B4B35OSY: Operační systémy

Lekce 2. Systémové volání

Petr Štěpán
stepan@fel.cvut.cz

12. října, 2023

Outline

1 Služby OS

2 Systém NOVA

3 API

4 Procesy

5 Vlákna

6 Bod aktivity

Obsah

1 Služby OS

2 Systém NOVA

3 API

4 Procesy

5 Vlákna

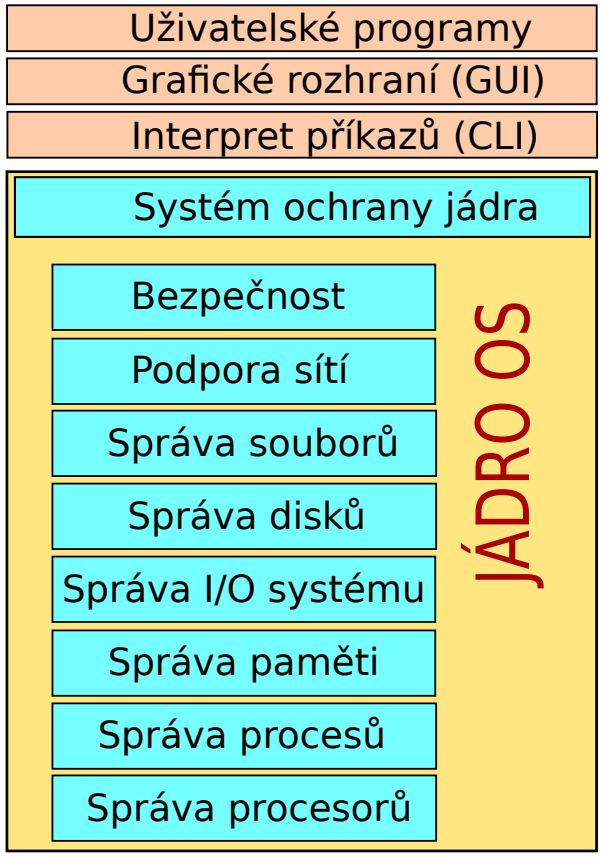
6 Bod aktivity

Co byste měli vědět z minulé hodiny

- Moderní procesor má systémový a uživatelský mód
  - Užvatelský mód je omezený, nemůže provádět všechny instrukce
- Přepnutí z uživatelského do systémového módu je možné jen při přerušení (případně speciální instrukce SYSCALL, SYSENTER)
  - Při obsluze přerušení získá program systémový mód
  - Jádro OS připraví vše k běhu OS a spustí první uživatelský program
  - Po statru je jádro OS přístupné pouze přes přerušení nebo speciální instrukce SYSCALL či SYSENTER

