B4B35OSY: Operační systémy

Lekce 2. Systémové volání

Petr Štěpán
stepan@fel.cvut.cz

6. října, 2022

Outline

1 Opakování

2 Služby OS

3 Systém NOVA

4 API

5 Procesy

6 Vlákna

Obsah

1 Opakování

2 Služby OS

3 Systém NOVA

4 API

5 Procesy

6 Vlákna

Kvíz

Kdy běží kód z jádra operačního systému?

A - Od začátku spuštění počítače neustále, vlastně simuluje běh uživatelských procesů

B - Jedno CPU neustále vykonává kód jádra OS, ostatní CPU vykonávají uživatelské procesy

C - Pouze při spuštění počítače a pak při přerušení nebo obsluze systémového volání

D - Jádro OS neběží vůbec, beží jen různé uživatelské programy

Kdy vlastně OS běží?

Jádro OS běží když:

- nastane přerušení nebo výjimka
- uživatelský program zavolá službu OS

Jindy neběží?

Ještě na začátku spuštění počítače, připraví vše pro běh procesů a spustí první proces. Pak už jen čeká na přerušení, výjimky a systémová volání.

Nastavení OS pomáhají uživatelské programy s právy správce počítače - démoni, volají služby jádra.

Kvíz

Jak se liší jádro OS a root?

A - Neliší se, je to vlastně totéž

B - Jádro OS může vše, root má omezené pravomoci jako jiné procesy

C - Jádro OS má omezené pravomoci, root může vše

D - Jádro OS i root mohou provést veškeré strojové instrukce bez omezení

Rozdíl mezi root a jádrem OS

Root je sice správce systému, ale jedná se jen o obyčejné procesy v uživatelském procesu, které mají více práv, ale nemohou přistupovat k HW. I root proces musí využívat systémové služby ke komunikaci s HW.

