

# UNIX

## Programování a správa systému I

Jan Kasprzak  
<kas@fi.muni.cz>  
<https://www.fi.muni.cz/~kas/>

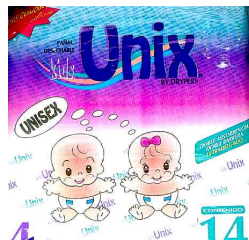
*Motto: Virtual memory is like a game you can't win;  
however, without VM there's truly nothing to lose.  
—Rik van Riel*

# Kapitola 1

## Úvod

# Předpoklady

- Programování v C – syntaxe, paměťový model, průběh kompilace.
- UNIX z uživatelského hlediska – shell, soubory, procesy.



# Cíle kursu

- Programování pod UNIXem  
rozhraní dle Single UNIX Specification.
- Jádro UNIXu  
principy činnosti, paměťový model, procesy.

# Ukončení předmětu

- Průběžné odpovědníky
  - 5 dní na splnění (otevřeno 12+ dní, případně neschopenka na posledních 7 dní)
  - nejvýše tři nesplněné
  - 4 otázky, +2 body/-1 bod
- Závěrečný test: výběr právě jedné možnosti, 15 otázek, +8/-4 body
- Předmět je těžký! - nepodcenit
- Zkontrolujte si zapsané ukončení

## Upozornění:

Naučit se z paměti slidy nestačí!

# Ukončení předmětu

- Průběžné odpovědníky
  - 5 dní na splnění (otevřeno 12+ dní, případně neschopenka na posledních 7 dní)
  - nejvýše tři nesplněné
  - 4 otázky, +2 body/-1 bod
- Závěrečný test: výběr právě jedné možnosti, 15 otázek, +8/-4 body
- Předmět je těžký! - nepodcenit
- Zkontrolujte si zapsané ukončení



## Upozornění:

Naučit se z paměti slidy nestačí!

# Obsah přednášky - I.

- Vývojové prostředí pod UNIXem - nástroje.
- Normy API pro jazyk C pro UNIX
- Program podle ANSI C - limity, start a ukončení programu, argumenty, proměnné prostředí, práce s pamětí, vzdálené skoky. Hlavičkové soubory a knihovny. Sdílené knihovny.
- **Jádro** - start jádra, architektura jádra, paměťový model, komunikace s jádrem, paralelismus v jádře.
- **Proces** - paměťový model, vznik a zánik procesu, program na disku.

# Obsah přednášky - II.

- **Vstupní/výstupní operace** – deskriptor, operace s deskriptory.
- **Soubory a adresáře** – i-uzel, atributy souboru, přístupová práva, speciální soubory.
- **Komunikace mezi procesy** – roura, signály.
- **Pokročilé I/O operace** – zamykání souborů, scatter-gather I/O, soubory mapované do paměti, multiplexování vstupů a výstupů.
- **Vlákna**



# Typografické konvence

- Specifická vlastnost pro Linux: 
- ... pro BSD , Solaris 
- ... pro IRIX , Red Hat/Fedoru 
- ... pro GNU nástroje: 



## Úkol:

Doma se zkuste zamyslet a vyřešit.



## Příklad: Tohle rozhodně nezkoušejte ^\_~

```
root@eva01# rm -rf /
```

# Další zdroje informací

- Tato prezentace:  
<https://www.fi.muni.cz/~kas/pv065/>
- PB173 Tématický vývoj aplikací v jazyce C/C++
  - Skupina Systémové programování - Linux
- PB152cv Operační systémy - cvičení
- Linux Weekly - <https://lwn.net/>
- Experimentujte! - fakultní linuxové počítače, vlastní stroj s Linuxem, <https://stratus.fi.muni.cz/>, ...

# Další zdroje informací

- Tato prezentace:  
<https://www.fi.muni.cz/~kas/pv065/>
- **PB173** Tématický vývoj aplikací v jazyce C/C++
  - Skupina **Systémové programování - Linux**
- PB152cv Operační systémy - cvičení
- Linux Weekly - <https://lwn.net/>
- Experimentujte! - fakultní linuxové počítače, vlastní stroj s Linuxem, <https://stratus.fi.muni.cz/>, ...

# Další zdroje informací

- Tato prezentace:  
<https://www.fi.muni.cz/~kas/pv065/>
- PB173 Tématický vývoj aplikací v jazyce C/C++
  - Skupina [Systémové programování - Linux](#)
- **PB152cv** Operační systémy - cvičení
- Linux Weekly - <https://lwn.net/>
- Experimentujte! - fakultní linuxové počítače, vlastní stroj s Linuxem, <https://stratus.fi.muni.cz/>, ...

# Další zdroje informací

- Tato prezentace:  
<https://www.fi.muni.cz/~kas/pv065/>
- PB173 Tématický vývoj aplikací v jazyce C/C++
  - Skupina [Systémové programování - Linux](#)
- PB152cv Operační systémy - cvičení
- **Linux Weekly** - <https://lwn.net/>
- Experimentujte! - fakultní linuxové počítače, vlastní stroj s Linuxem, <https://stratus.fi.muni.cz/>, ...

# Další zdroje informací

- Tato prezentace:  
<https://www.fi.muni.cz/~kas/pv065/>
- PB173 Tématický vývoj aplikací v jazyce C/C++
  - Skupina [Systémové programování - Linux](#)
- PB152cv Operační systémy - cvičení
- Linux Weekly - <https://lwn.net/>
- **Experimentujte!** - fakultní linuxové počítače, vlastní stroj s Linuxem, <https://stratus.fi.muni.cz/>, ...

# Kapitola 2

## Vývojové prostředí

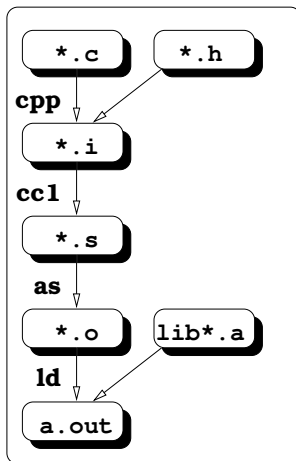


# Příklad: Rychlý start

```
$ cat > ritchie.c
#include <stdio.h>
main() { printf("Hello, world!\n"); }
^D
$ cc ritchie.c
$ ./a.out
Hello, world!
$
```



# Kompilace C-programu



# Kompilátor cc

- GNU C/C++
- LLVM/Clang
- Spouští další programy
  - `cpp(1)`
  - `comp/cc1`
  - `as(1)`
  - `ld(1)`
- Lze spouštět i jen jednotlivé části překladu.
- Start kompilace se řídí příponou souboru.

# Kompilátor cc

- GNU C/C++
- LLVM/Clang
- Spouští další programy
  - `cpp(1)`
  - `comp/cc1`
  - `as(1)`
  - `ld(1)`
- Lze spouštět i jen jednotlivé části překladu.
- **Start kompilace** se řídí příponou souboru.

# Konec kompilace:

- E - jen preprocesor.
- S - až po assembler.
- c - včetně assembleru.
- o *jméno* - jméno výstupního souboru.

# Parametry preprocesoru:

- Dmakro*
- Dmakro=hodnota* - nadefinuje makro pro preprocesor.
- Umakro* - ruší definici makra.
- Iadresář* - adresář pro hlavičkové soubory.
  - I-* - vypíná standardní adresáře (/usr/include).

# Parametry kompilátoru:

- O[číslo] - zapíná optimalizaci.
- g - zapíná generování ladících informací.
- p - profilovací informace pro prof(1).
- pg - profilovací informace pro gprof(1).

## Úkol:

Napište triviální program, který bude volat funkci printf(3) se dvěma parametry. Program zkompilujte s výstupem do assembleru bez optimalizace a s optimalizacemi. Jaké změny udělal optimalizující kompilátor? Vyzkoušejte dle možnosti různé verze kompilátoru a různé platformy.

# Parametry kompilátoru:

- O[číslo] - zapíná optimalizaci.
- g - zapíná generování ladících informací.
- p - profilovací informace pro prof(1).
- pg - profilovací informace pro gprof(1).



## Úkol:

Napište triviální program, který bude volat funkci printf(3) se dvěma parametry. Program zkompilujte s výstupem do assembleru bez optimalizace a s optimalizacemi. Jaké změny udělal optimalizující kompilátor? Vyzkoušejte dle možnosti různé verze kompilátoru a různé platformy.

# Parametry linkeru:

- *Ladresář* – adresář pro knihovny.
- *nostdlib* – bez standardních knihoven.
- *lknihovna* – přidá soubor *libknihovna.a*, případně *.SO*.
  - *static* – statické linkování.
  - *shared* – sdílené knihovny.
  - *s* – odstranit tabulku symbolů.



# Program v paměti

```
char znak;  
int funkce(int argument) {  
    int cislo;  
    /* ... */  
}
```

- **Text** – vlastní strojový kód (obvykle jen pro čtení/provádění). `&funkce`.
- **Data** – čtení i zápis. `&znak`
- **Zásobník** – čtení i zápis, zvětšuje se obvykle směrem k nižším adresám. `&cislo`

## Úkol:

Kam padne adresa `&argument`?

# Program v paměti

```
char znak;  
int funkce(int argument) {  
    int cislo;  
    /* ... */  
}
```

- **Text** – vlastní strojový kód (obvykle jen pro čtení/provádění). `&funkce`.
- **Data** – čtení i zápis. `&znak`
- **Zásobník** – čtení i zápis, zvětšuje se obvykle směrem k nižším adresám. `&cislo`



## Úkol:

Kam padne adresa `&argument`?

# Umístění proměnných v paměti

- **Na zásobníku** - automatické, deklarované uvnitř funkce.
- **V datové části** - static nebo mimo funkce.

```
int jezek;  
void funkce() {  
    int ptakopysk;  
    static int tucnak;  
    /* ... */  
}
```

# Viditelnost proměnných

- **Statické** - `static` - jen uvnitř modulu.
- **Globální** - mimo funkce a bez `static` - viditelné ze všech modulů.

# Viditelnost proměnných

```
/* — data.c — */
int odpoved;
static char *otazka;
/* — thought.c — */
extern int odpoved;
extern char *otazka;
hlubina_mysleni() {
    odpoved = 42;
    sleep(60*60*24*365*10000000);
    otazka = "Co dostaneme, když "
           "vynásobíme šest devíti?";
}
```

```
$ cc -c data.c
```

```
$ cc -c thought.c
```

```
$ cc -o hlubina data.o thought.o main.o
```

# Program make

- Řízená kompilace z více modulů
- Soubor `Makefile` nebo `makefile`

-f *Makefile* - soubor místo `makefile` nebo `Makefile`

- i - ignoruj chyby
- n - vypiš příkazy, ale neprováděj
- k - pokračuj i po chybě

# Proměnné

- *proměnná=hodnota*
- $\$(proměnná)$  - použití



## Příklad:

```
CC=gcc -g  
CFLAGS=$(OPT_FLAGS) $(DEBUG_FLAGS)
```

# Závislosti

- *cíl: prerekvizita ...*
- ze kterých zdrojů se vyrobí příslušný cíl
- implicitní závislosti podle přípon



## Příklad:

```
program.o: program.c program.h
```



# Akce



## Příklad:

```
$(CC) -c program.c
```

- Prefix: **tabulátor** (ne mezery!)
- Jak vyrobit nový *cíl* na základě změněných *prerekvizit*?



# Příklad: Makefile

```
CFLAGS=-O2
LDFLAGS=-s
# CFLAGS=-g
# LDFLAGS=-g
all: program
clean:
    -rm *.o a.out core
program: modul1.o modul2.o
    $(CC) -o $@ modul1.o modul2.o $(LDFLAGS)
    @echo "Kompilace hotova."
modul1.o: modul1.c program.h
modul2.o: modul2.c program.h
    $(CC) -c $(CFLAGS) modul2.c
```

# Alternativní nástroje

- **GNU Autotools** - autoconf, automake, libtool
- cmake
- scons
- ninja

# Alternativní nástroje

- GNU Autotools - autoconf, automake, libtool
- **cmake**
- scons
- ninja

# Alternativní nástroje

- GNU Autotools - autoconf, automake, libtool
- cmake
- **scons**
- ninja

# Alternativní nástroje

- GNU Autotools - autoconf, automake, libtool
- cmake
- scon
- **ninja**

# Další programy

**nm(1)****Výpis tabulky symbolů**

```
$ nm program
$ nm ritchie.o
00000000 t gcc2_compiled.
00000000 T main
          U printf                # opravdu?
```

**strip(1)****Odstranění tabulky symbolů**

```
$ strip executable
```

# Další programy

**nm(1)****Výpis tabulky symbolů**

```
$ nm program
$ nm ritchie.o
00000000 t gcc2_compiled.
00000000 T main
          U printf                # opravdu?
```

**strip(1)****Odstranění tabulky symbolů**

```
$ strip executable
```



# Další programy

**size(1)**

**Velikost objektového souboru**

```
$ size objfile
```

```
$ size x.o
```

text	data	bss	dec	hex	filename
20	3	0	23	17	x.o

# Informace o objektovém souboru

## objdump(1)

```
$ objdump -h ritchie.o
x.o:      file format elf64-x86-64
Sections:
Idx Name  Size      VMA           ...
  0 .text  00000010  0000000000000000 ...
          CONTENTS, ALLOC, LOAD, RELOC...
  1 .data  00000000  0000000000000000 ...
          CONTENTS, ALLOC, LOAD, DATA ...
...
...
...
...
```

- Výpis informací nejen z objektového souboru (\*.o).
- Funguje i jako disassembler.

# Knihovny



- Sada funkcí a proměnných s pevně definovaným rozhraním.
- Definice rozhraní - *hlavičkový soubor*.
- Umístění - adresáře `/lib`, `/usr/lib`.
  - Multiarch/multilib systémy - například `lib64`.
- Statické versus sdílené.
- Linkování v době *kompilace* versus v době *běhu*.