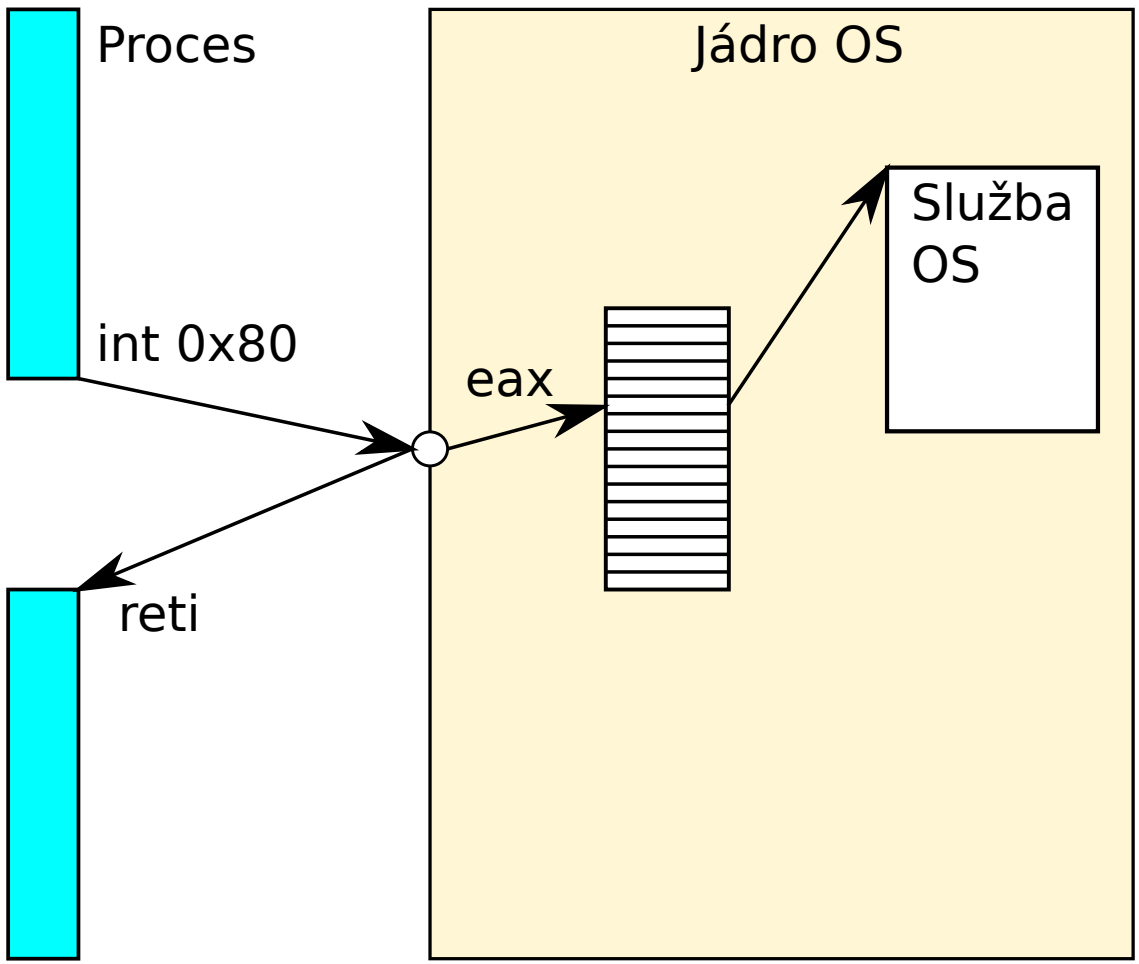
Složky OS

- 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



Ochrana jádra OS

- Uživatel má do jádra OS přístup pouze přes obsluhu přerušení, nebo podobný mechanismus



Služby jádra OS

x86 System Call – 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
```

Kvíz

Proč nastal segmentation fault?

A - Zapomněli jsme ošetřit zásobník programu

B - Zapomněli jsme inicializovat proces, a proto se nemohl vrátit ze systémového volání

C - Zapomněli jsme proces ukončit

D - Zapomněli jsme inicializovat data a tím se použil špatný ukazatel

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

        mov $0xfc,%eax          ; exit system call
        xorl %ebx, %ebx         ; exit status set ebx to 0
        int $0x80               ; call the system
```

Služby jádra OS

x86 System Call v jazyce C/C++

```c
#include <unistd.h>

int main()
{
    asm volatile (
        "int $0x80"
        :
        : "a" (4), "b" (1),
        "c" ("Hello World\n"),
        "d" (12)
        : "memory");
    return 0;
}
```

Kvíz:
Bude program fungovat?

A - nebude - zapomněli jsme ukončit proces

B - nebude - uvnitř jazyka C nelze volat systémové volání

C - bude - v jazyce C se nemusí volat exit

D - bude - exit zavolá funkce _start z knihovny libc

Služby jádra OS

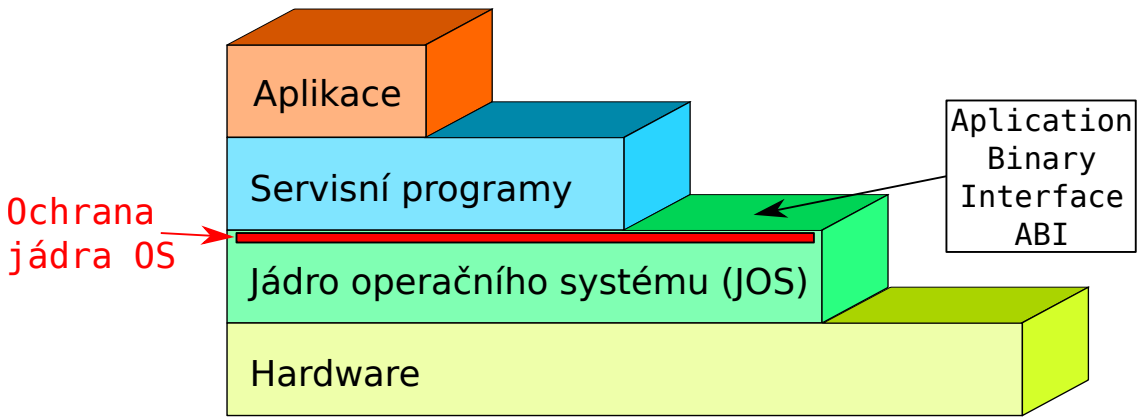
- Služby jádra jsou číslovány
  - Registr eax obsahuje číslo požadované služby
  - Ostatní registry obsahují parametry, nebo odkazy na parametry
  - Problém je přenos dat mezi pamětí jádra a uživatelským prostorem
    - malá data lze přenést v registrech – návratová hodnota funkce
    - velká data – uživatel musí připravit prostor, jádro z/do něj nakopíruje data, předává se pouze adresa (ukazatel)
- Linux system call table

