

Výlet do jádra Linuxu

(PB071 – Úvod do jazyka C)

Jiří Slabý

Fakulta Informatiky, Masarykova Univerzita

Brno, 17. května 2010

- Úvod do světa jádra, rozdíly
- Paralelismus
- Komunikace jádro ↔ uživatelský prostor
- Navazuje: přednáška R. Krejčího

- Co je to jádro OS?
 - Prostředník mezi procesy a HW
 - Ovládá a zpřístupňuje jej
- Běží ve speciálním režimu
- Programování jádra je specifické (jako každé jiné)

Proč umět programovat jádro?

- Nový hardware (žádné ovladače)
- Starý hardware (špatné ovladače)
- Nová funkcionality (lepší plánovač, síťový stack, atd.)
- Oprava chyb

- Co je to jádro OS?
 - Prostředník mezi procesy a HW
 - Ovládá a zpřístupňuje jej
- Běží ve speciálním režimu
- Programování jádra je specifické (jako každé jiné)

Proč umět programovat jádro?

- Nový hardware (žádné ovladače)
- Starý hardware (špatné ovladače)
- Nová funkcionality (lepší plánovač, síťový stack, atd.)
- Oprava chyb

Rozdíly program vs. jádro

Program (v C)

- Knihovny
 - *libc* (standardní funkce typu `printf`, `strlen`, `malloc`, ...)
 - Ostatní – přidávající funkcionality (*pthread*, *gmp*, ...)
- Oddělený paměťový prostor
 - Procesy se neovlivňují, pakliže to nebylo explicitně povoleno – zápis na adresu mimo alokovaných/mapovaných zabije aplikaci.
- Počáteční funkce `main`

Jádro

- Nic z výše uvedeného
- Některé funkce reimplementovány
 - `printf` jako `printk` v Linuxu, `printf` v BSD
- Žádná aritmetika s pohyblivou desetinnou čárkou apod.
- *Chyba v jádře často působí pád systému*

Rozdíly program vs. jádro

Program (v C)

- Knihovny
 - *libc* (standardní funkce typu `printf`, `strlen`, `malloc`, ...)
 - Ostatní – přidávající funkcionality (*pthread*, *gmp*, ...)
- Oddělený paměťový prostor
 - Procesy se neovlivňují, pakliže to nebylo explicitně povoleno – zápis na adresu mimo alokovaných/mapovaných zabije aplikaci.
- Počáteční funkce `main`

Jádro

- Nic z výše uvedeného
- Některé funkce reimplementovány
 - `printf` jako `printk` v Linuxu, `printf` v BSD
- Žádná aritmetika s pohyblivou desetinnou čárkou apod.
- *Chyba v jádře často působí pád systému*

- GNU C
 - Největší část, nutné části v assembleru
 - gcc alespoň 3.4.2
- Víceméně pevně daný styl
 - Tzv. *CodingStyle*
- Modulární
 - Různí správci
- Obsahuje jen nutné části
 - Např. boot-strap je práce pro GRUB, (E)LILO, QEMU, apod.
- Linkuje se do jednoho celku – monolit
 - Volání přímo (žádné předávání zpráv ve smyslu mikrojádra)
 - Avšak dovoluje přidávat (přilinkovat) za běhu kód ve formě modulů
- Počáteční funkci ("main") souboru definuje makro `module_init`
- Příklad:

Modul "Ahoj světe"

krtek.c	Makefile
<pre>#include <linux/module.h> static int my_init(void) { printk("Ahoj svete?\n"); return 0; } static void my_exit(void) { printk("Ahoj svete!\n"); } module_init(my_init); module_exit(my_exit);</pre>	<pre>KDIR=/lib/modules/\$(shell uname -r)/build KBUILD=\$(MAKE) -C \$(KDIR) M=\$(PWD) obj-m := krtek.o default: \$(KBUILD) modules</pre>

```
# insmod code.ko
# dmesg|tail -1
Ahoj svete?
```

```
# rmmod code
# dmesg|tail -1
Ahoj svete!
```

Modul "Ahoj světe"

krtek.c	Makefile
<pre>#include <linux/module.h> static int my_init(void) { printk("Ahoj svete?\n"); return 0; } static void my_exit(void) { printk("Ahoj svete!\n"); } module_init(my_init); module_exit(my_exit);</pre>	<pre>KDIR=/lib/modules/\$(shell uname -r)/build KBUILD=\$(MAKE) -C \$(KDIR) M=\$(PWD) obj-m := krtek.o default: \$(KBUILD) modules</pre>

```
# insmod code.ko
# dmesg|tail -1
Ahoj svete?
```

```
# rmmod code
# dmesg|tail -1
Ahoj svete!
```