# Statické knihovny

- **Statická knihovna** – archiv objektových souborů.
- **Linker** – vytáhne z knihovny \*.o soubory.
- **Spustitelný soubor** – obsahuje kopii \*.o z knihovny.
- **Staticky linkovaný program**
  - větší než dynamicky linkovaný,
  - neumí sdílet kód s jinými programy,
  - ale je v podstatě nezávislý.

# Formát statických knihoven

**ar(1)****Archivace souborů**

```
$ ar rcs libknihovna.a *.o  
$ ar t /usr/lib/libc.a
```

ar(1) je obecný archivátor, statické knihovny jsou jen jedno z použití.

**ranlib(1)****Index archívu**

```
$ ranlib soubor.a
```

- index všech symbolů ve všech objektových souborech archívu
- V některých systémech – totéž co ar -s

# Formát statických knihoven

## ar(1)

## Archivace souborů

```
$ ar rcs libknihovna.a *.o  
$ ar t /usr/lib/libc.a
```

ar(1) je obecný archivátor, statické knihovny jsou jen jedno z použití.

## ranlib(1)

## Index archívu

```
$ ranlib soubor.a
```

- index všech symbolů ve všech objektových souborech archívu
- V některých systémech – totéž co ar -s 🐼

# Sdílené knihovny


- Dynamicky linkované knihovny/moduly
  - kód, přiřazený k programu až po spuštění
  - sdílené knihovny nebo plug-iny
- Dynamický linker - `/lib/ld.so`
  - první načtená dynamická knihovna
  - stará se o přiřazení dalších knihoven


# Ovládání dynamického linkeru

`LD_LIBRARY_PATH` - seznam adresářů, oddělený dvojtečkami. Určuje, kde se budou hledat dynamicky linkované knihovny.

`LD_PRELOAD` - objekt, který bude přilinkován jako první. Např. pro předefinování knihovnické funkce. U set-uid a set-gid programů dynamický linker ignoruje výše uvedené proměnné.

`/etc/ld.so.conf` - globální konfigurace.

`/etc/ld.so.conf.d/` - usnadnění práce správcům balíčků. 

`ldconfig(8)` - generuje symlinky podle verzí a cache. 



# Linkování v době kompilace

- Linux libc4 (a.out), SunOS 4, SVr3
- **Umístění** – na pevně dané adrese v adresním prostoru procesu.
- **Run-time** – pouze přimapování sdílené knihovny.
- **Výhody** – rychlý start programu.
- **Nevýhody**:
  - složitá výroba
  - nemožnost linkování v době běhu
  - omezená velikost adresního prostoru (4GB pro 32-bitové systémy, musí vystačit pro všechny existující sdílené knihovny)
  - problém s verzemi.

# Linkování v době běhu - formát ELF

- Extensible Linking Format
- Executable and Linkable Format
- AT&T System V Release 4, Linux libc5+
- Křížové odkazy - řešeny v době běhu.
- Kód nezávislý na umístění
  - *position independent code*, PIC
- Verze symbolů - např. při změně parametrů
  - Linux libc6
- Výhody - dynamické linkování (např. plug-iny), možnost předefinovat symbol v knihovně.
- Nevýhody - pomalejší start programu, potenciálně pomalejší běh PIC kódu (je nutno alokovat jeden registr jako adresu začátku knihovny).
- Problém - nesdílitelné části kódu (křížové odkazy).
  - viz též prelink(8)

# Použité knihovny

## ldd(1)

## Loader dependencies

```
$ ldd [-dr] program
$ ldd /usr/bin/vi
libtermcap.so.2 => /lib/libtermcap.so.2.0.8
libc.so.5 => /lib/libc.so.5.4.36
```

- d Provede doplnění křížových odkazů a ohlásí chybějící funkce.
- r Totéž, případné chyby hlásí nejen u funkcí, ale i u datových objektů.

## Úkol:

Zjistěte, které programy jsou v systémových adresářích /bin a /sbin (nebo /etc) staticky linkované.

# Použité knihovny

## ldd(1)

## Loader dependencies

```
$ ldd [-dr] program
$ ldd /usr/bin/vi
libtermcap.so.2 => /lib/libtermcap.so.2.0.8
libc.so.5 => /lib/libc.so.5.4.36
```

- d Proveďte doplnění křížových odkazů a ohlásí chybějící funkce.
- r Totéž, případné chyby hlásí nejen u funkcí, ale i u datových objektů.



## Úkol:

Zjistěte, které programy jsou v systémových adresářích /bin a /sbin (nebo /etc) staticky linkované.

# Linkování na různých systémech



## Přednáška na dobrou noc

Ofek Shilon:

Shared Libraries in Windows, in Linux, and yes—in C++  
Přednáška z Cppcon 2023

[https://youtu.be/\\_enXuIxuNV4](https://youtu.be/_enXuIxuNV4)

# Hlavičkové soubory

- Definice rozhraní ke knihovnám – typové kontroly a podobně.
- Definice konstant – NULL, stdin, EAGAIN ...
- Definice maker – isspace(), ntohl(), ...
- Neobsahují vlastní definice funkcí, jen deklarace prototypů.
- Umístění: – adresář /usr/include a podadresáře.
- Poznámka k privátním symbolům: Symboly začínající podtržítkem jsou privátní symboly systému.

## Úkol:

Je v systému definována konstanta pro  $\pi$ ? Ve kterém hlavičkovém souboru? Jak se jmenuje tato konstanta?

# Hlavičkové soubory

- Definice rozhraní ke knihovnám – typové kontroly a podobně.
- Definice konstant – NULL, stdin, EAGAIN ...
- Definice maker – isspace(), ntohl(), ...
- Neobsahují vlastní definice funkcí, jen deklarace prototypů.
- Umístění: – adresář /usr/include a podadresáře.
- Poznámka k privátním symbolům: Symboly začínající podtržítkem jsou privátní symboly systému.



## Úkol:

Je v systému definována konstanta pro  $\pi$ ? Ve kterém hlavičkovém souboru? Jak se jmenuje tato konstanta?

# Ladění programu

- Symbolické ladící informace – přepínač `-g` u kompilátoru.
- Ladění na úrovni assembleru
- Soubor `core` – obraz paměti procesu v době havárie. Lze vytvořit i uměle například zasláním signálu `SIGQUIT (Ctrl-\)`. Slouží k posmrtné analýze programu.
- Pozor na `ulimit -c`
- Pozor na `systemd-coredump(8)` a `coredumpctl(1)`.
- Ladění běžícího procesu probíhá přes službu jádra `ptrace(2)` s pomocí souborového systému `/proc`.



# Debuggery

- GNU debugger – gdb. Ovládání z příkazové řádky.
- Grafické front-endy pro gdb: cgdb, ddd, kdbg, nemiver, xxgdb, ...
- Integrovaná vývojová prostředí – anjuta, eclipse, geany, kdevelop, ...

# Kapitola 3

## Normy API

# ANSI C

- Schváleno 1989.
- ANSI Standard X3.159-1989.
- Jazyk C plus standardní knihovna.
- 15 sekcí knihovny podle 15 hlavičkových souborů (stdlib.h, stdio.h, string.h, atd.)
- Základní přenositelnost programů v C.
- Oproti UNIXu nedefinuje proces ani vztahy mezi procesy.
- Novější revize:
  - ISO C99 (C++ komentáře, inline funkce, atd.)
  - ISO C11 (vlákna, atomické typy, ...)
  - ISO C18 (jen upřesnění)

# IEEE POSIX

- Portable Operating System Interface – IEEE 1003.
- API UNIXu – POSIX.1, nejnovější revize 2017.
- Rozhraní shellu – POSIX.2.
- Real-time extenze – POSIX.1b (dříve POSIX.4).
- Vlákna – POSIX.1c (dříve POSIX.4a).

# Single UNIX Specification

- The Open Group – sloučení OSF a X/Open.
- SUSv1 – 1994, „UNIX 95“.
- SUSv2 – 1997, „UNIX 98“.
- SUSv3 – 2002, „UNIX 03“.
- SUSv4 – 2008, POSIX:2008, revize 2012, 2016, 2018.
- Zahrnuje **POSIX.1** a další standardy.
- V současné době používaná „definice UNIXu“.

# Další normy

- Viz sekce [Conforming To](#) v manuálových stránkách.
- [X/Open XPG3,4](#): X/Open Portability Guide – rozšíření POSIX.1.
- [FIPS 151-1 a 151-2](#) – Federal Information Processing Standard; upřesnění normy POSIX.1.
- [SVID3](#) – System V Interface Description – norma AT&T (popisuje SVr4)
- [SVID4](#) – zahrnuje POSIX.1 1990.
- [BSD](#) – označení pro extenze z 4.x BSD.

# Volitelné vlastnosti v normách

- Volby při kompilaci (podporuje systém řízení prací?)
- Limity při kompilaci (jaká je maximální hodnota proměnné typu `int`?)
- Limity při běhu (kolik nejvíce znaků může mít soubor v tomto adresáři?)

# ANSI C Limity

- Všechny při kompilaci.
- `<limits.h>`: `INT_MAX`, `UINT_MAX`, atd.
- `<float.h>`: podobné limity pro reálnou aritmetiku.
- `<stdio.h>`: - konstanta `FOPEN_MAX`.



# POSIX.1 - detekce verzí

```
#define _POSIX_SOURCE ... nebo  
#define _POSIX_C_SOURCE 200809L  
#include <unistd.h>
```

Konstanta `_POSIX_VERSION` pak určuje verzi normy POSIX, kterou systém splňuje.  
Viz též `feature_test_macros(7)`.

# Globální limity v POSIX.1

## sysconf(2)

## Globální limity

```
#include <unistd.h>  
long sysconf(int name);
```

- Globální limity.
- Počet argumentů příkazové řádky.
- Počet dostupných procesorů.
- Velikost stránky.
- Frekvence časovače.
- ... a další.

# Souborové limity v POSIX.1

## pathconf(2)

## Souborové limity

```
#include <unistd.h>  
long pathconf(char *path, int name);  
long fpathconf(int fd, int name);
```

- Limity závislé na souboru.
- Max. počet pevných odkazů.
- Max. délka jména souboru.
- Velikost bufferu roury.
- ... a další.

# POSIX.1 compile-time limity

- ARG\_MAX
- CHILD\_MAX
- PIPE\_BUF
- LINK\_MAX
- \_POSIX\_JOB\_CONTROL
- ... a další.

Run-time limitům definovaným přes `sysconf(2)` a `[f]pathconf(2)` odpovídají i compile-time konstanty.

## Úkol:

Zjistěte a srovnajte POSIX.1 run-time a compile-time limity různých systémů.

# Kapitola 4

## Program v uživatelském prostoru

# Start programu

- **Linkování programu** - crt1.o, objektové moduly, knihovny, libc.a (nebo libc.so).
- **Vstupní bod** - závislý na binárním formátu. Ukazuje obvykle do crt1.o.
- **Mapování sdílených knihoven** - namapování dynamického linkeru do adresového prostoru procesu; spuštění dynamického linkeru.
- **Inicializace** - například konstruktory statických proměnných v C++. V GCC voláno z funkce `__main`.
- **Nastavení globálních proměnných (environ)**.
- **Volání funkce `main()`**.

# Start uživatelského programu

```
main()
```

Vstupní bod programu

```
int main(int argc, char **argv, char **envp);
```

`argc` – počet argumentů programu + 1.

`argv` – pole argumentů.

`envp` – pole proměnných z prostředí procesu  
(*jméno=hodnota*).

- Uložení stavu procesu do `argv[]` – nejčastěji přepsáním `argv[0]`. Nutné u programů, které akceptují heslo na příkazové řádce.
- Platí `argv[argc] == (char *)0`.

# Ukončení programu v C

- Při ukončení procesu je **návratová hodnota** vrácena rodičovskému procesu.
- **8-bitové číslo se znaménkem**
- **0** - úspěšné ukončení.
- **Nenulová hodnota** - chyba.
- **Ukončení procesu** - návrat z `main()` nebo `_exit(2)`.



# Ukončení programu v C

## exit(3)

## Ukončení programu

```
#include <stdlib.h>
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
void exit(int status);
```

- Knihovná funkce.
- Uzavření otevřených souborů (i vylití bufferů stdio).
- Volání statických destruktorků (v C++).
- Ukončení procesu.

# Uživatelský úklid v programu

## atexit(3)

## Vyvolání funkce při exit(3)

```
#include <stdlib.h>
int atexit(void (*function)(void));
```

Zařadí `function()` do seznamu funkcí, které se mají vyvolat při ukončení procesu pomocí `exit(3)`.

# Ukončení procesu

## `_exit(2)`

## Ukončení procesu

```
#include <unistd.h>
void _exit(int status);
```

Služba jádra pro ukončení procesu. Je volána například z knihovní funkce `exit(3)`.

# Násilné ukončení programu

## abort(3)

## Násilné ukončení

```
#include <stdlib.h>  
void abort(void);
```

Ukončí proces zasláním signálu SIGABRT a uloží obraz adresového prostoru procesu do souboru core.



### Úkol:

Napište program, který zavolá nějakou interní funkci, nastaví nějakou svoji proměnnou a zavolá abort(3). Přeložte s ladícími informacemi a spusťte. Debuggerem vyzkoušejte zjistit, ve které funkci a na kterém řádku došlo k havárii a jaký byl stav proměnných.

# Práce s argumenty programu

Bývá zvykem akceptovat přepínače (volby) s následující syntaxí:

- *-písmena* (`ls -lt`).
- *-písmeno argument* (`sed -f x.sed`)
- `--` (ukončení přepínačů)
- *--slovo* (`ls --full-time`).
- *--slovo argument* (`ls --color never`).
- *--slovo=argument* (`ls --color=never`).



## Úkol:

Jak smažete soubor jménem -Z?