  `http://faculty.nps.edu/cseagle/assembly/sys_call.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, riscv, …
  - 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

ABI Linuxu a ABI NOVA

`int 0x80`

- EIP uloží na zásobník
- FLAGS uloží na zásobník
- Adresu kam skočit bere z tabulky z paměti

`iret`

- EIP načte ze zásobníku
- FLAGS načte ze zásobníku

- návratové EIP uloží do registru, návratové ESP také do registru

`sysenter`

- FLAGS ignoruje, nejsou důležité
- Adresu kam skočit bere z interního registru

`sysexit`

- EIP načte ze registru
- FLAGS ignoruje, nejsou důležité
- ESP načte z registru

Kdy vlastně jádro OS běží?

Jádro OS běží když:

- nastane přerušení nebo vyjímka
- uživatelský program zavolá službu OS

Jindy neběží?

Při spuštění počítače připraví jádro OS prostředí pro běh procesů a spustí první proces.
Pak už jádro OS jen čeká na přerušení, vyjímky a systémová volání.

Rozdíl mezi root a jádrem OS

Root (administrátor ve Windows) je sice správce systému, ale z hlediska OS se jedná jen o obyčejné procesy v uživatelském prostoru, které mají více práv, ale samy nemohou přistupovat k HW.
I root proces musí využívat systémové služby k práci s HW.

Jádro OS není proces, jedná se o mnoho funkcí spouštěných přerušeními, vyjímkami a systémovými voláními uživatelských procesů.
Jádro OS může dělat úplně vše, co může počítač vykonat.

