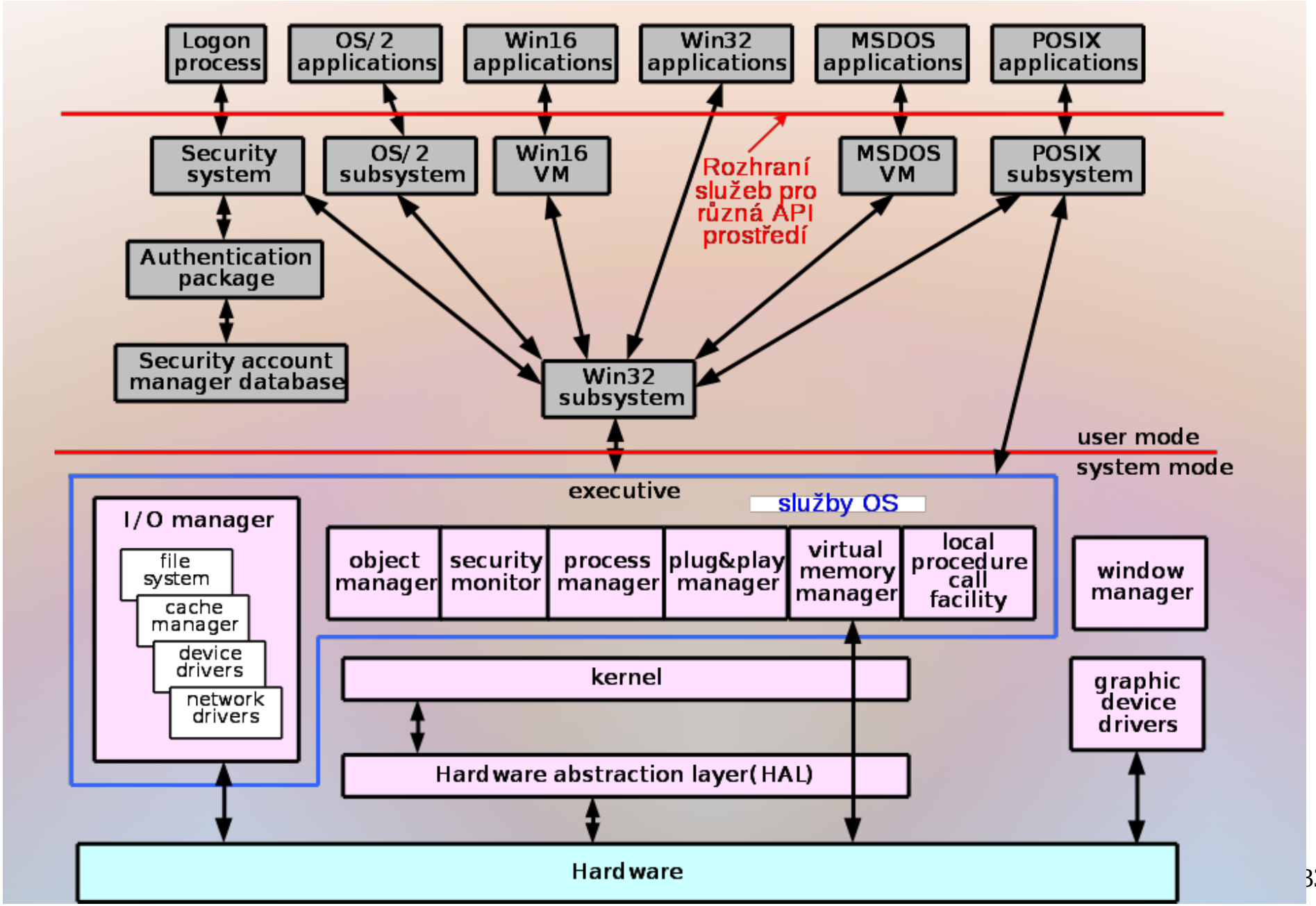


OS s µ-jádrem – výhody

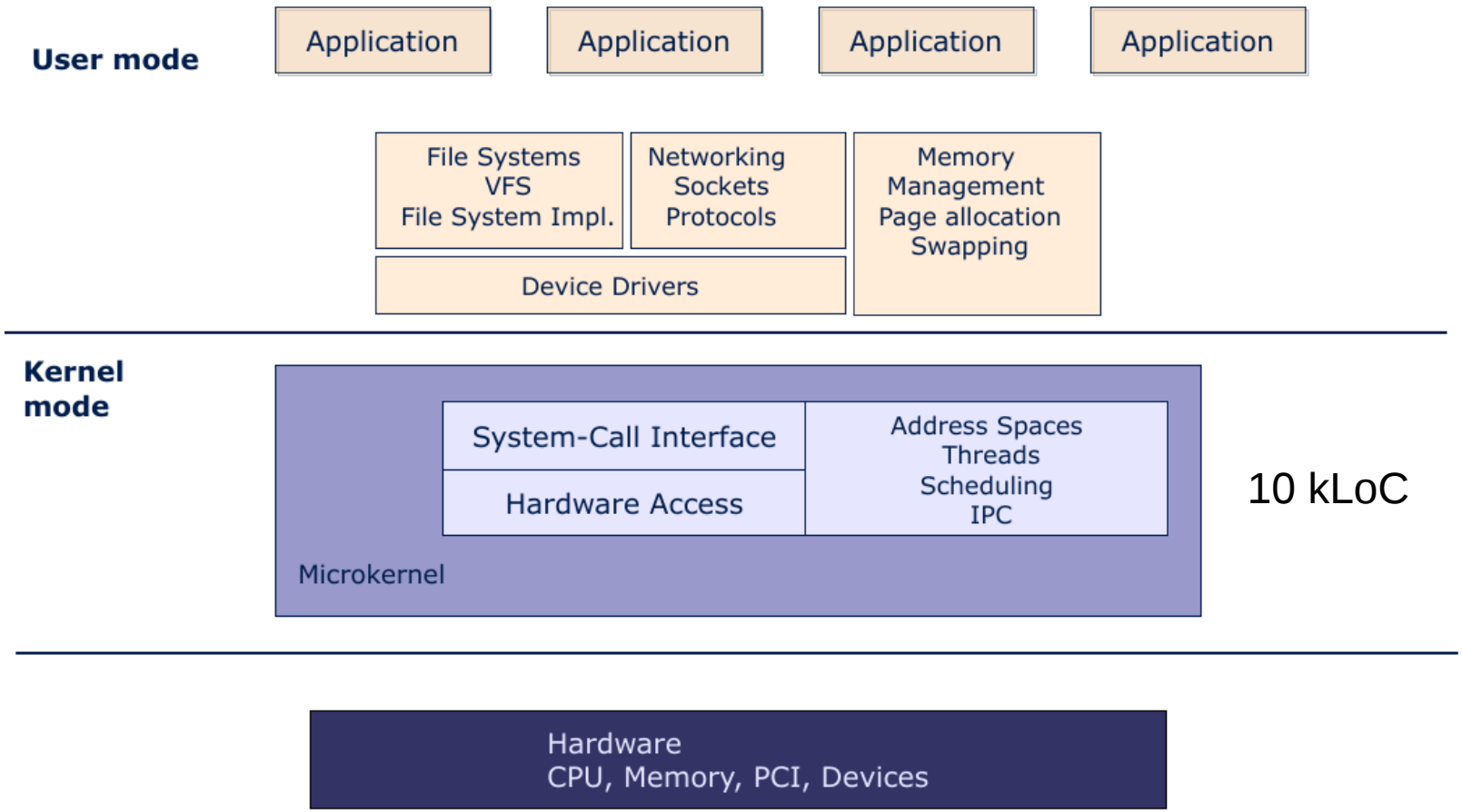
- OS se snáze přenáší na nové hardwarové architektury,
  - µ-jádro je malé
- Vyšší spolehlivost – modulární řešení
  - moduly jsou snáze testovatelné
- Vyšší bezpečnost
  - méně kódu se běží v privilegovaném režimu
- Pružnější, snáze rozšiřitelné řešení
  - snadné doplňování nových služeb a rušení nepotřebných
- 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