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

Obsah

1 Opakování

2 Služby OS

3 Systém NOVA

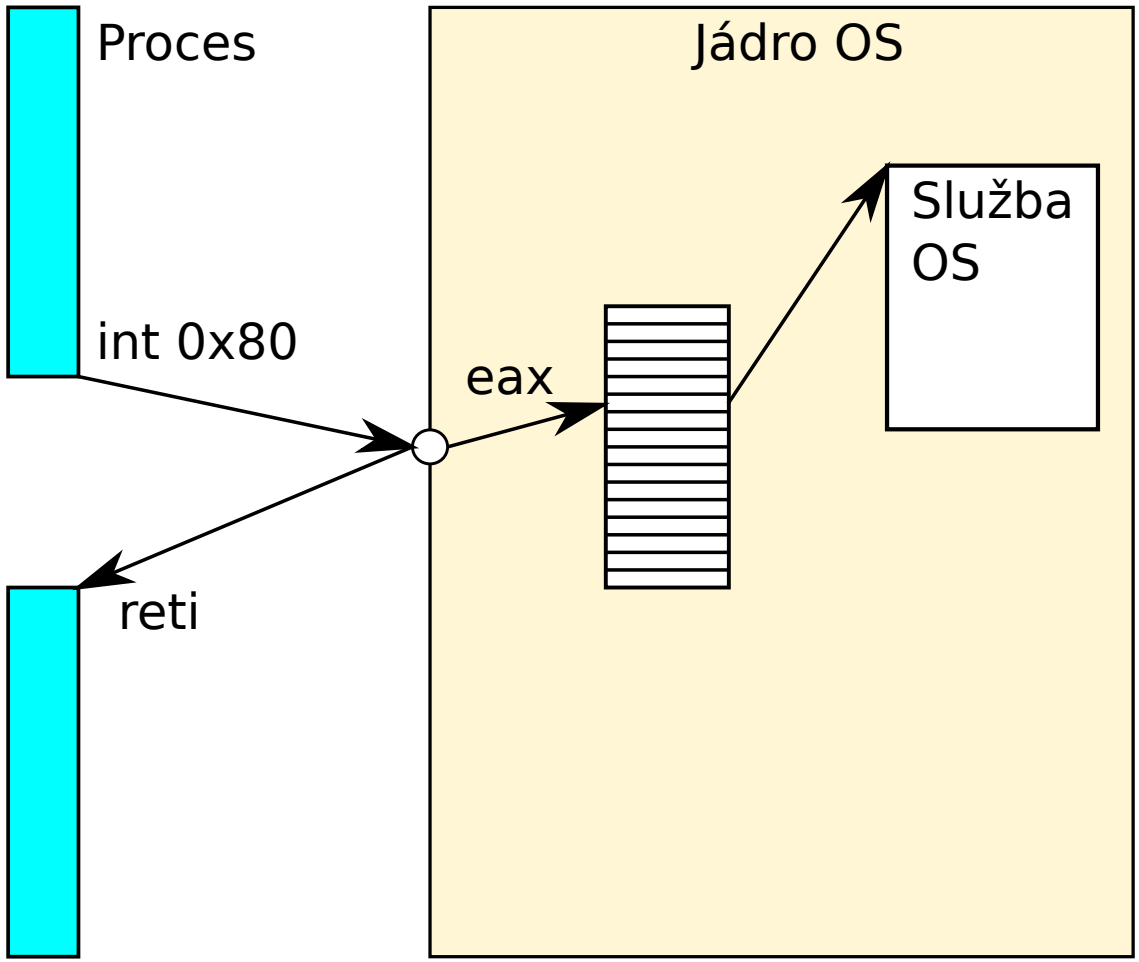
4 API

5 Procesy

6 Vlákna

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 proces ukončit

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

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

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, …
  - 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