# Zpracování přepínačů

## getopt(3)

## Zpracování přepínačů

```
#include <unistd.h>
int getopt(int argc, char **argv,
           char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

## Příklad: getopt(3)

```
while((c=getopt(argc, argv, "ab:--"))!=-1){
    switch (c) {
        case 'a':
            opt_a = 1;
            break;
        case 'b':
            option_b(optarg);
            break;
        case '?':
            usage();
    }
}
```

# Zpracování dlouhých přepínačů

## getopt\_long(3)



```
#include <getopt.h>
int getopt_long(int argc, char * const argv[],
               const char *optstring,
               const struct option *longopts,
               int *longindex);
```

## Knihovna POPT

- Modulární – např. vrstva Glib, GTK+ a GNOME.
- Reentrantní
- Uživatelské aliasy přepínačů



# Zpracování dlouhých přepínačů

## getopt\_long(3)



```
#include <getopt.h>
int getopt_long(int argc, char * const argv[],
    const char *optstring,
    const struct option *longopts,
    int *longindex);
```

## Knihovna POPT


- Modulární - např. vrstva Glib, GTK+ a GNOME.
- Reentrantní
- Uživatelské aliasy přepínačů

# Chybový stav služeb jádra

- Služba jádra – v případě chyby vrací -1 nebo NULL.
- Důvod chyby – v globální proměnné `errno`:

## `errno` Chybová hodnota služby jádra

```
#include <errno.h>  
extern int errno;
```

- Hodnoty konstant v `errno(3)` nebo `<sys/errno.h>` (popř. `<linux/errno.h>` .
- Seznam možných chyb – manpage služby jádra.
- Hodnota `errno` platná jen do příští chyby.

## Příklad: errno

```
retry: if (somesyscall(args) == -1) {
    switch(errno) {
    case EACCES:
        permission_denied();
        break;
    case EAGAIN:
        sleep(1);
        goto retry;
    case EINVAL:
        blame_user();
        break;
    }
}
```

# Textový popis chyby

**perror****Tisk chybového hlášení**

```
#include <stdio.h>
void perror(char *msg);
```

vytiskne zprávu msg a textovou informaci na základě proměnné errno.

## Příklad: perror(3)

```
if (somesyscall(args) == -1) {
    perror("somesyscall() failed");
    return -1;
}
```

Pro ENOENT vypíše následující:

```
somesyscall() failed: No such file or directory
```

# Získání chybové zprávy

## strerror(3)

## Textový popis chyby

```
#include <string.h>
```

```
char *strerror(int errnum);  
int strerror_r(int errnum, char *buf,  
              size_t len);
```

```
extern char *sys_errlist[];  
extern int sys_nerr;
```

# Proměnné prostředí

- Environment variables.
- Pole řetězců tvaru *jméno=hodnota*.
- Třetí argument funkce `main()`
- ... nebo přes globální proměnnou `environ`.

## getenv(3)

## Získání obsahu proměnné

```
#include <stdlib.h>  
char *getenv(char *name);
```

# Nastavení proměnných

## putenv(3), setenv(3)      Nastavení proměnné

```
#include <stdlib.h>
int putenv(char *str);
int setenv(char *name, char *value,
           int rewrite);
```

Argumentem putenv(3) je řetězec tvaru  
*proměnná=hodnota*.

# Rušení proměnných

## `unsetenv(3)`, `clearenv(3)` Rušení proměnných

```
#include <stdlib.h>
int unsetenv(char *name);
int clearenv();
```

`clearenv(3)` není součástí POSIX.1-2001.



### Úkol:

Zjistěte, ve které části adresového prostoru procesu jsou uloženy jeho argumenty a jeho proměnné prostředí. Mění se umístění proměnných, přidáváte-li do prostředí nové proměnné? (Doporučení: použijte formátovací znak `%p` funkce `printf(3)`)



# Alokace paměti

## malloc(3)

## Alokace paměti

```
#include <stdlib.h>
void *malloc(size_t size);
```

- Vrátí ukazatel na nový blok paměti.
- **Velikost:** minimálně size bajtů.
- Ukazatel je zarovnán pro libovolný typ proměnné.

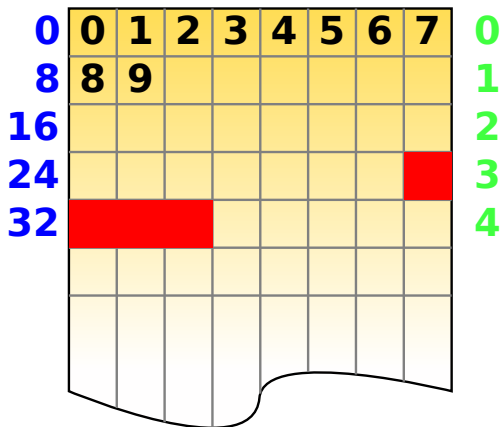
## calloc(3)

## Alokace pole

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

- Místo pro nmemb objektů velikosti size.
- Inicializováno nulami.

# Organizace fyzické paměti



# Alokace paměti

## realloc(3) Změna alokovaného bloku

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

- Změna velikosti dříve alokovaného místa.
- Může přemístit data na jiné místo (nepoužívat původní ukazatel!).

## free(3) Uvolnění dynamické paměti

```
#include <stdlib.h>
void free(void *ptr);
```

- **Pozor:** Některé systémy neakceptují free(NULL).

# Alokace na zásobníku

## alloca(3)

## Alokace na zásobníku

```
#include <alloca.h>
void *alloca(size_t size);
```

- Po ukončení funkce je automaticky uvolněno.
- Specifické pro kompilátor.
- Nelze použít free(3).

## Čtení na dobrou noc

Stack clash - třída bezpečnostních chyb.

<https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt>

# Alokace na zásobníku

## alloca(3)

## Alokace na zásobníku

```
#include <alloca.h>
void *alloca(size_t size);
```

- Po ukončení funkce je automaticky uvolněno.
- Specifické pro kompilátor.
- Nelze použít free(3).



## Čtení na dobrou noc

**Stack clash** - třída bezpečnostních chyb.

<https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt>

# Nízkoúrovňová alokace

## brk(2), sbrk(2) Velikost datového segmentu


```
#include <unistd.h>
int brk(void *end_of_data_segment);
void *sbrk(int increment);
```

- Nastavení velikosti datového segmentu.
- Používáno například funkcemi typu `malloc(3)`.
- Většina implementací `malloc(3)` neumí vrátet uvolněnou paměť zpět operačnímu systému.

# Problémy dynamické paměti

- Častý zdroj chyb
- Uvolnění dříve nealokované paměti.
- Vícenásobné uvolnění.
- Přetečení velikosti.
- Podtečení velikosti.
- Použití i po `realloc(3)`.
- ... problematická detekce.

## Ladící prostředky pro alokátor

- Electric Fence - využívá MMU. I jako `LD_PRELOAD`.
- Valgrind
- Vestavěné kontroly v GNU libc 

# Nelokální skoky

- Podobné jako goto (*OMG, rychle pryč! ^\_~*).
- Ukončení vnořených funkcí.
- Například v případě fatálních chyb.

## setjmp(3)

## Inicializace skoku

```
#include <setjmp.h>  
int setjmp(jmp_buf env);
```

- Inicializuje návratové místo.
- Při prvním volání vrací nulu.



# Volání nelokálního skoku

## longjmp(3)

## Nelokální skok

```
#include <setjmp.h>
void longjmp(jmp_buf env, int retval);
```

- Skok na místo volání setjmp().
- Návrátová hodnota je tentokrát retval.

### Úkol:

Co obsahuje struktura jmp\_buf?

- Struktura jmp\_buf - návratová adresa, vrchol zásobníku.

# Volání nelokálního skoku

## longjmp(3)

## Nelokální skok

```
#include <setjmp.h>
void longjmp(jmp_buf env, int retval);
```

- Skok na místo volání setjmp().
- Návrátová hodnota je tentokrát retval.



### Úkol:

Co obsahuje struktura jmp\_buf?

- Struktura jmp\_buf - návratová adresa, vrchol zásobníku.

# Volání nelokálního skoku

## longjmp(3)

## Nelokální skok

```
#include <setjmp.h>
void longjmp(jmp_buf env, int retval);
```

- Skok na místo volání setjmp().
- Návrátová hodnota je tentokrát retval.



### Úkol:

Co obsahuje struktura jmp\_buf?

- Struktura jmp\_buf - návratová adresa, vrchol zásobníku.

## Příklad: Použití nelokálního skoku

```
#include <setjmp.h>
jmp_buf env;
int main()
{
    if (setjmp(env) != 0)
        dispatch_error();

    ...
    somewhere_else();
}
void somewhere_else()
{
    if (fatal_error)
        longjmp(env, errno);
}
```

# Dynamické linkování

- Přidávání kódu k programu za běhu.
- Sdílené knihovny, plug-iny.
- Knihovna `libdl` (přepínač `-ldl` při linkování).

## dlopen(3) Otevření dynamického objektu

```
#include <dlfcn.h>
void *dlopen(char *file, int flags);
```

- Přidá objekt k procesu.
- Vyřeší křížové odkazy.
- Zavolá symbol `_init` (konstruktory, ...).

Parametr `flags`:

**RTLD\_NOW** - křížové odkazy řešit hned a vrátí chybu, jsou-li nedefinované symboly.

**RTLD\_LAZY** - křížové odkazy se řeší až při použití (jen funkce).

**RTLD\_GLOBAL** - globální symboly dány k dispozici dalším později linkovaným objektům.

Z **RTLD\_NOW** a **RTLD\_LAZY** musí být právě jedno použito.

## `dldclose(3)` Uzavření dynamické knihovny

```
#include <dldfcn.h>  
int dldclose(void *handle);
```

- Počítadlo použití.
- Zavolá symbol `_fini` (destruktory, ...).

## `dldsym(3)` Získání symbolu z knihovny

```
#include <dldfcn.h>  
void *dldsym(void *handle, char *symbol);
```

## `dlderror(3)` Chybové hlášení `libdld`

```
#include <dldfcn.h>  
char *dlderror();
```

## 📄 Příklad: Knihovna libdl

```
#include <dlfcn.h>
#include <stdio.h>
main() {
    void *knihovna = dlopen("/lib/libm.so",
        RTLD_LAZY);
    double (*kosinus)(double) =
        dlsym(knihovna, "cos");
    printf ("%f\n", (*kosinus)(1.0));
    dlclose(knihovna);
}
```



# Úkol: knihovna `libdl`

## ? Úkol: Knihovna `libdl`

Vytvořte následující program:

```
$ callsym knihovna symbol
```

Tento program načte jmenovanou knihovnu a zavolá *symbol* jako funkci bez parametrů.

Doplňte program o testování návratových hodnot funkcí `dlopen` a v případě chyby vypisujte chybové hlášení pomocí `dlderror(3)`.

# Lokalizace

- Přizpůsobení národnímu prostředí.
- Bez **rekompilace** programu.
- Možnost nastavovat na úrovni **uživatele**.
- Možnost nastavovat různé **kategorie**.

# Kategorie lokalizace

**LC\_COLLATE** - třídění řetězců.

**LC\_CTYPE** - typy znaků (písmeno, číslice, nepísmenný znak, převod velká/malá písmena, atd).

**LC\_MESSAGES** - jazyk, ve kterém se vypisují zprávy (viz též GNU gettext).

**LC\_MONETARY** - formát měnových řetězců (znak měny, jeho umístění, počet desetinných míst, atd).

**LC\_NUMERIC** - formát čísla (oddělovač desetin, oddělovač tisícovek apod.)

**LC\_TIME** - formát času, názvy dní v týdnu, měsíců atd.

... a další.

# Názvy locales

*jazyk[\_teritorium][.charset][@modifikátor]*

- **Jazyk** - dle ISO 639 (pro nás cs)
- **Teritorium** - dle ISO 3166 (pro nás CZ)
- **Znaková sada** - například (ISO8859-2 nebo UTF-8)
- **Modifikátor** - například (EURO)



## **Příklad: Názvy locales**

cs\_CZ.UTF-8, cs, cs\_CZ, en\_GB, de@EURO

# Proměnné prostředí

**LANG** - implicitní hodnota pro všechny kategorie.

**LC\_\*** - nastavení jednotlivých kategorií.

**LC\_ALL** - přebíjí výše uvedená nastavení pro všechny kategorie.

# Konfigurace lokalizace

## setlocale(3)

## Nastavení lokalizace

```
#include <locale.h>  
char *setlocale(int category, char *locale);
```

- Pro locale == NULL jen vrátí stávající nastavení.
- Pro locale == "" nastaví hodnotu podle proměnných prostředí.

Po startu programu je nastaveno locale „C“. Program by měl po startu volat následující funkci:

### Příklad: Inicializace locales

```
setlocale(LC_ALL, "");
```

# Lokalizované třídění

## strcoll(3) Porovnávání řetězců podle locale

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Jako strcmp(3), bere ohled na LC\_COLLATE.

## strxfrm(3) Transformace řetězce podle locale

```
#include <string.h>
size_t strxfrm(char *dest,
               char *src, size_t len);
```

- Převede src na dest délky maximálně len.
- Lze porovnávat pomocí strcmp(3).
- Je-li třeba alespoň len znaků, je hodnota dest nedefinována.

# České třídění

## Příklad: Uspořádání podle ČSN

plagiát	plaňka	platno
plachta	plankton	plátno
pláně	plášť	platnost
plánička	plat	
Plánička	plát	

### Úkol:

Napište pomocí `strxfrm(3)` program pro třídění standardního vstupu (podobný programu `sort(1)`).



# České třídění

## Příklad: Uspořádání podle ČSN

plagiát	plaňka	platno
plachta	plankton	plátno
pláně	plášť	platnost
plánička	plat	
Plánička	plát	

## Úkol:

Napište pomocí `strxfrm(3)` program pro třídění standardního vstupu (podobný programu `sort(1)`).

# Katalogy zpráv

- Pro kategorii LC\_MESSAGES.
- GNU `gettext` - překladové tabulky, vyhledávání řetězců.
- Zdrojové soubory: `.po`.
- Zkompilované soubory: `.mo`.

## Příklad: Příklad katalogu zpráv