Obsah

1 Služby OS

2 Systém NOVA

3 API

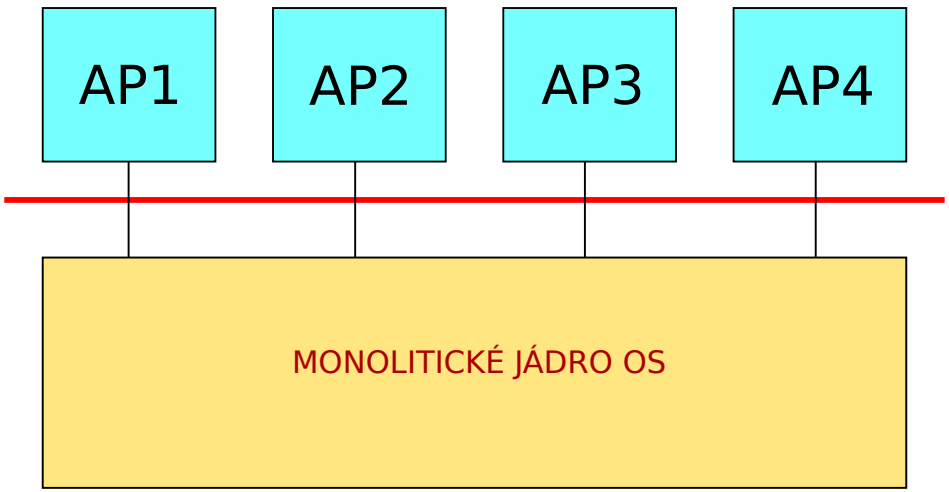
4 Procesy

5 Vlákna

6 Bod aktivity

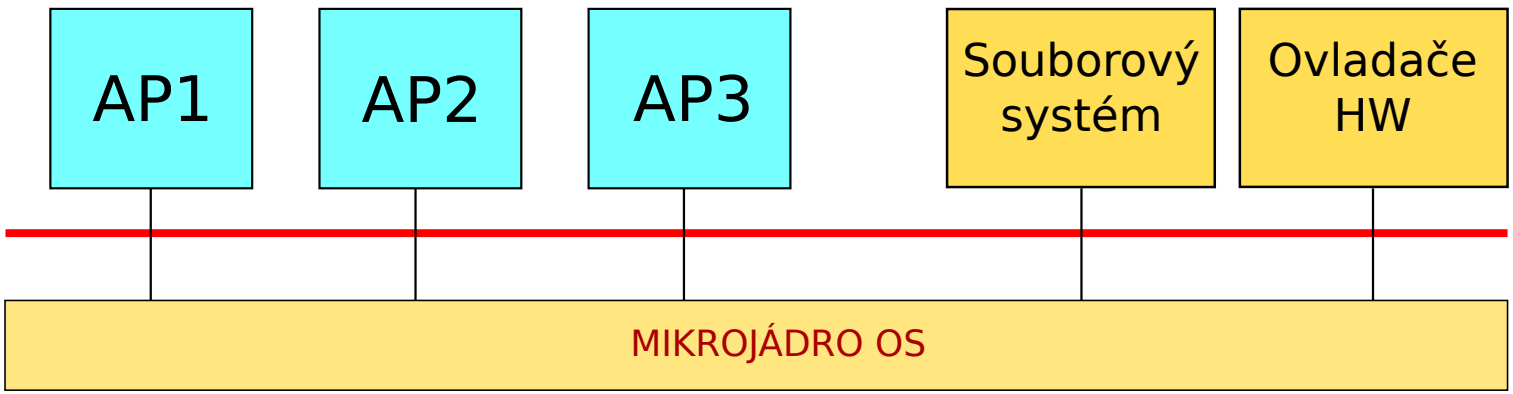
Achitektura jádra OS

- 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

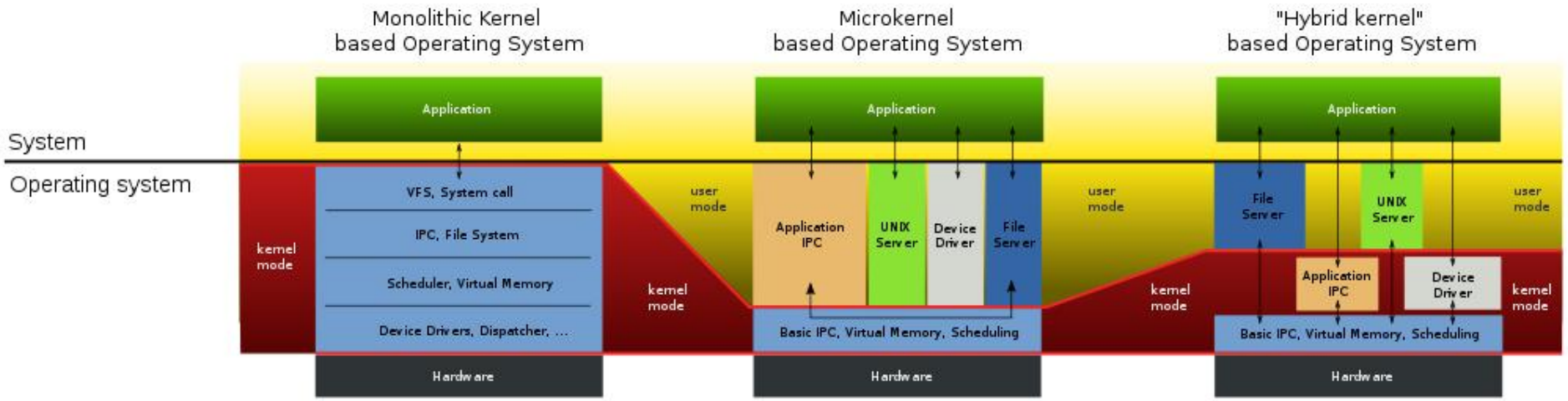


Procesově orientované jádro OS – mikrojádro

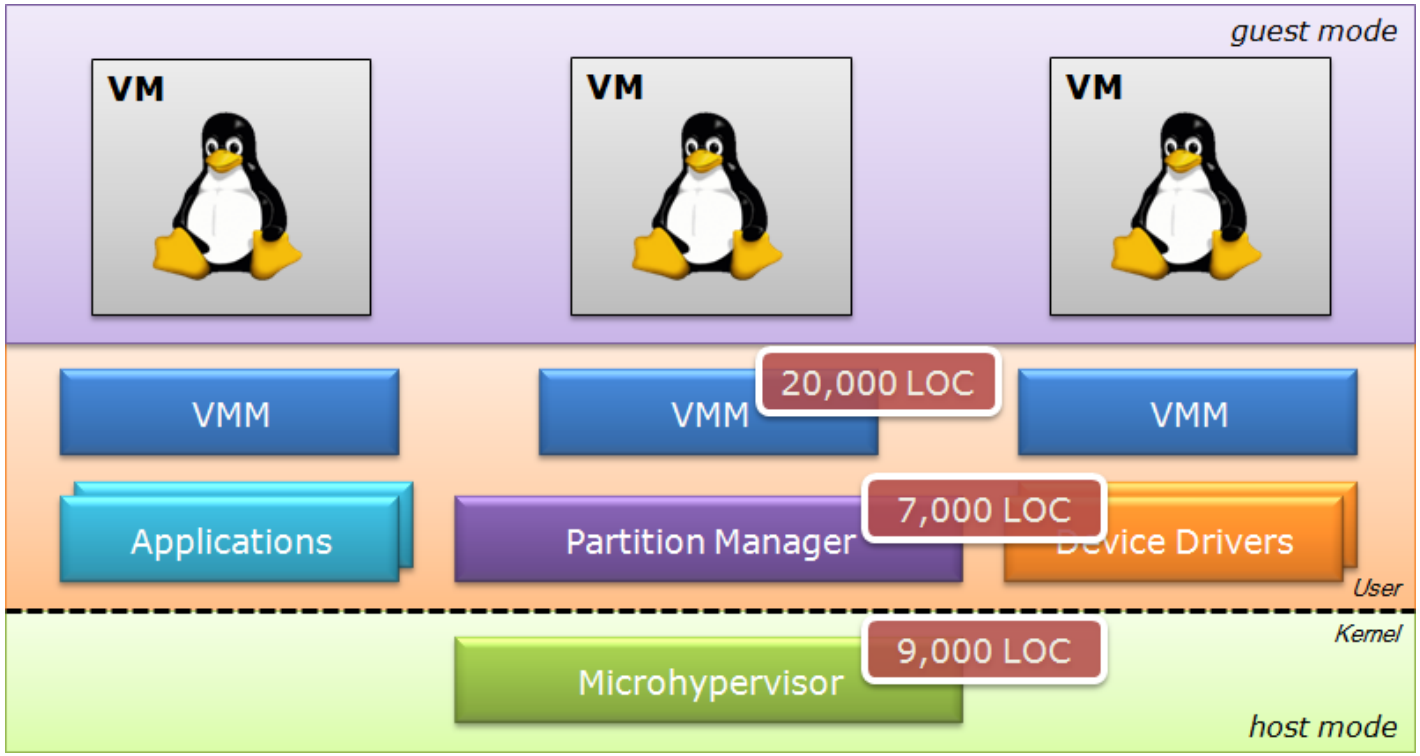
- OS je soustavou systémových procesů
- Funkcí jádra je tyto procesy separovat, ale umožnit přitom jejich kooperaci
  - Minimum funkcí je potřeba dělat v privilegovaném režimu
  - Jádro pouze ústředna pro přepojování zpráv
  - Řešení snadno implementovatelné i na multiprocesorech
- Malé jádro $\Rightarrow$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



Porovnání JOS



NOVA microhypervisor



- Systém začal jako experimentální systém na TU Dresden (< 2012) a Intel Labs (≥ 2012).
- `http://hypervisor.org/`, x86, GPL.
- My budeme využívat pouze část (2 kLoC) originálního jádra.

Nano úvod do C++

Systém NOVA je napsán v C++.

```
class A {
  public:
    enum B {Ex, Ey};
    int Ni;
    static int Si;
    A(int z): Ni(z) {}
    int f(int);
    static int Sf(int);
};
```

Znak : : je použit pro definici a pro odkazy na statické prvky třídy A

```
int A::f(int x) {
  return  -1*x;
}
```

```
int A::Sf(int x) {
  return  -2*x;
}
// globalni definice  promenne
int A::Si;
```

Znak : : je použit i při použití vnitřních struktur např. enum B

```
int m = A::Sf(A::Ex);
int n = A::Si;
```