2 úrovně zabezpečení

- *Uživatelský prostor* – kontrolované vykonávání instrukcí
- *Prostor jádra* – neomezené vykonávání (kód může cokoliv)
- Na x86 odpovídá privilege ring 0 a 3 – podpora CPU
- **Větší důraz na bezpečnost**
- Demontrace

Několik kontextů vykonávání kódu

- *Proces* – proces zavolá jádro
- *Prerušeni* – kontext právě přerušeno procesu
- V kontextu přerušeni nelze spát
 - Má za sebou cizí proces
 - Přerušeni jsou zakázána – není, kdo by proces vzbudil
- Obecněji atomický (zahrnuje spinlocky) vs. neatomický
- **Pozor na to, co se kde (ne)může volat**

2 úrovně zabezpečení

- *Uživatelský prostor* – kontrolované vykonávání instrukcí
- *Prostor jádra* – neomezené vykonávání (kód může cokoliv)
- Na x86 odpovídá privilege ring 0 a 3 – podpora CPU
- **Větší důraz na bezpečnost**
- Demontrace

Několik kontextů vykonávání kódu

- *Proces* – proces zavolá jádro
- *Prerušeni* – kontext právě přerušeno procesu
- V kontextu přerušeni nelze spát
 - Má za sebou cizí proces
 - Přerušeni jsou zakázána – není, kdo by proces vzbudil
- Obecněji atomický (zahrnuje spinlocky) vs. neatomický
- **Pozor na to, co se kde (ne)může volat**

Specifická alokace paměti

- Stránkový alokátor (`alloc_page`)
- SLAB alokátor (`kmalloc`)
- Alokace a namapování do virtuální paměti (`vmalloc`)
- **Paměť není souvislá**

Řízení událostmi (event-driven)

- Přerušování od HW, požadavky uživatele, komunikace modulů, atd.
- Důsledek: implicitní souběžnost kódu (viz dále)
- **Je nutné zachovat pořadí vykonání instrukcí?**

Copak se takhle dá programovat?

Dá.

Specifická alokace paměti

- Stránkový alokátor (`alloc_page`)
- SLAB alokátor (`kmalloc`)
- Alokace a namapování do virtuální paměti (`vmalloc`)
- **Paměť není souvislá**

Řízení událostmi (event-driven)

- Přerušování od HW, požadavky uživatele, komunikace modulů, atd.
- Důsledek: implicitní souběžnost kódu (viz dále)
- **Je nutné zachovat pořadí vykonání instrukcí?**

Copak se takhle dá programovat?

Dá.

Specifická alokace paměti

- Stránkový alokátor (`alloc_page`)
- SLAB alokátor (`kmalloc`)
- Alokace a namapování do virtuální paměti (`vmalloc`)
- **Paměť není souvislá**

Řízení událostmi (event-driven)

- Přerušování od HW, požadavky uživatele, komunikace modulů, atd.
- Důsledek: implicitní souběžnost kódu (viz dále)
- **Je nutné zachovat pořadí vykonání instrukcí?**

Copak se takhle dá programovat?

Dá.

Specifická alokace paměti

- Stránkový alokátor (`alloc_page`)
- SLAB alokátor (`kmalloc`)
- Alokace a namapování do virtuální paměti (`vmalloc`)
- **Paměť není souvislá**

Řízení událostmi (event-driven)

- Přerušování od HW, požadavky uživatele, komunikace modulů, atd.
- Důsledek: implicitní souběžnost kódu (viz dále)
- **Je nutné zachovat pořadí vykonání instrukcí?**

Copak se takhle dá programovat?

Dá.

Paralelismus

Souběžnost kódu I.

Multitasking a problémy s ním spojené

Příklad: 2 procesy provádějící následující kód

```
a = 2;  
a = a - a;
```

Může být a různé od 0?

Ano (jeden z mnoha chybných souběhů):

PROCES 1		PROCES 2
store 2, a		
load a, reg1		
<context switch>	->	store 2, a
		load a, reg1
		load a, reg2
		sub reg1, reg2
		store reg1, a
load a, reg2	<-	<context switch>
sub reg1, reg2		# 2 - 0
store reg1, a		# a je 2

Souběžnost kódu I.