```
#. ../themes/smaker/theme.jl
msgid "Height of title bar."
msgstr "Výška titulku."
```

# Katalogy zpráv

- Pro kategorii LC\_MESSAGES.
- GNU `gettext` - překladové tabulky, vyhledávání řetězců.
- Zdrojové soubory: `.po`.
- Zkompilované soubory: `.mo`.



## Příklad: Příklad katalogu zpráv

```
#. ../themes/smaker/theme.jl
msgid "Height of title bar."
msgstr "Výška titulku."
```

# Locales v programu v C

## `nl_langinfo(3)` Zjištění informací o locale

```
#include <langinfo.h>  
char *nl_langinfo(nl_item item);
```

item může být jedno z následujících:

`CODESET` - název znakové sady.

`D_T_FMT` - formát data a času (pro `strftime(3)`).

`D_FMT`, `T_FMT`

`DAY_1-7` - název dne v týdnu.

`ABDAY_1-7` - zkratka dne v týdnu.

`MON_1-12`, `ABMON_1-12`

`RADIXCHAR` - oddělovač desetinných míst.

`YESEXPR`, `NOEXPR` .

`CRNCYSTR` - symbol měny a umístění (+, -, .).

# Locales na příkazové řádce

## locale(1) Lokalizačně specifické informace

```
$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
...
LC_ALL=
$ locale -a
aa_DJ
...
zu_ZA.utf8
$ locale charmap
UTF-8
$ locale mon
leden; únor; březen; duben; květen; ...
```

# Znakové sady

## iconv(3)

## Konverze znakových sad

```
#include <iconv.h>
iconv_t iconv_open(char *tocharset,
                  char *fromcharset);
size_t iconv(iconv_t convertor,
             char **inbuf, size_t *inleft,
             char **outbuf, size_t *outleft);
int iconv_close(iconv_t convertor);
```

Název cílového kódování: název znakové sady +  
//TRANSLIT nebo //IGNORE. Viz též iconv(1).



### Úkol:

Napište jednoduchý konvertor z aktuální znakové sady  
(podle locale) do ASCII s transliterací.

# Definice locales

## localedef(8)

## Definice locale

```
$ localedef [-f charmap] [-i inputfile] outdir
```

Vytvoří binární podobu locale pro přímé použití v aplikacích.

# Kapitola 5

# Jádro systému



# Start systému - firmware.

- Uloženo v paměti ROM.
- Na PC odpovídá BIOSu.
- Test hardware.
- Zavedení systému z vnějšího média.
- Často poskytuje příkazový řádek (PROM monitor).
- Sériová konzola?

# Primární zavaděč systému

- Program v boot bloku disku.
- Pevná délka.
- Zavádí sekundární zavaděč.
- Na PC: master boot record – včetně tabulky oblastí.

# Sekundární zavaděč systému

- Načítá jádro.
- Předává jádru parametry.
- Některé poskytují příkazový řádek.
- Některé umí číst souborový systém (možnost bootovat libovolný soubor).
- Používá firmware k zavedení jádra.

# Unified Extensible Firmware Interface

- jednoduchý OS
  - rozpoznání partitions
  - souborový systém (VFAT)
  - formát spustitelného souboru
  - EFI shell
  - EFI proměnné prostředí
  - secure boot
  - Linux: `grub.efi`, `linux.efi`

# Unified Extensible Firmware Interface

- jednoduchý OS
- rozpoznání **partitions**
- souborový systém (VFAT)
- formát spustitelného souboru
- EFI shell
- EFI proměnné prostředí
- secure boot
- Linux: `grub.efi`, `linux.efi`

# Unified Extensible Firmware Interface

- jednoduchý OS
- rozpoznání partitions
- **souborový systém** (VFAT)
- formát spustitelného souboru
- EFI shell
- EFI proměnné prostředí
- secure boot
- Linux: `grub.efi`, `linux.efi`

# Unified Extensible Firmware Interface

- jednoduchý OS
- rozpoznání partitions
- souborový systém (VFAT)
- formát **spustitelného souboru**
- EFI shell
- EFI proměnné prostředí
- secure boot
- Linux: `grub.efi`, `linux.efi`

# Unified Extensible Firmware Interface

- jednoduchý OS
- rozpoznání partitions
- souborový systém (VFAT)
- formát spustitelného souboru
- EFI shell
- EFI proměnné prostředí
- secure boot
- Linux: `grub.efi`, `linux.efi`




# Unified Extensible Firmware Interface

- jednoduchý OS
- rozpoznání partitions
- souborový systém (VFAT)
- formát spustitelného souboru
- EFI shell
- EFI **proměnné prostředí**
- secure boot
- Linux: grub.efi, linux.efi


# Unified Extensible Firmware Interface

- jednoduchý OS
- rozpoznání partitions
- souborový systém (VFAT)
- formát spustitelného souboru
- EFI shell
- EFI proměnné prostředí
- **secure boot**
- Linux: `grub.efi`, `linux.efi`


# Unified Extensible Firmware Interface

- jednoduchý OS
- rozpoznání partitions
- souborový systém (VFAT)
- formát spustitelného souboru
- EFI shell
- EFI proměnné prostředí
- secure boot
- Linux: `grub.efi`, `linux.efi` 

# Start jádra: parametry jádra

- Systémová konzola
- Kořenový disk.
- Parametry pro ovladače zařízení.
- Ostatní parametry předány do uživatelského prostoru.
- `bootparam(7)` 

# Průběh inicializace jádra

- Virtuální paměť – co nejdříve (Linux < 2.0 vs. moduly).
- Inicializace konzoly (někdy dvoufázová: `early_printk()` .
- Inicializace CPU.
- Inicializace sběrnic (autokonfigurovaná zařízení).
- Inicializace zařízení.
- Vytvoření procesu číslo 0 (idle task, swapper, scheduler).
- Start vláken jádra (kflushd, kswapd, ...).
- Inicializace ostatních CPU a start idle procesů.
- Připojení kořenového systému souborů.
- Start procesu číslo 1 – obvykle `/sbin/init`.
- ... dále už uživatelský prostor.

# Inicializace zařízení

- **UNIX v7** – bloková/znaková zařízení, statické tabulky (bdevsw[], cdevsw[]).
- **Linux** – bloková/znaková/SCSI/síťová zařízení, dynamické tabulky.
- **Obsluha zařízení** – funkce pro otevření, čtení, zápis, řídicí operace, atd. Privátní data zařízení.
- **Rekonfigurace za běhu** – hot-plug/hot-unplug (USB apod.).

# Iniciální ramdisk

- Obsah ramdisku načten sekundárním zavaděčem do paměti spolu s jádrem.
- Jádro nemusí mít v sobě žádné ovladače kromě konzoly a souborového systému, který je na ramdisku.
- Inicializace a přilinkování modulů.
- Případné odmontování ramdisku.
- Dále pokračuje start systému připojením kořenového souborového systému a spuštěním `initu`.

# Ramdisk v Linuxu



- Komprimovaný soubor
- Obraz souborového systému nebo `cpio(1)` archív.
- Startovací skript `/linuxrc`.
- Mimo jiné určení kořenového svazku
- Po ukončení - přemontování jako `/initrd` nebo zrušení.



# Bootovací zprávy jádra



- Příkaz `dmesg` (8) 
- Uloženo ve `/var/log/dmesg`, `/var/log/boot.msg` nebo podobně 



## Příklad:

*bootovací zprávy reálného systému*

# Konfigurace jádra

- **System V** konfigurace jádra (/etc/system, /etc/conf/).
- **BSD** konfigurace jádra (/sbin/config, konfigurační soubory, adresáře pro kompilaci). 
- **Linux** – jako jiné programy (používá make). .config nebo /proc/config.gz. 

# Monolitické jádro

- Jeden soubor na disku.
- Všechny používané ovladače jsou uvnitř jádra.
- Často bez autodetekce zařízení.
- Paměť dostupná všem částem jádra stejně.
- Jedna část jádra (ovladač) může rozbít druhou.

# Mikrojaderné systémy

- CMU Mach, OSF Mach, L4, minix, Windows NT HAL, QNX, VxWorks, ...
- Co nemusí být v jádře, dát mimo něj.
- Procesy (*servery*) pro správu virtuální paměti, ovládání zařízení, disků a podobně.
- Dobře definovatelné podmínky činnosti (jen teoreticky, protože: DMA, SMI, IOMMU a další problémy).
- Předávání zpráv - malá propustnost, velká latence.

## Exkurze do historie

Linux is obsolete (1992)

[http://oreilly.com/catalog/opensources/  
/book/appa.html](http://oreilly.com/catalog/opensources/book/appa.html)

# Mikrojaderné systémy

- CMU Mach, OSF Mach, L4, minix, Windows NT HAL, QNX, VxWorks, ...
- Co nemusí být v jádře, dát mimo něj.
- Procesy (*servery*) pro správu virtuální paměti, ovládání zařízení, disků a podobně.
- Dobře definovatelné podmínky činnosti (jen teoreticky, protože: DMA, SMI, IOMMU a další problémy).
- Předávání zpráv - malá propustnost, velká latence.



## Exkurze do historie

Linux is obsolete (1992)

[http://oreilly.com/catalog/opensources/  
/book/appa.html](http://oreilly.com/catalog/opensources/book/appa.html)

# Modulární jádro

- Části (moduly), přidávané do jádra za běhu (odpovídá dynamicky linkovaným knihovnám v uživatelském prostoru).
- Ovladače, souborové systémy, protokoly, ...
- Přidávání ovladačů pouze při startu systému – AIX, Solaris < 10.
- Definovaná rozhraní, nikoliv adresní prostor.

# Modulární jádro v Linuxu



- Dynamické přidávání ovladačů podle potřeby.
- Závislosti mezi moduly (depmod(8)).
- Dynamická registrace ovladačů:  
register\_chrdev(), register\_blkdev(),  
register\_netdev(), register\_fs(),  
register\_binfmt() a podobně.
- Dohledávání pomocí identifikátorů sběrnice (např. PCI ID).


# Procesy v jádře

- Při startu – kontext procesu číslo 0 – později idle task.
- Idle task **nemůže být zablokován** uvnitř čekací rutiny.



## Definice: Kontext

Stav systému, příslušný běhu jednoho procesu/vlákná.

- Přepnutí kontextu – výměna právě běžícího procesu za jiný.
- Linux – struct task\_struct, current 



# Procesy uvnitř jádra



## Otázka:

Pod jakým kontextem mají běžet služby jádra?

- UNIX – použije se kontext volajícího procesu.
- Dva režimy činnosti procesu – user-space a kernel-space.
- Mikrokernel – předá se řízení jinému procesu (serveru).
- Nutno vyřešit přístup do user-space (např. pro `write(2)`).

# Procesy uvnitř jádra



## Otázka:

Pod jakým kontextem mají běžet služby jádra?

- **UNIX** – použije se kontext volajícího procesu.
- **Dva režimy činnosti procesu** – user-space a kernel-space.
- **Mikrokernel** – předá se řízení jinému procesu (serveru).
- Nutno vyřešit přístup do user-space (např. pro `write(2)`).

# Přerušeni

- Žádost o pozornost hardwaru
- **Obsluha** - nepřerušitelná nebo priority.
- **Horní polovina** - co nejkratší, nepřerušitelná. Např. přijetí packetu ze sítě, nastavení vyslání dalšího packetu. Interrupt time.
- **Spodní polovina** - náročnější úkoly, přerušitelné. Obvykle se spouští před/místo předání řízení do uživatelského prostoru. Například: směrování, výběr dalšího packetu k odvysílání. Softirq time.
- **Preemptivní/nepreemptivní jádro** - může dojít k přepnutí kontextu kdekoli v jádře?

# Zpracování přerušeni v jádře



## Otázka:

Pod jakým kontextem lze provádět přerušeni?


- Zvláštní kontext - nutnost přepnutí kontextu → zvýšení doby odezvy (latence) přerušeni. Navíc je nutno případně mít více kontextů pro možná paralelně běžící přerušeni.
- UNIX (ve většině implementací): Přerušeni se provádí pod kontextem právě běžícího procesu. Obsluha přerušeni nesmí zablokovat proces.
- Linux bez samostatného kontextu, uvažuje se o threaded handlers .

# Zpracování přerušení v jádře



## Otázka:

Pod jakým kontextem lze provádět přerušení?

- **Zvláštní kontext** - nutnost přepnutí kontextu → zvýšení doby odezvy (latence) přerušení. Navíc je nutno případně mít více kontextů pro možná paralelně běžící přerušení.
- **UNIX** (ve většině implementací): Přerušení se provádí pod kontextem právě běžícího procesu. Obsluha přerušení nesmí zablokovat proces.
- **Linux** bez samostatného kontextu, uvažuje se o threaded handlers .

# Odložené vykonání kódu

- Funkce, vykonaná později (po návratu z přerušení, při volání scheduleru, atd.)
- Spodní polovina obsluhy přerušení.
- Časově nekritický kód
- Může být přerušen
- Linux – bottom half, tasklety, workqueues, ...

# Virtuální paměť

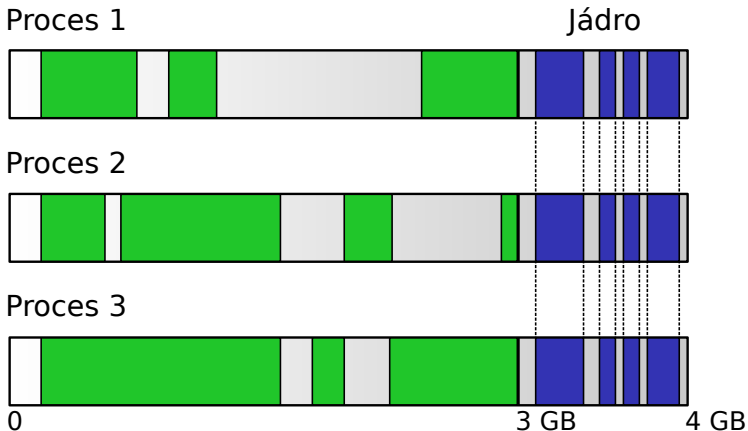
- **Virtuální adresa** – adresa z hlediska instrukcí CPU.
- **Překlad mezi virtuální a fyzickou adresou** – stránková tabulka.
- Každý proces má svoji virtuální paměť: každý proces má svoji stránkovou tabulku.
- **Výpadek stránky** (page fault) – stránka není v paměti, stránkový adresář neexistuje, stránka je jen pro čtení a podobně.
- **Obsluha výpadku stránky** – musí zjistit, jestli jde (například) o copy-on-write, o žádost o natažení stránky z odkládacího prostoru, o naalokování stránky, nebo jestli jde o skutečné porušení ochrany paměti procesem.

# Translation Look-aside Buffer

- TLB – asociativní paměť několika posledních použitých párů (*virtuální adresa, fyzická adresa*).
- Přepnutí kontextu – vyžaduje vyprázdnění TLB, v případě virtuálně adresované cache také vyprázdnění cache.
- Přepnutí mezi vlákny je rychlejší.
- Softwarový TLB – OS-specifický formát stránkových tabulek.
- Lazy TLB switch – uvnitř jádra lze ušetřit.



# Prostor jádra a uživatelský prostor



# Prostor jádra a uživatelský prostor

- **Virtuální paměť jádra** – obvykle mapována na nejvyšších adresách.
- **Paměť jádra** – mapována do **všech procesů** stejně.
- **Přepnutí do režimu jádra** – zpřístupnění horních (virtuálních) adres.
- **Alternativa** – jádro má samostatnou VM (ale: TLB flush při volání jádra nebo přerušení); 4:4 GB split.

# Meltdown

- Postranní kanál úniku informací
- Speklativní vykonávání instrukcí
- Intel - odkládá kontrolu práv na později
- Načte do cache paměť jádra
- Obsah zjistitelný sledováním času
- Obrana - nemít namapované jádro (zpomalení)



Přednáška na dobrou noc ^ \_ ~

Vojtěch Pavlík:

Spectre a Meltdown - Jak fungují a co s nimi

<https://www.youtube.com/watch?v=rwbs-PN0Vpw>

# Meltdown

- Postranní kanál úniku informací
- Speklativní vykonávání instrukcí
- Intel - odkládá kontrolu práv na později
- Načte do cache paměť jádra
- Obsah zjistitelný sledováním času
- Obrana - nemít namapované jádro (zpomalení)



Přednáška na dobrou noc ^ \_ ~

Vojtěch Pavlík:

Spectre a Meltdown - Jak fungují a co s nimi

<https://www.youtube.com/watch?v=rwbs-PN0Vpw>

# Meltdown

- Postranní kanál úniku informací
- Speklativní vykonávání instrukcí
- Intel – odkládá kontrolu práv na později
  - Načte do cache paměť jádra
  - Obsah zjistitelný sledováním času
  - Obrana – nemít namapované jádro (zpomalení)



Přednáška na dobrou noc ^ \_ ~

Vojtěch Pavlík:

Spectre a Meltdown – Jak fungují a co s nimi

<https://www.youtube.com/watch?v=rwbs-PN0Vpw>

# Meltdown

- Postranní kanál úniku informací
- Speklativní vykonávání instrukcí
- Intel - odkládá kontrolu práv na později
- Načte do cache paměť jádra
- Obsah zjistitelný sledováním času
- Obrana - nemít namapované jádro (zpomalení)



Přednáška na dobrou noc ^ \_ ~

Vojtěch Pavlík:

Spectre a Meltdown - Jak fungují a co s nimi

<https://www.youtube.com/watch?v=rwbs-PN0Vpw>

# Meltdown

- Postranní kanál úniku informací
- Speklativní vykonávání instrukcí
- Intel - odkládá kontrolu práv na později
- Načte do cache paměť jádra
- Obsah zjistitelný sledováním času
- Obrana - nemít namapované jádro (zpomalení)



Přednáška na dobrou noc ^ \_ ~

Vojtěch Pavlík:

Spectre a Meltdown - Jak fungují a co s nimi

<https://www.youtube.com/watch?v=rwbs-PN0Vpw>

# Meltdown

- Postranní kanál úniku informací
- Speklativní vykonávání instrukcí
- Intel - odkládá kontrolu práv na později
- Načte do cache paměť jádra
- Obsah zjistitelný sledováním času
- Obrana - nemít namapované jádro (zpomalení)



Přednáška na dobrou noc ^ \_ ~

Vojtěch Pavlík:

Spectre a Meltdown - Jak fungují a co s nimi

<https://www.youtube.com/watch?v=rwbs-PN0Vpw>



# Meltdown

- Postranní kanál úniku informací
- Speklativní vykonávání instrukcí
- Intel - odkládá kontrolu práv na později
- Načte do cache paměť jádra
- Obsah zjistitelný sledováním času
- Obrana - nemít namapované jádro (zpomalení)



## Přednáška na dobrou noc ^\_~

Vojtěch Pavlík:

Spectre a Meltdown - Jak fungují a co s nimi

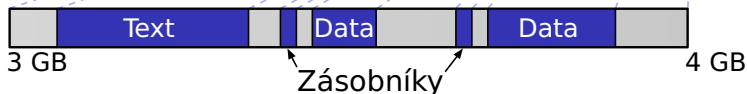
<https://www.youtube.com/watch?v=rwbs-PN0Vpw>

# Virtuální paměť uvnitř jádra

Proces



Prostor jádra



- Zásobník v jádře - pro každý thread/kontext.
- Linux - 1 stránka/thread, nastavitelné 2 stránky/thread 🐧.

# Jádro a fyzická paměť

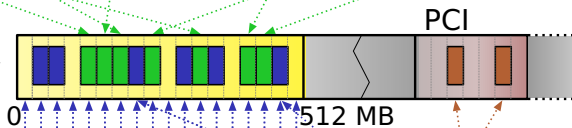
- **Fyzická paměť** – mapována také 1:1 do paměťové oblasti jádra (Linux bez CONFIG\_HIGHMEM).
- **Použití víc než 4 GB paměti na 32-bitových systémech** – Intel PAE, 36-bitová fyzická adresa.
- **Virtuální alokace** – dočasné zpřístupnění fyzické paměti uvnitř jádra.

# Jádro a fyzická paměť

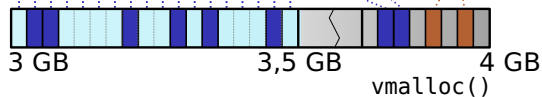
Procesy



Fyzické adresy



Jádro



# Fyzická paměť 32-bitového Linuxu



## Úkol:

Kolik fyzické paměti může obsloužit 32-bitový Linux bez CONFIG\_HIGHMEM, má-li 128 MB vyhrazeno pro virtuální alokace?

# Paměť z hlediska hardwaru

- **Fyzická adresa** – adresa na paměťové sběrnici, vycházející z CPU (0 je to, co CPU dostane, vystaví-li nuly na všechny bity adresové sběrnice).
- **Virtuální adresa** – interní v CPU. Instrukce adresují paměť touto adresou.
- **Sběrnicevá adresa** – adresa místa v paměti tak, jak je vidí ostatní zařízení.
- **IOMMU** – překlad adres mezi sběrnici a operační pamětí. Příklad: AGP GART, AMD Opteron IOMMU.

## Úkol:

K čemu může sloužit IOMMU? Proč mít odlišné fyzické a sběrnicevé adresy?

# Paměť z hlediska hardwaru

- **Fyzická adresa** – adresa na paměťové sběrnici, vycházející z CPU (0 je to, co CPU dostane, vystaví-li nuly na všechny bity adresové sběrnice).
- **Virtuální adresa** – interní v CPU. Instrukce adresují paměť touto adresou.
- **Sběrnicevá adresa** – adresa místa v paměti tak, jak je vidí ostatní zařízení.
- **IOMMU** – překlad adres mezi sběrnici a operační pamětí. Příklad: AGP GART, AMD Opteron IOMMU.



## Úkol:

K čemu může sloužit IOMMU? Proč mít odlišné fyzické a sběrnicevé adresy?

# Přístup do uživatelského prostoru

- **Přístup do user-space:** proces předá jádru ukazatel (např. buffer pro `read(2)`).
- **Robustnost** - user-space nesmí způsobit pád jádra.
- **Validace před použitím?** Problémy ve vícevláknových programech (přístup versus změna mapování v jiném vlákně).

## Úkol:

Přístup do uživatelského prostoru není možný uvnitř ovladače přerušení (proč?).



# Přístup do uživatelského prostoru

- **Přístup do user-space:** proces předá jádru ukazatel (např. buffer pro `read(2)`).
- **Robustnost** - user-space nesmí způsobit pád jádra.
- **Validace před použitím?** Problémy ve vícevláknových programech (přístup versus změna mapování v jiném vlákně).



## Úkol:

Přístup do uživatelského prostoru není možný uvnitř ovladače přerušení (proč?).

# Přístup do user-space v Linuxu