`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

- EIP uloží do registru, 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

Obsah

1 Opakování

2 Služby OS

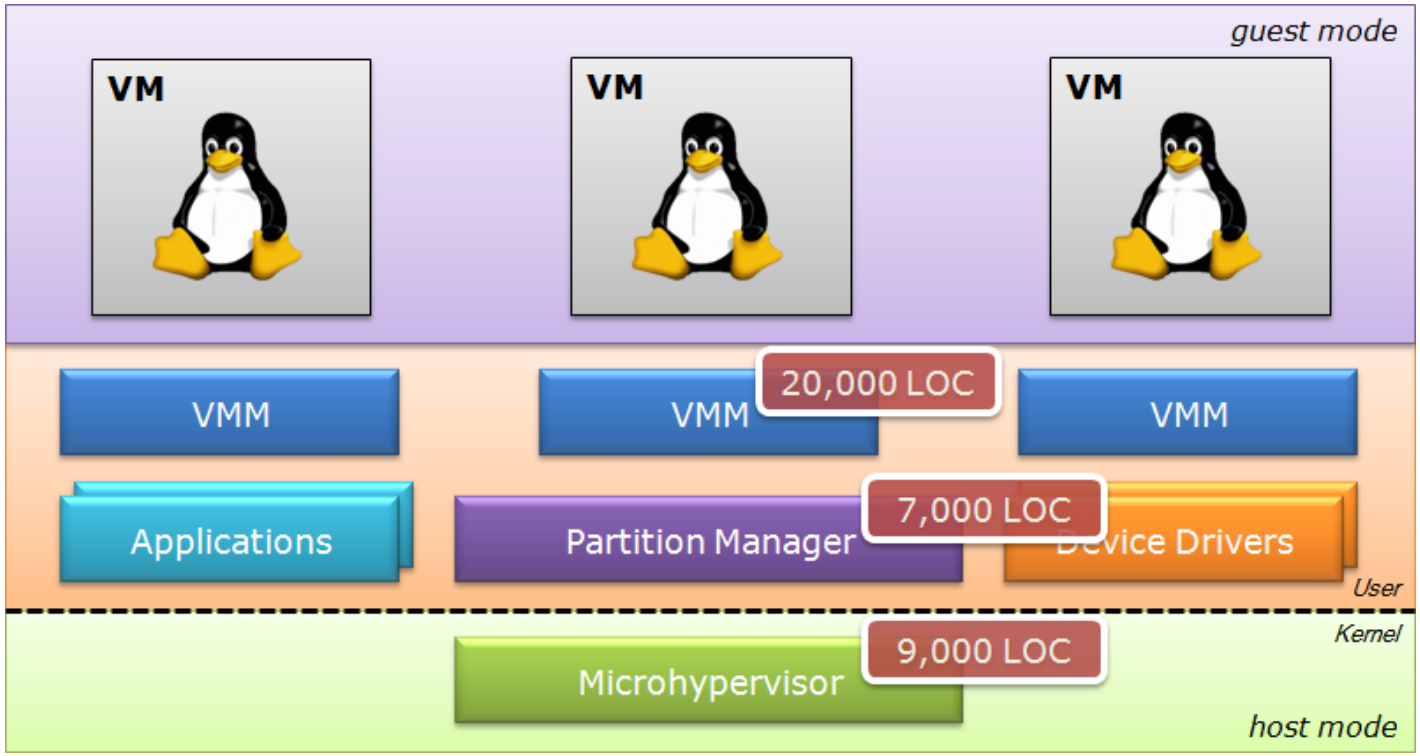
3 Systém NOVA

4 API

5 Procesy

6 Vlákna

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;
```

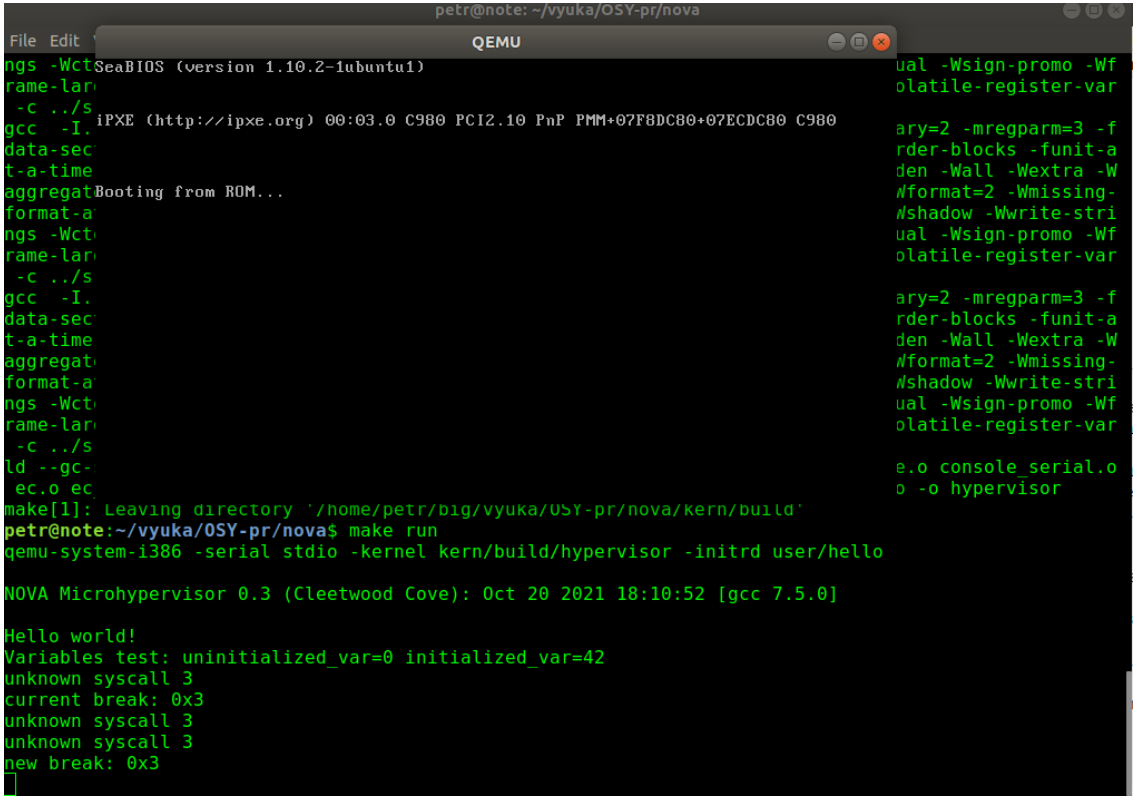
Znak . je použit pro přístup k prvkům instance třídy:

```
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 Opakování

2 Služby OS

3 Systém NOVA

4 API

5 Procesy

6 Vlákna

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

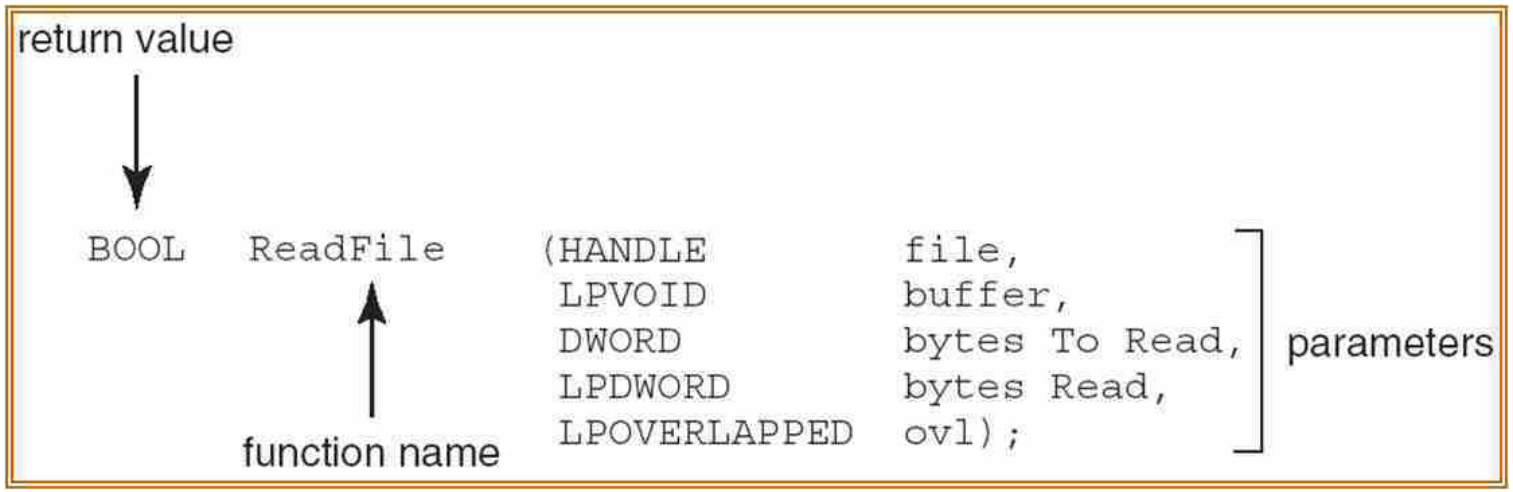
- Práce se soubory
  - open, close, read, write, lseek
- Správa souborů a adresářů
  - mkdir, rmdir, link, unlink, mount, umount, chdir, chmod, stat
- Správa procesů
  - fork, waitpid, execve, exit, kill, signal

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 Opakování

2 Služby OS

3 Systém NOVA

4 API

5 Procesy

6 Vlákna

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) service ve Win32
  - Init spustí také
    - textové terminály proces `getty`, který čeká na uživatele login 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

```
#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

```
#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

Služby OS - wait

`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

Obě funkce:

- přijmou 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 potomka

Kvíz – Volání Wait

Co se stane, když rodič nezavolá systémové volání wait?

A - Proces nemůže skončit a neustále běží

B - Proces skončí, ale program a všechna data zůstávají v paměti

C - Proces zůstane v počítači jako živá mrtvola – zombie

D - Nic se nestane, proces je vymazán ze systému

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`

Ukončení procesu

- Proces provede poslední instrukci programu a žádá OS o ukončení voláním služby `exit(status)`
  - Stavová data procesu-potomka (status) se musí předat procesu-rodiči, který typicky čeká na potomka pomocí `wait()`
  - Zdroje (paměť, otevřené soubory) končícího procesu jádro samo uvolní
- Proces může skončit také:
  - přílišným nárokem na paměť (požadované množství paměti není a nebude k dispozici) – *OOM killer* v Linux
  - běžící kód vygeneruje výjimku CPU, kterou jádro neumí vyřešit:
    - aritmetickou chybou (dělení nulou, arcsin(2), ...)
    - pokusem o narušením ochrany paměti („zabloudění“ programu)
    - pokusem o provedení nedovolené (privilegované) operace (zakázaný přístup k hardwarovému prostředku)
    - …
  - žádostí rodičovského procesu (v POSIXu signál)
    - Může tak docházet ke kaskádnímu ukončování procesů
    - V POSIXu lze proces „odpojit“ od rodiče – démon
  - a v mnoha dalších chybových situacích