Multitasking a problémy s ním spojené

Příklad: 2 procesy provádějící následující kód

```
a = 2;  
a = a - a;
```

Může být a různé od 0?

Ano (jeden z mnoha chybných souběhů):

PROCES 1		PROCES 2
store 2, a		
load a, reg1		
<context switch>	->	store 2, a
		load a, reg1
		load a, reg2
		sub reg1, reg2
		store reg1, a
load a, reg2	<-	<context switch>
sub reg1, reg2		# 2 - 0
store reg1, a		# a je 2

Řešení

- Zámky
 - Vytvoří kritickou sekci
 - 2 procesy nemohou vstoupit do kritické sekce (až na semaforey)
- Jádro poskytuje několik typů zámků
 - *mutex* – spící, nevytváří atomický kontext, lze v něm spát
 - *semafor* – podobný mutexu, obsahuje počítadlo, je zanořitelný
 - *spinlock* – nespící (busy-waiting), atomický
 - *BKL* – zastaralý, hrubozrnný, nepoužívat
 - A další, speciální. . .

Náš kód tedy má vypadat takto:

```
spin_lock(&my_lock);  
a = 2;  
a = a - a;  
spin_unlock(&my_lock);
```

Co když přijde po `a = 2` přerušení obsahující stejný kód?

Řešení

- Zámky
 - Vytvoří kritickou sekci
 - 2 procesy nemohou vstoupit do kritické sekce (až na semaforey)
- Jádro poskytuje několik typů zámků
 - *mutex* – spící, nevytváří atomický kontext, lze v něm spát
 - *semafor* – podobný mutexu, obsahuje počítadlo, je zanořitelný
 - *spinlock* – nespící (busy-waiting), atomický
 - *BKL* – zastaralý, hrubozrnný, nepoužívat
 - A další, speciální...

Náš kód tedy má vypadat takto:

```
spin_lock(&my_lock);  
a = 2;  
a = a - a;  
spin_unlock(&my_lock);
```

Co když přijde po `a = 2` přerušení obsahující stejný kód?

Řešení

- Zámky
 - Vytvoří kritickou sekci
 - 2 procesy nemohou vstoupit do kritické sekce (až na semaforey)
- Jádro poskytuje několik typů zámků
 - *mutex* – spící, nevytváří atomický kontext, lze v něm spát
 - *semafor* – podobný mutexu, obsahuje počítadlo, je zanořitelný
 - *spinlock* – nespící (busy-waiting), atomický
 - *BKL* – zastaralý, hrubozrnný, nepoužívat
 - A další, speciální...

Náš kód tedy má vypadat takto:

```
spin_lock(&my_lock);  
a = 2;  
a = a - a;  
spin_unlock(&my_lock);
```

Co když přijde po $a = 2$ přerušení obsahující stejný kód?

Souběžnost kódu III.



- Deadlock
 - Jádro je řeší ignorováním
 - Spoléhá na programátora, že nenastanou

V našem příkladě

PROCES		PŘERUŠENÍ
<code>spin_lock(&my_lock);</code> <code>a = 2;</code> <code><interrupt></code>	<code>-></code>	<code>spin_lock(&my_lock);</code> <code><deadlock></code>
<code><deadlock></code>		

Řešení 2

- Na základě kontextu se používají různé formy spinlocků
 - Obecně zákaz přerušení na aktuálním CPU

Tedy výsledek:

```
spin_lock_irq(&my_lock);  
a = 2;  
a = a - a;  
spin_unlock_irq(&my_lock);
```

V našem příkladě

PROCES

```
spin_lock(&my_lock);  
a = 2;  
<interrupt>      ->  
<deadlock>
```

PŘERUŠENÍ

```
spin_lock(&my_lock);  
<deadlock>
```

Řešení 2

- Na základě kontextu se používají různé formy spinlocků
 - Obecně zákaz přerušení na aktuálním CPU

Tedy výsledek:

```
spin_lock_irq(&my_lock);  
a = 2;  
a = a - a;  
spin_unlock_irq(&my_lock);
```