```
status = get_user(result, pointer);
status = put_user(result, pointer);
get_user_ret(result, pointer, retval);
put_user_ret(result, pointer, retval);
copy_user(to, from, size);
copy_to_user(to, from, size);
copy_from_user(to, from, size);
...
```

# Implementace v Linuxu



- **Využití hardwaru CPU** - kontrola přístupu do paměti. Přidání kontroly do `do_page_fault()`.
- **Tabulka výjimek** - adresa instrukce, která může způsobit chybu, opravný kód.
- **Normální běh** - cca 10 instrukcí bez skoku.
- **ELF sekce** - pro generování druhého toku instrukcí.
- Viz též `linux/arch/x86/include/asm/uaccess.h`, např. `__put_user_goto_u64()`.

# Použití uživatelského ukazatele



- Porovnání s `PAGE_OFFSET` (3 GB na 32-bitovém systému). Je-li větší, chyba.
- Použití ukazatele - není-li platný, výjímka CPU.
- Obsluha výjimky - je adresa instrukce v tabulce výjimek? Ano: zavolat opravný kód.
- Jinak: interní chyba jádra (kernel oops).



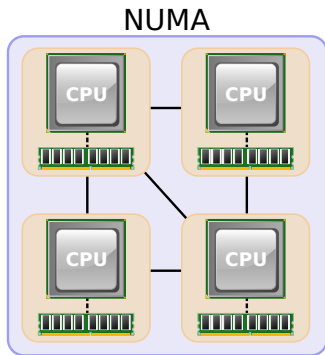
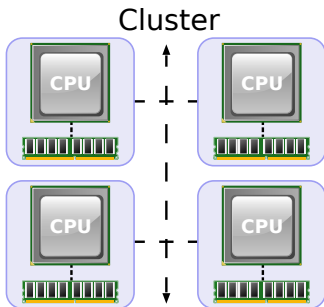
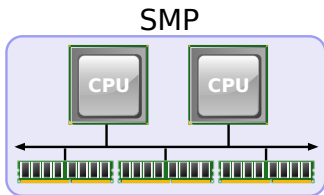
## Problém: volání služby jádra zevnitř jádra

- Nutno předem oznámit. Linux: `set_fs(KERNEL_DS)`
- Například: `net/socket.c: kernel_sendmsg()`.

# Paralelní stroje

- **SMP** – symetrický multiprocessing. Společný přístup všech CPU k paměti.
- **NUMA** – hierarchická paměť – z určitých CPU rychlejší přístup než z jiných (cc-NUMA – cache coherent).
- **Multipočítače** – na částech systému běží zvláštní kopie jádra (clustery a podobně).
- **Problémy** – cache ping-pong, zamykané přístupy na sběrnici, afinity přerušování.


# Paralelní stroje



# Zamykání kódu

- **Paralelismus** – v jednom okamžiku mohou tytéž data modifikovat různé procesy (kontexty).
- **Na jednom CPU** – v kterémkoli okamžiku může být proces přerušen a tentýž kód může provádět i jiný proces.
- **Problém** – manipulace s globálními datovými strukturami (alokace paměti, seznam volných i-uzlů, atd.).

# Zamykání na jednom CPU

- Postačí ochrana proti přerušení
- Zákaz přerušení na CPU - instrukce cli a sti, v Linuxu funkce cli() a sti() 
- Problém - proměnná doba odezvy systému.



# Na paralelním systému

- **Large-grained (hrubozrný) paralelismus** – jeden zámek kolem celého jádra (Linux: `lock_kernel()`, `unlock_kernel()`). Paralelismus možný pouze v uživatelském prostoru. Jednodušší na implementaci, méně výkonný.
- **Fine-grained paralelismus** – zámky kolem jednotlivých kritických sekcí v jádře. Náročnější na implementaci, možnost vzniku netriviálně detekovatelných chyb. Vyšší výkon (několik IRQ může běžet paralelně, několik procesorů zároveň běžících v kernelu).
- **Zamykání v SMP** – nutnost atomických instrukcí (test-and-set) nebo detekce změny nastavené hodnoty (MIPS). Zamčení sběrnice (prefix `lock` na i386).

# Semaforey

- Exkluzivní přístup ke kritické sekci
- Určeno i pro dlouhodobé čekání
- Lze volat pouze s platným uživatelským kontextem
- Linux - `up()`, `down()`, `down_interruptible()`.

# Spinlocky

- Krátkodobé zamykání
- Nezablokuje proces - proces čeká ve smyčce, až se zámek uvolní.
- V Linuxu - `spin_lock_init(lock)`,  
`spin_lock_irqsave(lock)`,  
`spin_unlock_irqrestore(lock)` a podobně.

# R/W zámky

- Paralelní čtení – exkluzivní zápis
- Linux – struct rwlock, struct rwsem.
- Problémy – priority? upgrade r-zámku na w-zámek (deadlock).

# Read-copy-update

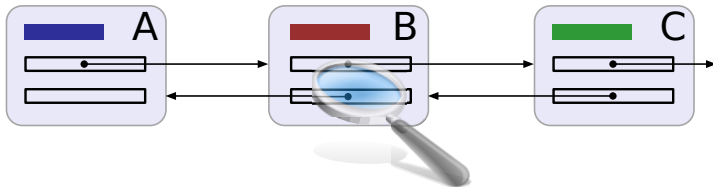
- **RCU** – původně Sequent (Dynix/PTX), později IBM, implementace i v Linuxu.
- **Atomické instrukce** – pomalé (stovky taktů; přístup do hlavní paměti).
- **Obvyklá cesta** (např. čtení) by měla být rychlá.



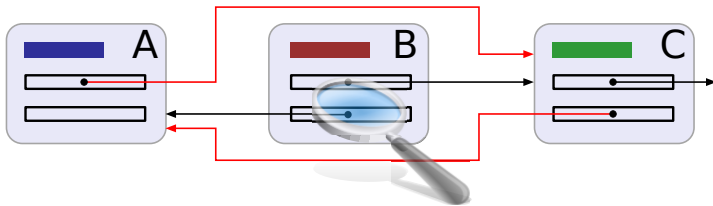
## Princip činnosti RCU

- Vytvoření **kopie** struktury.
- **Publikování** nové verze (změna ukazatele).
- **Uvolnění** původní verze.

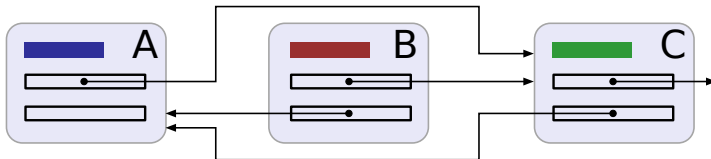
# 📄 Příklad: Seznamy a RCU



# 📄 Příklad: Seznamy a RCU

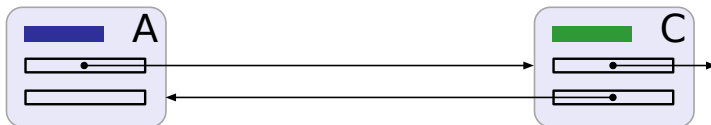


# 📄 Příklad: Seznamy a RCU





# 📄 Příklad: Seznamy a RCU



# Jak implementovat RCU?

- **Slabě uspořádané architektury** – instrukce čtení (nebo i zápisu) mohou být přeuspořádány.
- Nutnost explicitních **paměťových bariér** (speciální instrukce CPU nebo direktivy kompilátoru).
- **Omezující podmínka** – kdy lze uvolnit starou verzi?
- **Linux**: omezující podmínka – přepnutí kontextu na všech procesorech. Odložené vykonání funkce po splnění podmínky.

# Další zdroje informací



## The RCU API, 2019 edition

<https://lwn.net/Articles/777036/>



## Vyzkoušejte v user-space

[https://liburcu.org/membarrier\(2\)](https://liburcu.org/membarrier(2))




## Perfbook2

Paul McKenney:

Is Parallel Programming Hard,  
And, If So, What Can You Do About It?

<https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html>

# Alokátor paměti v jádře

- **Alokace paměti** – z globálních zdrojů (paměť jádra je ve všech procesech stejná).
- **Různé nároky** – malé/velké bloky, požadavek na fyzicky spojitý prostor, zákaz zablokování, atd.
- **Alokace během přerušování** – nesmí uspat proces. Vyhrazený předem uvolněný prostor. Linux: GFP\_ATOMIC .

# Alokace paměti v jádře Linuxu



- **Nejnižší úroveň:** `get_free_pages()`. Alokátor stránek.
- **Malé alokace:** `kmalloc(size, flags)` - alokace do velikosti stránky. Fyzicky souvislá.
- **Větší alokace:** `vmalloc(size, flags)` - zásah do stránkových tabulek, ne nutně fyzicky souvislé.
- **Alokace sběrnicevého prostoru:** `ioremap()`. Na některých architekturách nelze přímý přístup.


# Cache alokovaných objektů

- V jádře: velké množství **stejných objektů** (i-uzly, adresářové položky, hlavičky packetů, ...).

## Problémy velkého množství alokací

- Stejně zarovnání v cache.
- Zbytečné inicializace.
- Studené (cache-cold) objekty.
- Zamykání při alokaci.
- Reakce na tlak ve virtuální paměti.

# SLAB alokátor

- **Objektový alokátor** – Jeff Bonwick (1994), SunOS.
- **Slab** – struktura uvnitř stránky: metadata, objekty.
- **Volné místo** – využito pro cache coloring.
- **Stav slabu** – obsazený, částečně obsazený, volný.
- **VM pressure** – uvolnění volných slabů.
- **Paralelizace** – částečně volný slab pro každý procesor.
- **Cache-cold/hot objekty** – lze specifikovat při alokaci i uvolnění.
- **Konstruktor, destruktork** (volitelné).
- **Linux** – /proc/slabinfo .

# Přednáška na dobrou noc



## LinuxDays 2023

Vlastimil Babka:

Jak jsem smazal slaby (z kernelu)

<https://www.youtube.com/watch?v=UkqIhvBejRI>



# Časovače

- **Časovač** – nutnost vyvolat přerušení po určité době.
- **Atributy** – čas a funkce, která se vyvolá po vypršení času.
- **Funkce v Linuxu** – `add_timer()`, `del_timer()`.
- **Zablokování procesu** – `current->timeout`.

# Čekací fronty

- **Wait queues** - seznam procesů, zablokovaných čekáním na určitou událost (načtení bufferu, dokončení DMA, atd.)
- **Čekající proces** - zařazen do fronty pomocí funkce `sleep_on(q)` nebo `interruptible_sleep_on(q)`.
- **Probuzení procesů** - `wake_up(q)` které zavolá jiný proces nebo IRQ handler. Probudí všechny procesy ve frontě.
- Problém *thundering herd* a `wake_one()`.
- **Přepnutí kontextu** - funkce `schedule()`.

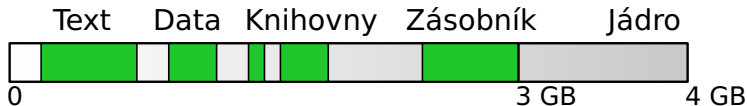
# Kapitola 6

## Procesy

# Procesy

- **Proces** – běžící program.
- **Proces** – kontext procesoru se samostatnou VM.
- **Vlákna (threads)** – kontexty sdílející VM.

# Paměť procesu:



- **Paměť jádra** - přístupná pouze v režimu jádra.
- **Zero page** - zachycení použití neplatných pointerů. U 64-bitových systémů obvykle mezi 0 a 4 GB.
- **Hlavička procesu** - System V (Bach):
  - Záznam v tabulce procesů (viditelný z jádra všem procesům),
  - *u-oblast* - viditelná jen procesu samotnému.
- **Vlákna** - každé má svůj zásobník.

# Atributy procesu

- Čtení – např. programem ps (1).
- Implementace – nad virtuálním souborovým systémem /proc nebo nad /dev/mem.

Atributy procesu jsou:

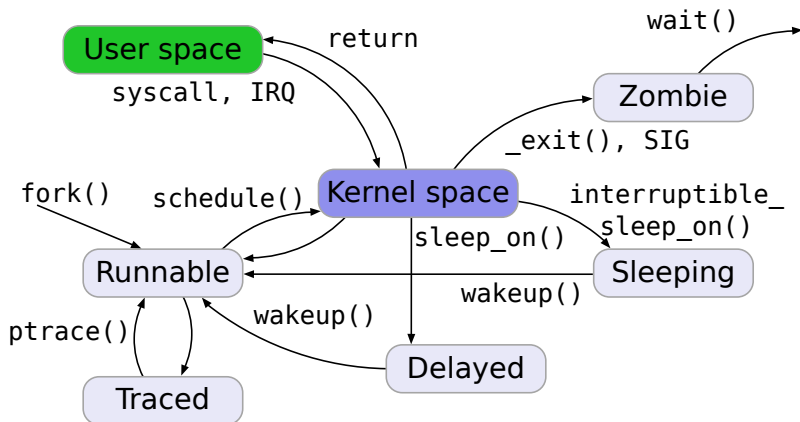
- Stav procesu – viz dále.
- Program counter – čítač instrukcí; místo, kde je proces zablokován (WCHAN).
- Číslo procesu – PID.
- Rodič procesu – PPID (rovno 1, pokud neexistuje).
- Priorita procesu

(pokračování)

# Další atributy procesu

- Vlastník procesu - (real) UID.
- Skupina procesu - (real) GID.
- Skupina procesů, *session* - seskupování procesů do logických celků.
- Reakce na signály, Čekající signály
- Časy běhu
- Pracovní a kořenový adresář
- Tabulka otevřených souborů
- Odkazy na potomky
- *Limity* - na velikost souboru, max. spotřebovaný čas, max. počet otevřených souborů atd (`setrlimit(2)`).

# Stavy procesu





# Služba jádra

- Kód definován v jádře
- Přepnutí oprávnění CPU
- Charakterizována svým číslem
- Glue funkce v knihovně.
- Mechanismus - software interrupt, call gate.
- Nastavení `errno`
- Přerušitelné/nepřerušitelné služby jádra - `EINTR`.
- Druhá kapitola referenční příručky
- `syscall(2)`

# Knihovní funkce

- Kód definován v adresním prostoru procesu
- Lze předefinovat (napsat vlastní funkci)
- Možnost příchodu signálu během provádění
- Nemusí být reentrantní
- Třetí kapitola referenční příručky

# Vznik procesu

## fork(2)

## Vytvoření procesu

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork();
```