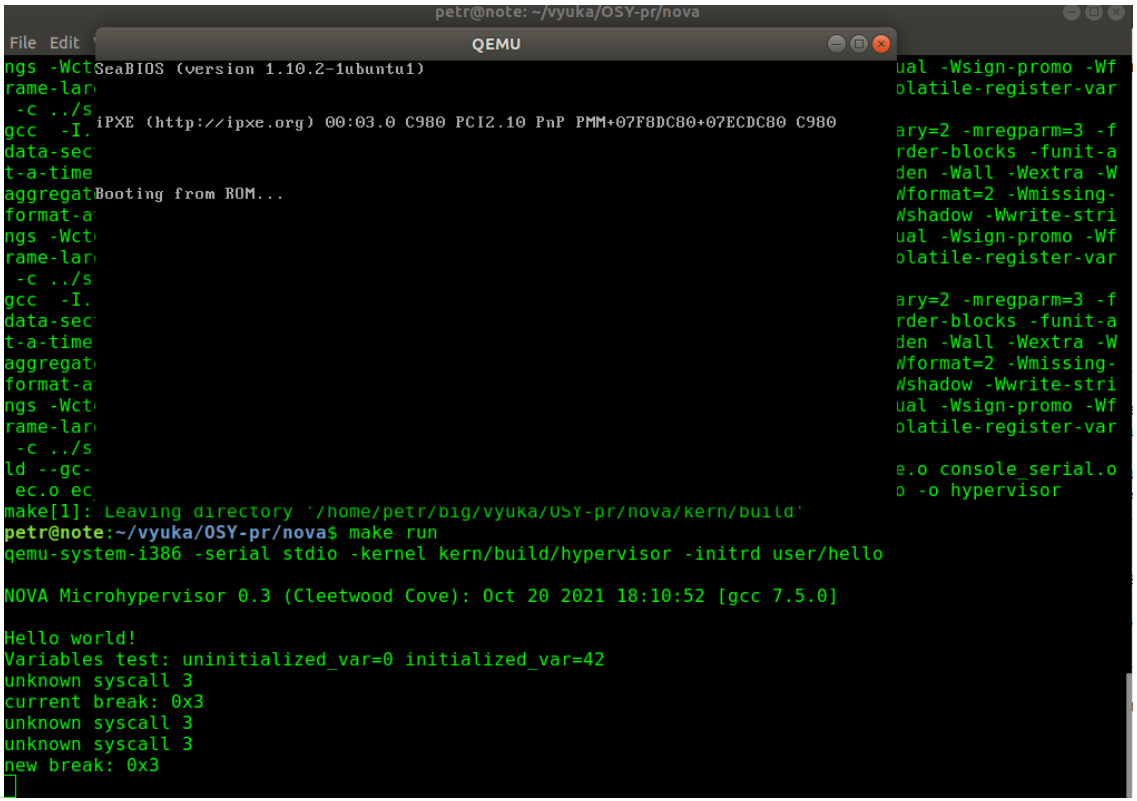
Znak . je použit pro přístup k prvkům instance třídy:

```
A a(10);
int m = a.f(A::Ey);
int n = a.Ni;
```

Začínáme

```
unzip nova.zip
cd nova
make # Compile everything
make run # Run it in Qemu emulator
```

Na obrazovce uvidíte asi toto:



Co se vlastně stalo?

- Spustila se emulace i386 počítače
- qemu-system-i386 -serial stdio -kernel kern/build/hypervisor -initrd user/hello
  - Výstup sériové linky vidíte na standardním výstupu terminálu
    - Výstup na sériový port je to první, co operační systém ovládá
    - Jsou zde ladicí výpisy, které jsou nezbytné pro ladění jádra OS
  - V okně vidíte to, co by bylo vidět na obrazovce simulovaného počítače
    - Tedy vlastně jen start BIOSU a bootování systému
  - Systém je nastartován v módu multi boot, tedy OS získá adresu uživatelského programu `user/hello`, který jako první proces spustí.
- Co je v nova.tgz?
  - `user/` – program hello, který je spuštěn jako první proces
  - `kern/` – naše oříznuté jádro NOVA
    - vy budete pracovat hlavně s `kern/src/ec_syscall.cc`
    - ale prohlédněte si celé jádro, hlavně havičkové soubory v `kern/include`

Systémové volání NOVA

Co dělá uživatel?

- Prohlédněte si `user/hello.c`

```
unsigned syscall2(unsigned w0,
                  unsigned w1){
  asm volatile (
    "   mov %%esp, %%ecx;"
    "   mov $1f, %%edx  ;"
    "   sysenter        ;"
    "1:                 ;"
    : "+a" (w0) : "S" (w1)
    : "ecx", "edx", "memory");
  return w0;
}
```

Místo zásobníku využívá registry:

- ecx – obsahuje ukazatel na zásobník po návratu ze systémového volání
- edx – obsahuje adresu, kam se vrátit po ukončení systémového volání ($1f je návěstí 1:)
- eax – číslo systémového volání
- esi – první argument (S)
- edi – druhý argument (D)

Systémové volání NOVA

Co dělá jádro?