Obsah

1 Opakování

2 Služby OS

3 Systém NOVA

4 API

5 Procesy

6 Vlákna

Kvíz – Vlákna

Co jsou vlákna?

A - Vlákna mohou vykonávat různé funkce v rámci jednoho procesu s vlastním zásobníkem

B - Vlákna musí vykonávat stejnou funkci, ale běží paralelně v rámci jednoho procesu

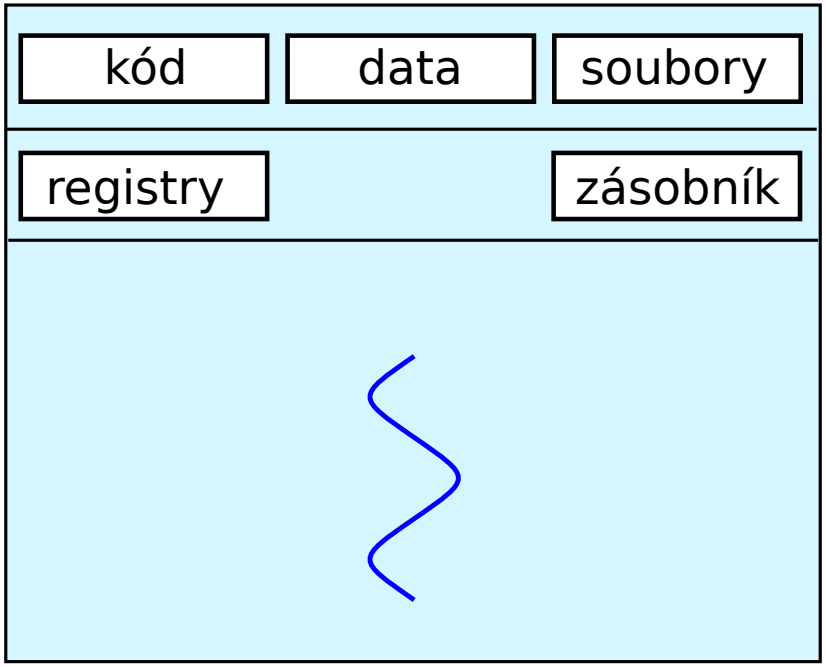
C - Jedná se vlastně o paralelní běh různých procesů se stejným programem