- Vytvoří potomka procesu.
- Rodiči vrátí číslo potomka.
- Potomkovi vrátí nulu.

# Potomek versus rodič

Potomek dědí téměř vše od rodiče. Výjimky jsou:

- PID
- PPID
- Zámky na souborech.
- Návratová hodnota `fork(2)`.
- Signál od časovače.
- Čekající signály.
- Hodnoty spotřebovaného strojového času.

# Čekání na ukončení potomka

## wait\*(2)

## Zjištění stavu potomka

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status,
               int options);
```

- Počká na ukončení potomka.
- Pokud je status nenulový ukazatel, uloží do něj informace o změně stavu potomka.

# Informace o stavu potomka


- WIFEXITED(status)** - proces skončil pomocí `_exit(2)`.  
Návratový kód zjistíme pomocí `WEXITSTATUS(status)`.
- WIFSIGNALED(status)** - potomek byl ukončen signálem. Číslo signálu zjistíme pomocí `WTERMSIG(status)`. Navíc SVR4 i 4.3BSD (ale ne POSIX.1) definují makro `WCOREDUMP(status)`, které nabývá hodnoty pravda, byl-li vygenerován core soubor.
- WIFSTOPPED(status)** - proces byl pozastaven. Důvod pozastavení zjistíme makrem `WSTOPSIG(status)`.

# Upřesnění waitpid(2)

Parametr `options` je nula nebo logický součet následujících:

**WNOHANG** - nezablokuje se čekáním.

**WUNTRACED** - i při pozastavení nebo ladění potomka.

**WCONTINUED** - i při znovuspuštění potomka (Linux > 2.6.9 ).

## wait3(2), wait4(2) potomka

## Čekání na ukončení

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
pid_t wait3(int *status, int opts,
            struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int opts,
            struct rusage *rusage);
```

Počká na potomka a zároveň získá informace o jeho využití systémových prostředků. Viz též `getrusage(2)`.



## 📄 Příklad: fork() a wait() - I.

```
switch (pid = fork()) {  
case 0:  
    potomek();  
    break;  
case -1:  
    perror("fork() failed");  
    exit(1);  
default:  
    rodic(pid);  
    break;  
}  
...
```

## Příklad: fork() a wait() - II.

```
potomek() {  
    ...  
    exit(status);  
}  
  
rodic(pid) {  
    int status;  
    waitpid(pid, &status, 0);  
    ...  
}
```

# Spuštění jiného programu

- Parametrem je **spustitelný soubor**.
- Nahradí text (a VM) procesu jiným textem.
- Začne vykonávat nový program.
- **Nevzniká** nový proces!

# Služby jádra třídy exec\*(2,3)

## exec(3)

## Spuštění procesu

```
#include <unistd.h>
extern char **environ;
int execl(char *path, char *arg, ...);
int execlp(char *path, char *arg, ...);
int execl_e(char *path, char *arg, ...,
            char **envp);
int execv(char *path, char **argv);
int execvp(char *path, char **argv);
int execve(char *path, char **argv,
           char **envp);
```

## Další vlastnosti exec\*(2,3)

- Argumenty příkazové řádky: **včetně nultého**. Využití: např. login shell, `ldd(1)`, ...
- Uzavře deskriptory s příznakem `FD_CLOEXEC` (POSIX.1 vyžaduje např. u adresářů).
- Obvykle: `execve(2)` je služba jádra, zbytek knihovní funkce implementované pomocí ní.

# Vyvolání shellu

## system(3)

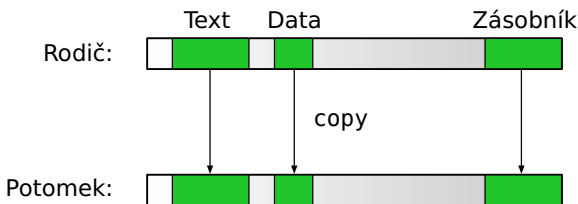
## Vyvolání příkazu shellu

```
#include <stdlib.h>  
int system(char *string);
```

Spustí příkaz `/bin/sh -c string` jako potomka a počká na jeho dokončení.