- uloží všechny registry na zásobník viz. `kern/src/entry.S`
- zjistí typ systémového volání podle registru eax viz. `kern/src/ec_syscall.cc`

```
void Ec::syscall_handler (uint8 a) {
  Sys_regs * r = current->sys_regs();
  Syscall_numbers number =
    static_cast<Syscall_numbers> (a);

  switch (number) {
    case sys_print: {
      char *data=reinterpret_cast<char*>(r->esi);
      unsigned len = r->edi;
      for (unsigned i = 0; i < len; i++)
         printf("%c", data[i]);
      break; }
    case sys_sum: {
      int first_number = r->esi;
      int second_number = r->edi;
      r->eax = first_number + second_number;
      break; }
    default:
      printf ("unknown syscall %d\n", number);
      break;
    };
    ret_user_sysexit();
}
```

Kvíz

Jak to, že jste zatím ve svých programech nepoužívali instrukci `int 0x80` ani `syscall/sysenter`?

A - Vaše programy nepoužívaly systémová volání.

B - Vaše programy přímo přistupovaly k HW.

C - Vaše programy využívaly funkce, které použili `int 0x80` nebo `syscall/sysenter`.

D - Windows nepodporuje systémová volání

Obsah

1 Služby OS

2 Systém NOVA

3 API

4 Procesy

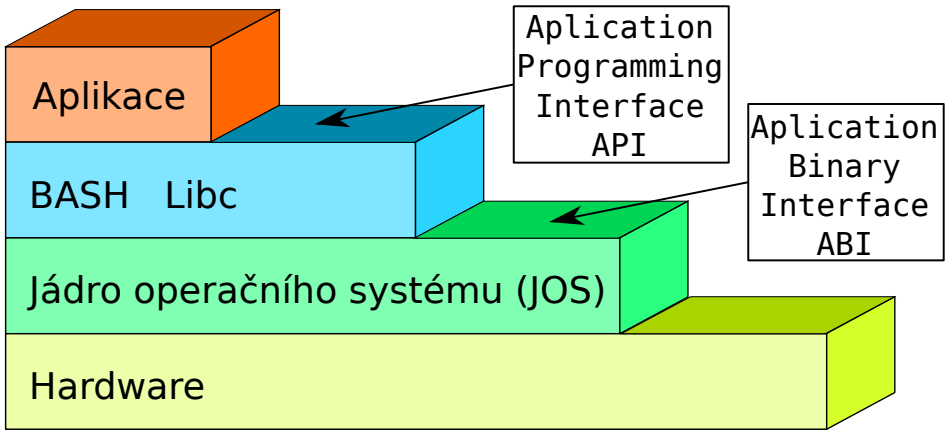
5 Vlákna

6 Bod aktivity

Application Programming Interface – API

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)



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

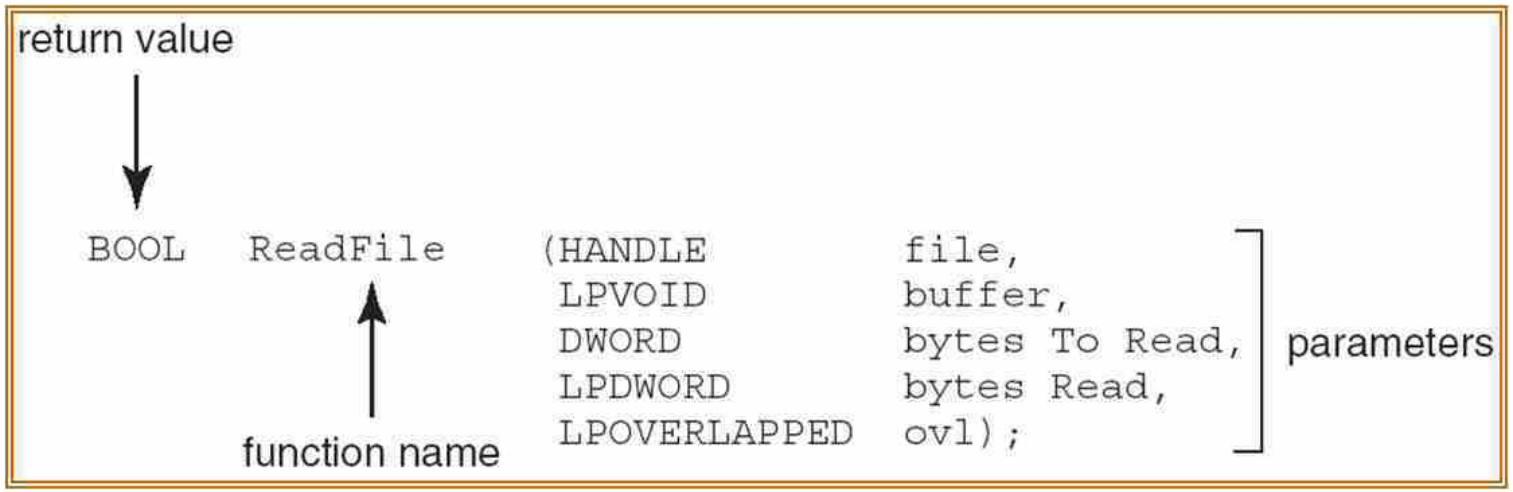
- Práce se soubory
  - open, close, read, write, lseek
- Správa souborů a adresářů
  - mkdir, rmdir, link, unlink, mount, umount, chdir, chmod, stat
- Správa procesů
  - fork, waitpid, execve, exit, kill, signal