D - Vlákna nemohou běžet na různých jádrech CPU

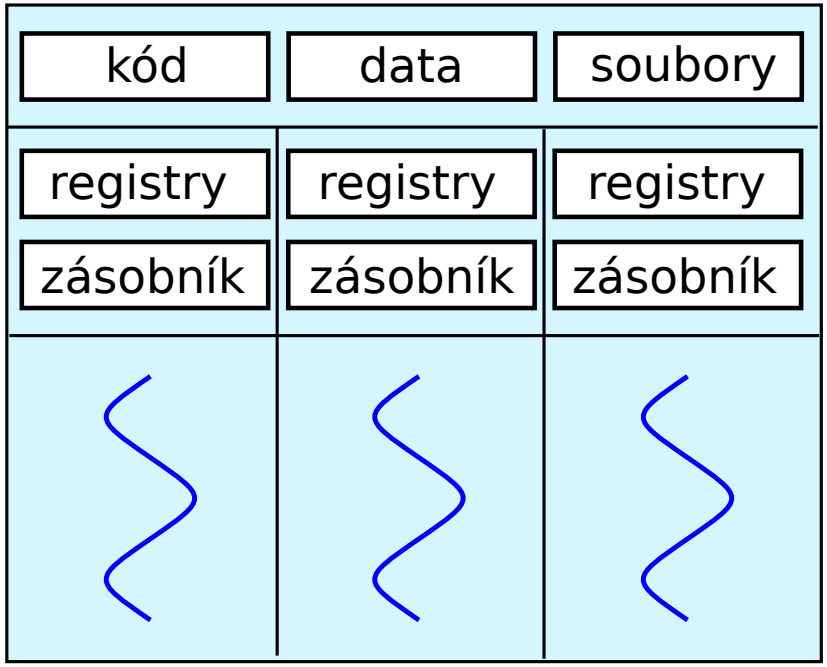
Program, proces, vlákno

- Program:
  - je soubor (např. na disku) přesně definovaného formátu obsahující
    - instrukce,
    - data
    - údaje potřebné k zavedení do paměti a inicializaci procesu
- Proces:
  - je spuštěný program – objekt jádra operačního systému provádějící výpočet podle programu
  - je charakterizovaný svým paměťovým prostorem a kontextem (prostor v RAM se přiděluje procesům – nikoli programům!)
  - může vlastnit (kontext obsahuje položky pro) otevřené soubory, I/O zařízení a komunikační kanály, které vedou k jiným procesům, …
  - obsahuje jedno či více vláken
- Vlákno:
  - je sekvence instrukcí vykonávaných procesorem
  - sdílí s ostatními vlákny procesu paměťový prostor a další atributy procesu (soubory, …)
  - má vlastní hodnoty registrů CPU

Procesy a vlákna



jednovláknový proces



vícevláknový proces

Vlákno – thread

Vlákno – thread

- Objekt vytvářený v rámci procesu a viditelný uvnitř procesu
- Tradiční proces je proces tvořený jediným vláknem
- Vlákna podléhají plánování a přiděluje se jim strojový čas i procesory
- Vlákno se nachází ve stavech: běží, připravené, čekající, ukončené
- Když vlákno neběží, je kontext vlákna uložený v TCB (Thread Control Block) – analogie PCB
  - Linux má stejnou strukturu `task_struct` pro procesy i pro vlákna
    - na informace společné s procesem (např. správa paměti) se vlákno odkazuje k procesu
  - Každý proces je tedy vlastně alespoň jedno vlákno
- Vlákno může přistupovat k globálním proměnným a k ostatním zdrojům svého procesu, data jsou sdílena všemi vlákny stejného procesu
  - Změnu obsahu globálních proměnných procesu vidí všechna ostatní vlákna téhož procesu
  - Soubor otevřený jedním vláknem je viditelný pro všechna ostatní vlákna téhož procesu

Proces

Co patří komu?

kód programu	proces
lokální proměnné	vlákno
globální proměnné	proces
otevřené soubory	proces
zásobník	vlákno
správa paměti	proces
čítač instrukcí	vlákno
registry CPU	vlákno
plánovací stav	vlákno
uživatelská práva	proces

Účel vláken

- Přednosti
  - Vlákno se vytvoří i ukončí rychleji než proces
  - Přepínání mezi vlákny je rychlejší než mezi procesy
  - Dosáhne se lepší strukturalizace programu
- Příklady
  - Souborový server v LAN
    - Musí vyřizovat během krátké doby několik požadavků na soubory
    - Pro vyřízení každého požadavku se zřídí samostatné vlákno
  - Symetrický multiprocesor
    - na různých procesorech mohou běžet vlákna souběžně
  - Menu vypisované souběžně se zpracováním prováděným jiným vláknem
    - Překreslování obrazovky souběžně se zpracováním dat
  - Paralelizace algoritmu v multiprocesoru

Možnosti implementace vláken

Vlákna na uživatelské úrovni

- OS zná jenom procesy
- Vlákna vytváří uživatelská knihovna, která střídavě mění spuštěná vlákna procesu
- Pokud jedno vlákno na něco čeká, ostatní vlákna nemohou běžet, protože jádro OS označí jako čekající celý proces
- Pouze staré systémy, nebo jednoduché OS, kde nejsou vlákna potřeba