# fork(2) bez stránkování

- **System bez stránkování** – kopírování celého adresního prostoru.
- Následuje-li `exec(2)`, nový adresní prostor se nahradí.



# Optimalizace vfork(2)



## vfork(2)

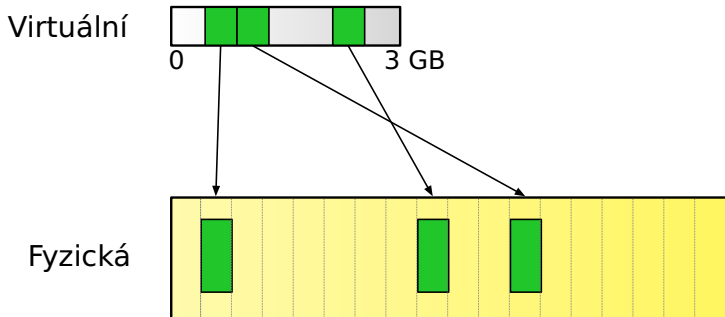
## Virtuální fork()

```
#include <sys/types.h>
#include <vfork.h>
pid_t vfork();
```

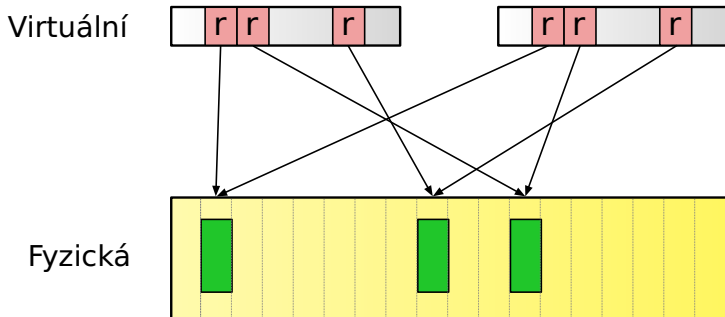
- Vytvoří potomka bez kopírování adresového prostoru.
- Rodič je pozastaven dokud potomek nevyvolá `exec(2)` nebo `_exit(2)`.
- Zavedeno původně jako BSD extenze.



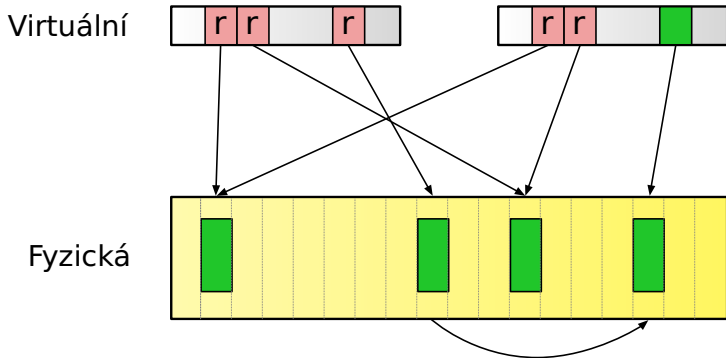
# fork(2) se stránkováním



# fork(2) se stránkováním



# fork(2) se stránkováním



# Systémy se stránkováním

- Unifikovaný systém diskových bufferů a virtuální paměti – sdílení stránek s diskovými buffery – stránka má svůj *obraz v souboru*.
- Sdílení stránek téhož souboru, mapovaných do různých procesů.
- `fork(2)` – sdílení dat mezi rodičem a potomkem, copy-on-write.
- **Sdílené knihovny** – stejný mechanismus (sdílená knihovna = paměťově mapovaný soubor).

# Stránkování na žádost

- Demand-paging
- Text procesu se nenačítá do paměti, pouze se označí, odkud se má načíst.
- Přístup k textu: výpadek stránky; stránka se načte ze souboru.
- Při nedostatku paměti lze přímo zrušit z paměti (bez swapování), později lze novou načíst. **Text file busy**.
- Výhoda - nenačítá se celý text, který se možná ani nevyužije (např. chybné parametry na příkazové řádce).

# I/O operace

- `mmap(2)` – nemusí se načítat soubor do paměti, načtou se jen jednotlivé stránky v případě potřeby.
- `read(2)` – v případě, že čteme do bufferu zarovnaného s velikostí stránky, může systém pouze namapovat (copy-on-write) stránku z buffer cache.

## Otázka:

Je rychlejší zkopírovat stránku nebo ještě jednou namapovat tutéž stránku?

# I/O operace

- `mmap(2)` – nemusí se načítat soubor do paměti, načtou se jen jednotlivé stránky v případě potřeby.
- `read(2)` – v případě, že čteme do bufferu zarovnaného s velikostí stránky, může systém pouze namapovat (copy-on-write) stránku z buffer cache.



## Otázka:

Je rychlejší zkopírovat stránku nebo ještě jednou namapovat tutéž stránku?

# Alokace paměti

- Služba `sbrk(2)` pouze posune konec dat, nealokuje nové stránky.
- Přístup k nově alokovanému prostoru – výpadek stránky, obsluha přidělí novou stránku.
- Výhody – paměť se přiděluje až v okamžiku použití. Viz pole ve Fortranu.



# Memory overcommitment

Má systém počítat, kolik paměti ještě „dluží“ procesům?

- **Ano:** nenastane situace, kdy OS nemůže dostát svým slibům a musí násilně ukončit proces.
- **Ne:** nedojde tak brzo k vyčerpání zdrojů.
- Některé systémy mají možnost nastavit míru overcommitmentu.

## Otázka:

Jak operační systém pozná, že došla paměť?

# Memory overcommitment

Má systém počítat, kolik paměti ještě „dluží“ procesům?

- **Ano:** nenastane situace, kdy OS nemůže dostát svým slibům a musí násilně ukončit proces.
- **Ne:** nedojde tak brzo k vyčerpání zdrojů.
- Některé systémy mají možnost nastavit míru overcommitmentu.



## Otázka:

Jak operační systém pozná, že došla paměť?

# Výhody stránkovacích systémů

- Šetří se systémové zdroje – demand paging, alokace paměti až v případě použití.
- Zvýšení rychlosti – ušetří se kopírování paměti, které je úzkým místem současných počítačů.
- Sdílení paměti – unifikovaný systém VM a diskových bufferů lépe využívá paměť.

# Program na disku

- **Binární formát** – Určuje strukturu souboru, ze kterého se bere text programu
- **Rozpoznání formátu** – magické číslo na začátku souboru. Z user-space příkaz `file(1)`, soubor `/etc/magic`.

# Binární formát *script*

- **Hlavička** – 0x2123 (nebo 0x2321 na big-endian systému). Textová podoba – #!. Následuje jméno (cesta) interpreteru, který se na daný soubor spustí, plus jeho parametry.
- **Příklad** – #!/usr/bin/perl -ne, program v Perlu.
- **Jméno souboru** – předáno interpreteru jako další parametr. Takto lze psát spustitelné soubory i ve formě skriptů, nejen jako binární programy ve strojovém kódu.

# Starší binární formáty

- a.out, x.out, COFF (common object file format)
- Minimálně čtyři sekce – hlavička, text, inicializovaná data, neinicializovaná data (BSS).
- Velikost základních částí – vypisuje program `size(1)`.
- Další sekce – ladící informace, tabulka symbolů, ...

# Binární formát ELF

- Executable and Linking Format
- Stejný formát pro \*.o soubory i pro spustitelné programy.
- Sekce – mají textová jména, lze přidávat další sekce. Lze specifikovat, kam se která sekce má instalovat do paměti.
- Možná rozšíření – několik sekcí pro kód, z jednoho sekvenčního assemblerového textu lze generovat několik sekvencí kódu. Ikona spustitelného souboru, a podobně.

# Přístupová práva procesu

- Pro UID a GID platí podobná pravidla.
- **Reálné** a **efektivní** UID.
- **Saved UID** (pokud je `_POSIX_SAVED_IDS`).
- Většina přístupových práv se prověřuje proti efektivnímu UID.
- Typy `uid_t` a `gid_t`, 16 nebo 32 bitů.



# Čtení přístupových práv

## getuid(2), getgid(2)

## Čtení UID/GID

```
#include <sys/types.h>
#include <unistd.h>
```

```
uid_t  getuid();
uid_t  geteuid();
gid_t  getgid();
gid_t  getegid();
```

# Nastavení přístupových práv

## setuid(2)

## Změna efektivního UID

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

- `uid == 0`: nastaví reálné, efektivní i uložené UID na `uid`.
- Jinak je-li `uid` rovno reálnému nebo uloženému UID, změní pouze efektivní UID na `uid`.
- Jinak končí s chybou `EPERM`.

# Záměna UID

## setreuid(2)

## Výměna r-e UID

```
#include <sys/types.h>
#include <unistd.h>
```

```
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

4.3BSD extenze pro systémy bez uloženého UID/GID.

# Uložené ID

- Pokud je definováno `_POSIX_SAVED_IDS`
- SVR4 podporuje uložená ID.
- FIPS 151-1 vyžaduje tuto vlastnost.
- **Změna reálného UID:** pouze superuživatel.
- **Efektivní UID** je nastaveno službou `exec(2)`, pokud má příslušný program nastavený `set-uid` bit. Jinak se efektivní UID nemění.
- **Uložené UID:** při `exec(2)` se kopíruje z efektivního UID.

## 📄 Příklad: Změny UID procesu

Mějme set-uid program, který patří uživateli číslo 1337 a je spuštěn uživatelem číslo 8086. UID procesu se může měnit například takto:

Akce	reálné	efektivní	uložené
Start programu	8086	1337	1337
setuid(8086)	8086	8086	1337
setuid(1337)	8086	1337	1337
exec()	8086	1337	1337

*nebo:*

setuid(8086)	8086	8086	1337
exec()	8086	8086	8086

# Změna efektivního UID

## seteuid(2) Nastavení efektivního UID

```
#include <sys/types.h>  
#include <unistd.h>
```

```
int seteuid(uid_t uid);  
int setegid(gid_t gid);
```

- Umožní superuživatelskému procesu změnit jen efektivní UID.
- Vyžaduje systém podporující uložená UID.

# Doplňková GID

- **Starší verze UNIXu** – při přihlášení uživatele: UID a GID podle souboru `/etc/passwd`, změna GID pomocí `newgrp(1)`.
- **Novější systémy** – doplňková (supplementary) GID.
- **Zavedeno v 4.2 BSD.**
- Seznam **doplňkových GID** (kromě reálného, efektivního a uloženého).
- **Inicializace** – při přihlášení podle `/etc/group`.
- **Přístupová práva** – efektivní GID a všechna doplňková GID.
- **NGROUPS\_MAX** – limit počtu doplňkových GID.
- **FIPS 151-1** – povinné a `NGROUPS_MAX` aspoň 8.

## getgroups(2)

## Získání doplňkových GID

```
#include <sys/types.h>  
#include <unistd.h>
```

```
int getgroups(int size, gid_t grouplist[]);
```

- Do pole `grouplist[]` uloží doplňková GID až do počtu `size`.
- Vrábí počet skutečně zapsaných položek pole `grouplist[]`.
- Je-li `size` nulové, vrátí počet doplňkových GID pro daný proces.



## setgroups(2)      Nastavení doplňkových GID

```
#include <sys/types.h>  
#include <unistd.h>
```

```
int setgroups(int size, gid_t grouplist[]);
```

Nastaví doplňková GID pro proces. Tuto funkci smí používat pouze superuživatel.

**initgroups(3)****GID podle /etc/group**

```
#include <grp.h>
#include <sys/types.h>
```

```
int initgroups(char *user, gid_t group);
```

- Nastaví doplňková GID podle /etc/group.
- Navíc do seznamu skupin přidá skupinu group.
- Používá se při přihlašování.
- Knihovně funkce - volá setgroups(2).

# Další atributy procesu

## getpid(2), getppid(2)

## Číslo procesu

```
#include <sys/types.h>  
#include <unistd.h>
```

```
pid_t getpid();  
pid_t getppid();
```

Zjištění čísla procesu a čísla rodičovského procesu.

# Systémové zdroje

- **Uživatelský čas** – čas strávený vykonáváním user-space kódu.
- **Systémový čas** – čas strávený vykonáváním služeb jádra.
- **Reálný čas** – čas, který uběhl na hodinách.
- U+S lze počítat i včetně potomků.

## Úkol:

Jaká nerovnost platí pro uživatelský, systémový a reálný čas?

# Systémové zdroje

- **Uživatelský čas** – čas strávený vykonáváním user-space kódu.
- **Systémový čas** – čas strávený vykonáváním služeb jádra.
- **Reálný čas** – čas, který uběhl na hodinách.
- U+S lze počítat i včetně potomků.



## Úkol:

Jaká nerovnost platí pro uživatelský, systémový a reálný čas?

## times(2) Získání časových informací

```
#include <sys/times.h>

clock_t times(struct tms *buf);
struct tms {
    time_t tms_utime;
    time_t tms_stime;
    time_t tms_cutime;
    time_t tms_cstime;
}
```

- Vrací reálný čas od nějakého okamžiku v minulosti.
- Poslední dva údaje jsou včetně potomků.
- Počet tiků systémového časovače.

# Další systémové zdroje

## getrusage(2) Spotřebované systémové zdroje

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int getrusage(int who, struct rusage *r);
```

- Parametr who je buďto `RUSAGE_SELF` nebo `RUSAGE_CHILDREN`.