Příklad OS s µ-jádrem – Windows XP

(podle marketingových letáků M$)



Skutečný systém s µ-jádrem L4Re

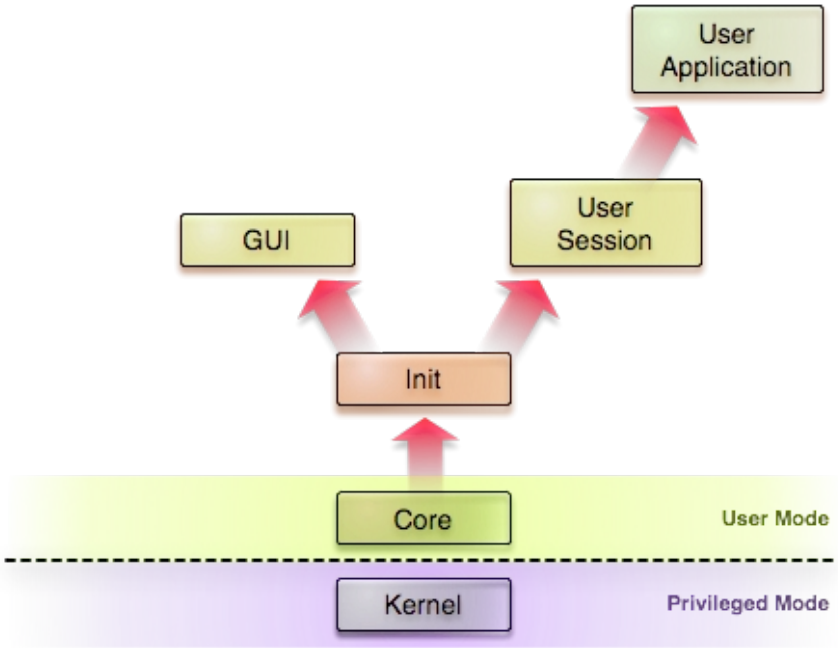


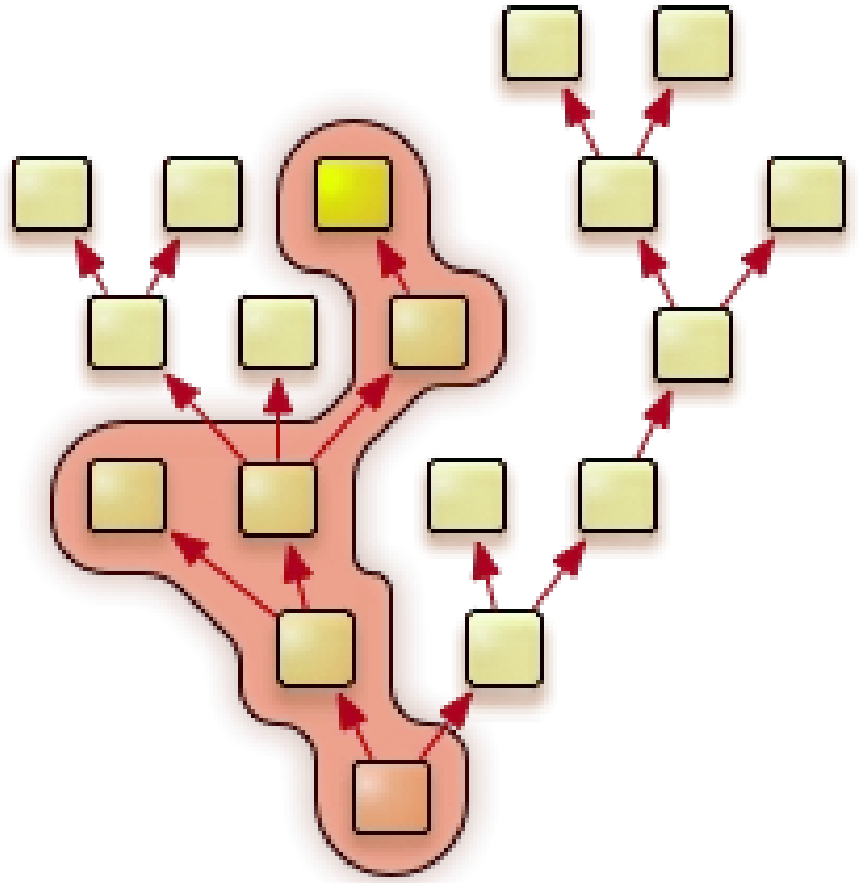
http://os.inf.tu-dresden.de

http://www.kernkonzept.com/

Skutečný systém s µ-jádrem Genode

http://genode.org/





Jeden z cílů: Omezit velikost “Trustued computing base”

Systémová volání µ-jádra NOVA (http://hypervisor.org/)

- 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

- Víc jich není
- PD = protection domain = proces
- EC = execution context
- SC = scheduling context
- PT = portal
- SM = semafor

Vytváření provozní verze OS

- Operační systém je obvykle připraven tak, aby běžel na jisté třídě hardwarových platforem / sestav počítače
- OS musí být konfigurovatelný na konkrétní platformu/sestavu
- Konfigurace
  - Na základě informace týkající se konkrétní požadované konfigurace a konkrétního hardwarového systému vytváří provozní verzi OS odpovídající skutečné skladbě HW prostředků
- Zavaděč systému (*Bootstrap program*)
  - Program uchovávaný v ROM, který umí nalézt jádro (zpravidla na disku), zavést ho do paměti a spustit jeho inicializaci a další provádění
  - Často součást tzv. „firmwaru“ (EFI, BIOS, ...)
- Zavádění systému (*Booting*)