V našem příkladě

PROCES

```
spin_lock(&my_lock);  
a = 2;  
<interrupt>      ->  
<deadlock>
```

PŘERUŠENÍ

```
spin_lock(&my_lock);  
<deadlock>
```

Řešení 2

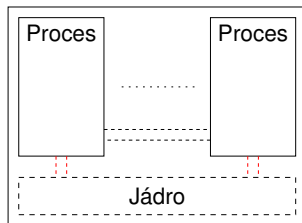
- Na základě kontextu se používají různé formy spinlocků
 - Obecně zákaz přerušení na aktuálním CPU

Tedy výsledek:

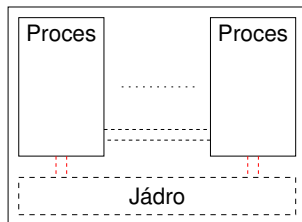
```
spin_lock_irq(&my_lock);  
a = 2;  
a = a - a;  
spin_unlock_irq(&my_lock);
```

Komunikace

jádro ↔ uživatelský prostor



- Základní forma komunikace: volání funkcí
- Co se děje, když se z programu zavolá `fwrite`?
 - `fwrite` je funkce *libc*
 - Vnitřní buffery, volání `write`
 - `write` je (stále) funkce *libc*
 - Nyní nutno volat funkci jádra
- Jádro nelze volat přímo
 - Důvod: dvě úrovně zabezpečení



- Základní forma komunikace: volání funkcí
- Co se děje, když se z programu zavolá `fwrite`?
 - `fwrite` je funkce *libc*
 - Vnitřní buffery, volání `write`
 - `write` je (stále) funkce *libc*
 - Nyní nutno volat funkci jádra
- Jádro nelze volat přímo
 - Důvod: dvě úrovně zabezpečení

- Nutno provést přepnutí privilegií (obsah registrů)
 - Tzv. system call (syscall)

1) speciální instrukcí procesoru

- Na x86 *int* (80), *sysenter* nebo *syscall*, na ia64 *break* nebo *epc*, na ARM *swi* atd.
- Princip vesměs stejný
 - Přeruší se vykonávání (podobné IRQ z HW)
 - Uloží se registry
 - Začne se vykonávat kód jádra
 - Provede se potřebné volání
 - Obnoví se skoro všechny registry (a oprávnění)
 - Pokračuje se ve vykonávání programu
- Nepatří mezi nejrychlejší operace
 - Vylepšené instrukce: *sysenter*, *syscall*, *epc*
- Příklad

- Nutno provést přepnutí privilegií (obsah registrů)
 - Tzv. system call (syscall)

1) speciální instrukcí procesoru

- Na x86 *int* (80), *sysenter* nebo *syscall*, na ia64 *break* nebo *epc*, na ARM *swi* atd.
- Princip vesměs stejný
 - Přeruší se vykonávání (podobné IRQ z HW)
 - Uloží se registry
 - Začne se vykonávat kód jádra
 - Provede se potřebné volání
 - Obnoví se skoro všechny registry (a oprávnění)
 - Pokračuje se ve vykonávání programu
- Nepatří mezi nejrychlejší operace
 - Vylepšené instrukce: *sysenter*, *syscall*, *epc*
- Příklad

- Nutno provést přepnutí privilegií (obsah registrů)
 - Tzv. system call (syscall)

1) speciální instrukcí procesoru

- Na x86 *int* (80), *sysenter* nebo *syscall*, na ia64 *break* nebo *epc*, na ARM *swi* atd.
- Princip vesměs stejný
 - Přeruší se vykonávání (podobné IRQ z HW)
 - Uloží se registry
 - Začne se vykonávat kód jádra
 - Proveďte se potřebné volání
 - Obnoví se skoro všechny registry (a oprávnění)
 - Pokračuje se ve vykonávání programu
- Nepatří mezi nejrychlejší operace
 - Vylepšené instrukce: *sysenter*, *syscall*, *epc*
- Příklad

- Nutno provést přepnutí privilegií (obsah registrů)
 - Tzv. system call (syscall)

1) speciální instrukcí procesoru

- Na x86 *int* (80), *sysenter* nebo *syscall*, na ia64 *break* nebo *epc*, na ARM *swi* atd.
- Princip vesměs stejný
 - Přeruší se vykonávání (podobné IRQ z HW)
 - Uloží se registry
 - Začne se vykonávat kód jádra
 - Proveďte se potřebné volání
 - Obnoví se skoro všechny registry (a oprávnění)
 - Pokračuje se ve vykonávání programu
- Nepatří mezi nejrychlejší operace
 - Vylepšené instrukce: *sysenter*, *syscall*, *epc*
- Příklad

Do jádra jsme se dostali, co dál?