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 – nyní obsahuje i 64 bitové rozhraní v rámci Win32
- Nová windows 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.

Obsah

1 Služby OS

2 Systém NOVA

3 API

4 Procesy

5 Vlákna

6 Bod aktivity

Proces

- Výpočetní proces (job, task) – spuštěný program
- Proces je identifikovatelný jednoznačným číslem v každém okamžiku své existence
  - PID – Process IDentifier
- Co tvoří proces:
  - Obsahy registrů procesoru (čítač instrukcí, ukazatel zásobníku, příznaky FLAGS, uživatelské registry, FPU registry)
  - Otevřené soubory
  - Použitá paměť:
    - Zásobník – .stack
    - Data – .data
    - Program – .text
- V systémech podporujících vlákna bývá proces chápán jako obal či hostitel svých vláken

Proces – požadavky na OS

- Umožňovat procesům vytváření a spouštění dalších procesů
- Prokládat – „paralelizovat“ vykonávání jednotlivých procesů s cílem maximálního využití procesoru/ů
  - Minimalizovat dobu odezvy procesu prokládáním běhů procesů
- Přidělovat procesům požadované systémové prostředky
  - Soubory, V/V zařízení, synchronizační prostředky
- Umožňovat vzájemnou komunikaci mezi procesy
- Poskytovat aplikačním procesům funkčně bohaté, bezpečné a konzistentní rozhraní k systémovým službám
  - Systémová volání
- Ukončit proces a uvolnit používané systémové prostředky

Vznik procesu

- Rodičovský proces vytváří procesy-potomky
  - pomocí služby OS. Potomci mohou vystupovat v roli rodičů a vytvářet další potomky, …
  - vzniká tak strom procesů
- Sdílení zdrojů mezi rodiči a potomky:
  - rodič a potomek mohou sdílet všechny zdroje původně vlastněné rodičem (obvyklá situace v POSIXových systémech)
  - potomek může sdílet s rodičem podmnožinu zdrojů rodičem k tomu účelu vyčleněnou
  - potomek a rodič jsou plně samostatné procesy, nesdílí žádný zdroj
- Souběh mezi rodiči a potomky:
  - Možnost 1: rodič čeká na dokončení potomka
  - Možnost 2: rodič a potomek mohou běžet souběžně
- V POSIXových systémech je každý proces potomkem jiného procesu
  - Výjimka: proces č. 1 (init, systemd, …) vytvořen při spuštění systému
  - Spustí řadu procesů a skriptů (rc), ty inicializují celý systém a vytvoří démony (procesy běžící na pozadí bez přístupu na terminál) $\sim$ service ve Win32
  - Init spustí také
    - textové terminály procesy `getty`, které čekají na uživatele $\rightarrow$ login $\rightarrow$ uživatelův shell
    - grafické terminály *display manager* a *greeter* (grafický login)

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 dat 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

Příklad fork - kvíz

Příklad A

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int f=fork(), ff=-1;
  ff=fork();

  printf("Hello %i %i\n", f, ff);
  return 0;
}
```

Program A vytiskne Hello?

A - 1x

B - 2x

C - 3x

D - 4x

Příklad fork

Příklad B

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int f=fork(), ff=-1;
  if (f==0) {
    ff=fork();
  }
  printf("Hello %i %i\n", f, ff);
  return 0;
}
```

Program B vytiskne Hello?

A - 1x

B - 2x

C - 3x

D - 4x