  - Zavedením jádra a předáním řízení na jeho vstupní bod se spustí činnost celého systému

Zavádění systému

- Jádro
  - Inicializace procesoru (assembler)
  - Inicializace dalšího HW (DRAM)
  - Inicializace subsystémů OS (síť, grafika, I/O, …)
  - Komunikace s firmwarem, plug‘n‘play, inicializace ovladačů
  - Spuštění prvního uživatelského procesu
- Uživatelský prostor
  - Řízeno konfiguračními soubory
  - Spuštění dalších procesů (servery, shell, login screen, ...)

OS jsou funkčně složité

OS	Rok	Počet služeb jádra (*system calls*)
Unix	1971	33
Unix	1979	47
SunOS4.1	1989	171
4.3 BSD	1991	136
SunOS4.5	1992	219
SunOS5.6 (Solaris)	1997	190
Linux 2.0	1998	229
WinNT 4.0	1999	**3 443**

- Obrovská složitost vnitřních algoritmů (jádra) OS
  - Počty cyklů CPU spotřebovaných ve WinXP při
    - Zaslání zprávy mezi procesy: 6K – 120 K (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

OS jsou velmi rozsáhlé

- Historie Windows
  - 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	Počet řádků kódu [SLOC]
Windows 3.1	1992	3 mil.
Windows NT 3.5	1993	4 mil.
Windows 95	1995	15 mil.
Windows NT 4.0	1996	16 mil.
Windows 98 SR-2	1999	18 mil.
Windows 2000 SP5	2002	30 mil.
Windows XP SP2	2005	**48 mil.**
Windows 7	2010	**??? (**není známo)

To je dnes vše.

Otázky?