- V jednom z registrů je číslo syscallu
- Jádro v tabulce zjistí adresu funkce a zavolá ji
- Ta si zkontroluje (zdroj chyb) a zkopíruje vstupy
- Pro `write` se zavolá odpovídající souborový systém
- Do jednoho z registrů se zapíše návratová hodnota

2) skokem do speciálního prostoru

- Tzv. virtuální syscalls
- *vsyscall* (zastaralé) a *vdso*
- Zdrojový kód v jádře v arch/.../{vdso,vsyscall}*
- Zvláštní dynamická knihovna
 - Je na speciálně označené stránce v jádře
 - Při skoku sem, zvýšena privilegia
- Takto implementováno několik málo volání
 - V závislosti na arch, např. `gettimeofday`, `getcpu`

```
$ ldd /bin/true
linux-vdso.so.1 => (0x00007fffbe3ff000)
libc.so.6 => /lib64/libc.so.6 (0x00007facbe6e8000)
/lib64/ld-linux-x86-64.so.2 (0x00007facbea48000)
$ cat /proc/self/maps
...
```

2) skokem do speciálního prostoru

- Tzv. virtuální syscalls
- *vsyscall* (zastaralé) a *vdso*
- Zdrojový kód v jádře v arch/.../{vdso,vsyscall}*
- Zvláštní dynamická knihovna
 - Je na speciálně označené stránce v jádře
 - Při skoku sem, zvýšena privilegia
- Takto implementováno několik málo volání
 - V závislosti na arch, např. `gettimeofday`, `getcpu`

```
$ ldd /bin/true
linux-vdso.so.1 => (0x00007fffbe3ff000)
libc.so.6 => /lib64/libc.so.6 (0x00007facbe6e8000)
/lib64/ld-linux-x86-64.so.2 (0x00007facbea48000)
$ cat /proc/self/maps
...
```

3) ostatní formy komunikace

- Syscally trochu jinak
 - Nelze pro každou funkcionalitu/ovladač přidat jeden syscall
 - Ovladače vytváří `/dev/*` zařízení
 - Nad nimi se používá `open/read/write/ioctl/.../close`
 - `ioctl` je rozšířeným způsobem komunikace

```
int ioctl(int fd, int command, unsigned long argument);
```

- Sdílená paměť
 - Producer-consumer
 - Uživatel vidí i na data např. od síťové karty
 - Bez kopírování dat
 - Pozor na koherentnost dat!
 - Některé architektury používají bounce buffery

3) ostatní formy komunikace

- Syscally trochu jinak
 - Nelze pro každou funkcionalitu/ovladač přidat jeden syscall
 - Ovladače vytváří `/dev/*` zařízení
 - Nad nimi se používá `open/read/write/ioctl/.../close`
 - `ioctl` je rozšířeným způsobem komunikace

```
int ioctl(int fd, int command, unsigned long argument);
```

- Sdílená paměť
 - Producer-consumer
 - Uživatel vidí i na data např. od síťové karty
 - Bez kopírování dat
 - Pozor na koherentnost dat!
 - Některé architektury používají bounce buffery

- Jádro se v mnoha ohledech liší od programů
 - Někdy to může vypadat, že stejná je jen syntax
- Paralelismus není vždy výhra
 - Pro vývojáře spíše naopak
- Komunikace s uživateli je náročná
 - Uživatelé jsou zlí

- Zaujalo vás téma?
 - Mnoho projektů, kde se lze s jádrem utkat
 - PB173 – Ovladače jádra – Linux

- Jádro se v mnoha ohledech liší od programů
 - Někdy to může vypadat, že stejná je jen syntax
- Paralelismus není vždy výhra
 - Pro vývojáře spíše naopak
- Komunikace s uživateli je náročná
 - Uživatelé jsou zlí

- Zaujalo vás téma?
 - Mnoho projektů, kde se lze s jádrem utkat
 - PB173 – Ovladače jádra – Linux

Děkuji za pozornost!

Dotazy?

Přivítejme Radka Krejčího s navazující přednáškou.

Děkuji za pozornost!

Dotazy?

Přivítejme Radka Krejčího s navazující přednáškou.