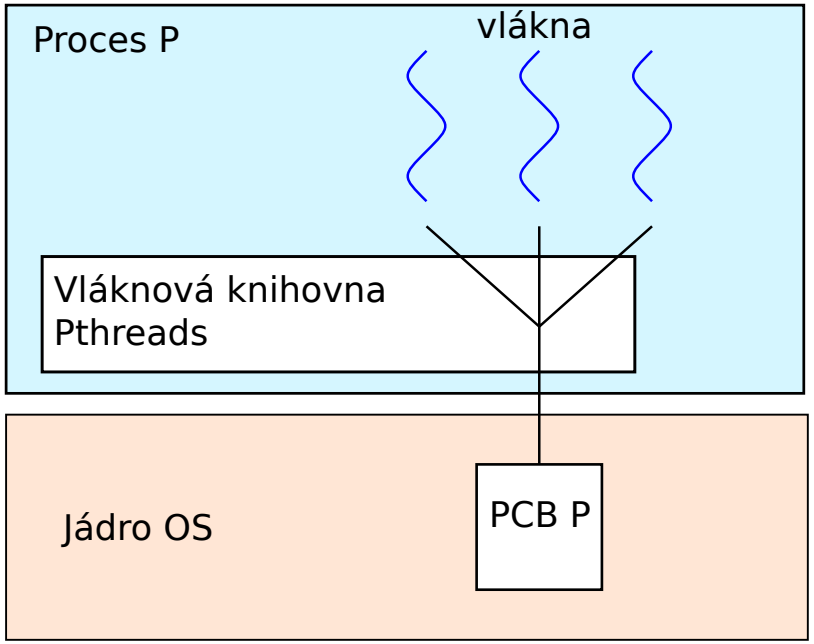
Vlákna na úrovni jádra OS

- Procesy a vlákna jsou plně podporované v jádře
- Moderní operační systémy (Windows, Linux, OSX, Android)
- Vlákno je jednotka plánování činnosti systému

Vlákna na uživatelské úrovni

Problémy vláken na uživatelské úrovni

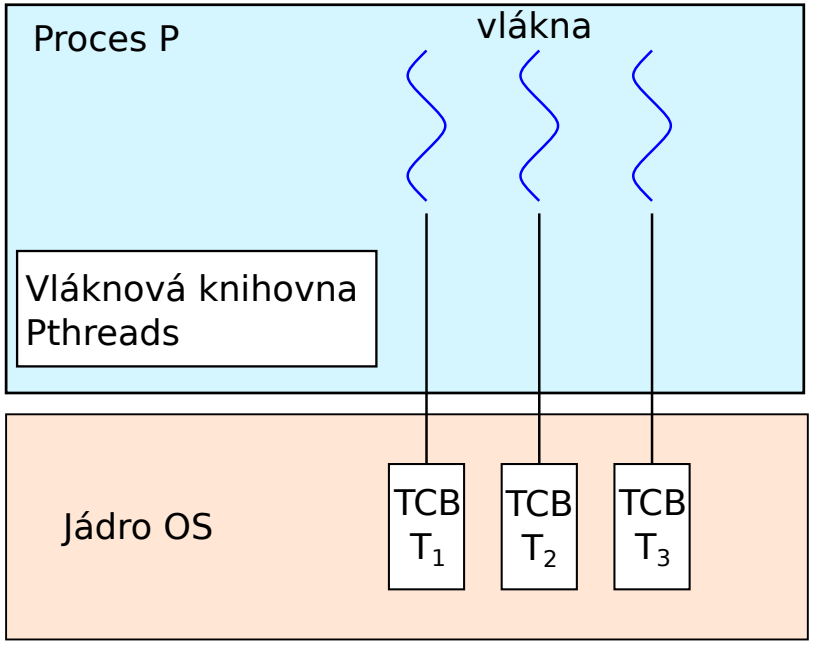
- Jedno vlákno čeká, všechny vlákna čekají
- Proces čeká, ale stav vlákna je běžící
- Dvě vlákna nemohou běžet skutečně paralelně, i když systém obsahuje více CPU



Vlákna v jádře OS

Kernel-Level Threads (KLT)

- Veškerá správa vláken je realizována OS
- Každé vlákno v uživatelském prostoru je zobrazeno na vlákno v jádře (model 1:1)
- JOS vytváří, plánuje a ruší vlákna
- Jádro může plánovat vlákna na různé CPU, skutečný multiprocessing
- Nyní všechny moderní OS: Windows, OSX, Linux, Android



Vlákna v jádře OS

- Výhody:
  - Volání systému neblokuje ostatní vlákna téhož procesu
  - Jeden proces může využít více procesorů
  - Skutečný paralelismus uvnitř jednoho procesu – každé vlákno běží na jiném procesoru
  - Tvorba, rušení a přepínání mezi vlákny je levnější než přepínání mezi procesy
  - Netřeba dělat cokoliv s přidělenou pamětí
- Nevýhody:
  - Systémová správa je režijně nákladnější než u čistě uživatelských vláken
  - Klasické plánování není ”spravedlivé”: Dostává-li vlákno své časové kvantum, pak procesy s více vlákny dostávají více času
    - Moderní OS ale používají spravedlivé plánování

Pthreads

- Pthreads je knihovna poskytující API pro vytváření a synchronizaci vláken definovaná standardem POSIX.
- Knihovna Pthreads poskytuje unifikované API:
  - Nepodporuje-li JOS vlákna, knihovna Pthreads bude pracovat čistě s ULT
  - Implementuje-li příslušné jádro KLT, pak toho knihovna Pthreads bude využívat
  - Pthreads je tedy systémově závislá knihovna
- Vlákna Linux:
  - Linux nazývá vlákna tasks
  - Linux má stejnou strukturu `task_struct` pro procesy i pro vlákna
  - Lze použít knihovnu pthreads
  - Vytváření vláken je realizováno službou OS `clone()`

Pthreads API

Příklad: Samostatné vlákno, které počítá součet prvních n celých čísel

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

int sum;                                        /* sdílená data */
void *runner(void *param) {   /* funkce realizující vlákno */
    int upper = atoi(param);  int i;  sum = 0;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
            sum += i;
    }
    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    pthread_t tid;                          /* identifikátor vlákna*/
    pthread_attr_t attr;                    /* atributy vlákna */
    pthread_attr_init(&attr);                       /* inicializuj implicitní atributy
    pthread_create(&tid, &attr, runner, argv[1]);   /* vytvoř vlákno */
    pthread_join(tid,NULL);                         /* čekej až vlákno skončí */
    printf("sum = %d\n",sum);
}
```

Vlákna ve Windows

- Aplikace ve Windows běží jako proces tvořený jedním nebo více vlákny
- Windows implementují mapování 1:1
- Někteří autoři dokonce tvrdí, že Proces se nemůže vykonávat, neboť je jen kontejnerem pro vlákna a jen ta jsou schopná běhu
- Každé vlákno má:
  - svůj identifikátor vlákna
  - sadu registrů (obsah je ukládán v TCM)
  - samostatný uživatelský a systémový zásobník
  - privátní datovou oblast

Kvíz – Vlákna v Javě

Mohou být v Javě vlákna?

A - Nemohou, Java je interpretovaný jazyk

B - Mohou, jsou jen implementovány na úrovni Java virtual machine

C - Mohou, jen když je Java přeložena do strojového kódu

D - Mohou, jde o nová vlákna Java virtual machine

Vlákna v Javě

- Vlákna v Javě:
  - Java má třídu „Thread“ a instancí je vlákno
  - Samozřejmě lze z třídy Thread odvodit podtřídu a některé metody přepsat
  - JVM pro každé vlákno vytváří jeho Java zásobník, kde jsou lokální třídy nedostupné pro ostatní vlákna
  - JVM spolu se základními Java třídami vlastně vytváří virtuální stroj obsahující jak vlastní JVM tak i na něm běžící OS podporující vlákna
  - Pokud se jedná o OS podporující vlákna pak jsou vlákna JVM mapována 1:1 na vlákna OS

Vlákna v Javě

Dva příklady jak vytvořit vlákno v Javě

```
class CounterThread extends Thread {
    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}


Thread counterThread = new CounterThread();

counterThread.start();
```

```
class Counter implements Runnable {
    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}

Runnable counter = new Counter();
Thread counterThread = new Thread(counter);

counterThread.start();
```