# Struktura rusage

```
struct timeval ru_utime; /* user time used */
struct timeval ru_stime; /* system time used */
long ru_maxrss; /* maximum resident set size */
long ru_ixrss; /* integral shared memory size */
long ru_idrss; /* integral unshared data size */
long ru_isrss; /* integral unshared stk size */
long ru_minflt; /* page reclaims */
long ru_majflt; /* page faults */
long ru_nswap; /* swaps */
long ru_inblock; /* block input operations */
long ru_oublock; /* block output operations */
long ru_nsignals; /* signals received */
long ru_nvcsw; /* voluntary context switches */
long ru_nivcsw; /* involuntary ctxt switches */
...
```



# Omezení systémových zdrojů

## getrlimit(2), setrlimit(2)

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
int getrlimit(int resource,
              struct rlimit *rlim);
int setrlimit(int resource,
              struct rlimit *rlim);
struct rlimit {
    rlim_t rlim_cur; /* Soft limit */
    rlim_t rlim_max; /* Hard limit */
};
```

- Běžný uživatel - změny soft limitu až do výše hard limitu.

# Typy systémových limitů

Parametr resource může být jeden z následujících:

`RLIMIT_CORE` : velikost souboru core

`RLIMIT_CPU` : strojový čas

`RLIMIT_FSIZE` : velikost vygenerovaného souboru

`RLIMIT_DATA` : velikost datové oblasti

`RLIMIT_STACK` : velikost zásobníku

`RLIMIT_RSS` : resident set size

- většinou neimplementováno – proč?

`RLIMIT_NPROC` : počet procesů daného uživatele

`RLIMIT_NOFILE` : počet otevřených souborů

`RLIMIT_MEMLOCK` : uzamčená paměť

`RLIMIT_AS` : velikost virtuální paměti

# Priorita procesu

- uživatelská priorita (NI) - záměr správce systému
- dispečerská priorita (PRI) - zohlednění interaktivity a podobně



## Čtení na dobrou noc

Fixing SCHED\_IDLE, scheduling classes and policies  
<https://lwn.net/Articles/805317/>

## nice(2)

## Změna priority procesu

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

```
int nice(int inc);
```

Přičte inc k prioritě volajícího procesu. Superuživatel může uvést negativní inkrement.

# Priorita procesu

## getpriority(2)

## Čtení priority procesu

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority(int which, int who);
int setpriority(int which, int who, int pri);
```

Hodnota parametru which je jedna z následujících:

**PRIO\_PROCESS** – priorita procesu.

**PRIO\_PGRP** – priorita skupiny procesů.

**PRIO\_USER** – priorita procesů daného uživatele.

- Je-li who == 0, uvažuje se volající proces, skupina procesů nebo uživatel.
- Viz též [renice\(1\)](#).

# Kooperativní multitasking

## sched\_yield(2) Kooperativní multitasking

```
#include <sched.h>
int sched_yield();
```

Předá řízení jinému procesu, pokud je takový proces k dispozici. **Nepoužívat!**



### Čtení na dobrou noc

Subject: No nuances, just buggy code  
From: Linus Torvalds  
<https://www.realworldtech.com/forum/?threadid=189711&curpostid=189752>

# Skupiny procesů

- Každý proces je v právě jedné skupině.
- V každé skupině je jeden **vedoucí proces**.
- **Číslo skupiny** je číslo vedoucího procesu.
- **Existence skupiny** – dokud má aspoň jednoho člena.
- **Využití:** zasílání signálu, přístup k terminálu (viz `termios(4)`), změna priority, job control.

# Nastavení skupin procesů

## setpgid(2), setpgrp(2) Skupiny procesů

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

```
int setpgid(pid_t pid, pid_t pgid);  
pid_t setpgrp(void);
```

- Je-li pid nebo pgid 0, bere se PID aktuálního procesu.
- setpgrp() je totéž co setpgid(0, 0).

# Čtení skupin procesů

## getpgid(2), getpgrp(2)

## Skupiny procesů

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

```
pid_t getpgid(pid_t pid);  
pid_t getpgrp(void);
```

- Zjistí číslo skupiny procesu (nebo procesu samotného, je-li `pid = 0`).
- `getpgid(0)` je totéž co `getpgrp()`.



# Sessions

- Procesy na jednom terminálu
- V rámci session: více skupin procesů
- Číslo session - číslo vedoucího procesu.

## getsid(2), setsid(2)

```
#include <unistd.h>

pid_t getsid(pid_t pid);
pid_t setsid(void);
```

- **Selže**, je-li proces vedoucím procesem skupiny.

# Démoni

- **Démon** – proces běžící na pozadí bez řídicího terminálu.
- `fork(2)`
- Rodič: `_exit(2)`
- `setsid(2)`
- Pracovní adresář – změnit na `/`.
- Otevřené soubory – uzavřít.
- Std. deskriptory 0, 1 a 2 – otevřít na `/dev/null`.

# Vytvoření démona

## daemon(3)

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

```
int daemon(int nochdir, int noclose);
```

# Kapitola 7

## I/O operace

# I/O operace



## Filozofie fungování UNIXu

Všechno je soubor.

- Soubor – základní jednotka při zpracování I/O operací z pohledu služeb jádra.
- Deskriptor – malé celé číslo – odkaz na otevřený soubor.
- Standardní deskriptory – 0, 1, 2 (POSIX.1: symbolické konstanty `STDIN_FILENO`, `STDOUT_FILENO` a `STDERR_FILENO`).

# I/O operace



## Filozofie fungování UNIXu

Všechno je soubor.

- **Soubor** – základní jednotka při zpracování I/O operací z pohledu služeb jádra.
- **Deskriptor** – malé celé číslo – odkaz na otevřený soubor.
- **Standardní deskriptory** – 0, 1, 2 (POSIX.1: symbolické konstanty `STDIN_FILENO`, `STDOUT_FILENO` a `STDERR_FILENO`).

# Otevření souboru

## open(2), creat(2)

## Otevření souboru

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int open(char *path, int flags);
int open(char *path, int flags, mode_t mode);
int creat(char *path, mode_t mode);
```

flags je jedno z O\_RDONLY, O\_WRONLY, O\_RDWR nebo O\_PATH, plus logický součet některých z konstant:

## Parametry open (2)

`O_CREAT` - vytvoření souboru, pokud neexistuje.

`O_EXCL` - chyba, pokud soubor existuje.

`O_TRUNC` - zarovnání souboru na nulovou délku.

`O_APPEND` - před každým zápisem do souboru je ukazatel pozice v souboru nastaven na konec souboru (jako u `lseek(2)`).

`O_NONBLOCK`, `O_NDELAY` - otevření v neblokujícím režimu.

`O_SYNC` - synchronní výstup.



# Čtení souboru

## read(2)

## Čtení souboru

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

```
ssize_t read(int fd, void *buf, size_t count);
```

- Načte **nejvýše** count bajtů ze souboru do bufferu buf.
- Vrátí -1 v případě chyby,
- 0 na konci souboru,
- jinak počet načtených bajtů.

# Zápis do souboru

## write(2)

## Zápis do souboru

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

```
ssize_t write(int fd, void *buf, size_t count);
```

- Pokusí se zapsat nejvýše count bajtů do souboru.
- Zápis začíná na současné pozici v souboru;
- u souborů otevřených s parametrem O\_APPEND se před zápisem aktuální pozice přesune na konec souboru.

**close(2)****Uzavření deskriptoru**

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

```
int close(int fd);
```

Uzavře deskriptor (a uvolní případné zámky, které proces měl pro tento deskriptor). Uzavření provádí jádro automaticky také při ukončení procesu.

## 📄 Příklad: Kopírování souborů - I.

```
#define BUFSIZE      (1<<14)
char *name1, *name2, *p, buffer[BUFSIZE];
int fd1, fd2, l1, l2;
...
if ((fd1 = open(name1, O_RDONLY)) == -1) {
    perror("Opening input file");
    exit(1);
}
if ((fd2 = open(name2,
                O_WRONLY|O_CREAT|O_TRUNC,
                0666)) == -1){
    perror("Opening output file");
    exit(2);
}
```

## 📄 Příklad: Kopírování souborů - II.

```
while ((l1 = read(fd1,buffer,BUFSIZE)) > 0) {
    for (p=buffer; (l2=write(fd2,p,l1))>0;
        p+=l2)
        if(!(l1 -= l2))
            break;
    if (l2 <= 0) {
        perror("Writing output file");
        exit(3);
    }
}
if (l1 < 0) {
    perror ("Reading input file");
    exit(4);
}
close(fd1);
close(fd2);
```

# Pozice v souboru

## lseek(2) Nastavení pozice v souboru

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

```
off_t lseek(int fd, off_t offset, int odkud);
```

Parametr odkud nabývá těchto hodnot:

**SEEK\_SET** - offset od začátku souboru.

**SEEK\_CUR** - offset od aktuální pozice souboru.

**SEEK\_END** - offset od konce souboru.

- Viz též `llseek(2)`.
- Na některé typy souborů nelze použít `lseek(2)`.
- Ukazatel pozice je společný pro čtení i zápis.

# Příznak O\_APPEND



## Úkol:

Jaký je rozdíl v chování následujících dvou úseků kódu?

# Příznak O\_APPEND

Varianta 1:

```
if ((fd = open(filename, O_WRONLY)) == -1) {
    perror("open");
    exit(1);
}
if (lseek(fd, 0L, SEEK_END) == -1) {
    perror("lseek");
    exit(2);
}
if (write(fd, buffer, size) == -1) {
    perror("write");
    exit(3);
}
```



# Příznak `O_APPEND`

Varianta 2:

```
if ((fd = open(filename, O_WRONLY|O_APPEND))
    == -1) {
    perror("open");
    exit(1);
}
if (write(fd, buffer, size) == -1) {
    perror("write");
    exit(3);
}
```

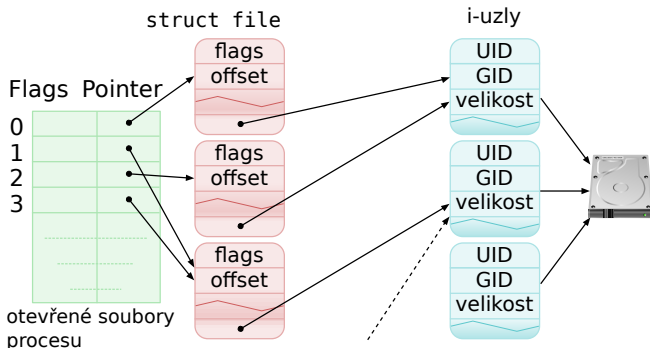
# O\_APPEND a čtení ze souboru



## Úkol:

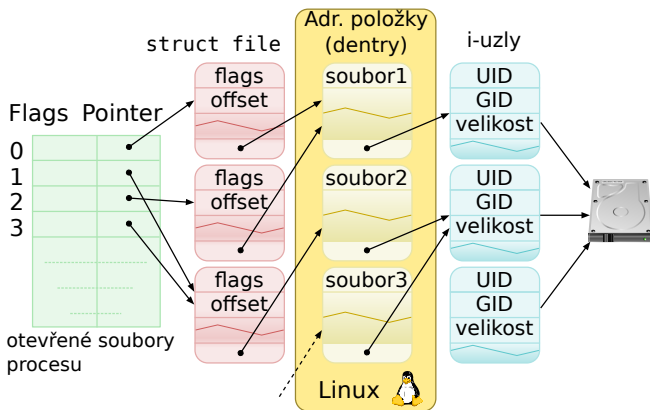
Otevřete-li soubor s `O_RDWR|O_APPEND`, můžete pomocí `lseek(2)` číst data z kteréhokoli místa souboru? A můžete také měnit soubor v kterémkoli jeho místě? Napište program, který toto ověří a pokuste se odhadnout, jakým způsobem je `O_APPEND` flag obsluhován v jádře systému.

# Tabulka otevřených souborů



- BSD říká i-uzlům v paměti `vnode` .

# Tabulka otevřených souborů - Linux



- Linux: cache adresářových položek (**dentry**)



# Duplikování deskriptoru

**dup(2), dup2(2)**

**Duplikace deskriptoru**

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

```
int dup(int oldfd);
```

```
int dup2(int oldfd, int newfd);
```

- Duplikování – nový odkaz na [tutéž](#) strukturu file.
- Použití: přesměrování v shellu.



## Úkol:

Jak se liší funkce následujících dvou úseků kódu?

### Varianta 1:

```
fd1=open("file",O_WRONLY|O_CREAT,0666);  
fd2=dup(fd1);  
write(fd1,"Hello, world\n",13);  
write(fd2,"Hello, world\n",13);  
close(fd1); close(fd2);
```

### Varianta 2:

```
fd1=open("file",O_WRONLY|O_CREAT,0666);  
fd2=open("file",O_WRONLY|O_CREAT,0666);  
write(fd1,"Hello, world\n",13);  
write(fd2,"Hello, world\n",13);  
close(fd1); close(fd2);
```

# Změna vlastností deskriptoru

## fcntl(2) Změna vlastností deskriptoru

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

## Příkazy pro `fcntl(2)`

`F_DUPFD` - duplikuje deskriptor `fd` do `arg`, podobně jako `dup2(2)`.

`F_GETFD` - čte flagy deskriptoru (pouze `FD_CLOEXEC`).

`F_SETFD` - nastavuje flagy deskriptoru (`FD_CLOEXEC`).

`F_GETFL` - čte flagy struktury `file`. Viz druhý parametr `open(2)`.

`F_SETFL` - nastavuje flagy struktury `file`. Lze nastavovat např. `O_APPEND`, `O_NONBLOCK`, `O_ASYNC` a `O_SYNC`. Nikoliv měnit čtení na zápis a naopak.

`F_GETLK`, `F_SETLK` - zamykání souboru (viz dále).



# Práce s I/O zařízeními

**ioctl(2)****Práce s I/O zařízením**

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

```
int ioctl(int fd, int cmd, long arg);
```

- Není v POSIX.1.
- I/O zařízení: reprezentováno souborem.
- Některé operace: nelze převést na čtení/zápis dat (např. SMART informace disku).



## Příklad: ioctl(2)

Nastavení signálu DTR na sériové lince na logickou 1:

```
fd = open("/dev/ttyS0", O_RDWR);  
ioctl(fd, TIOCMGET, &set_bits);  
set_bits |= TIOCM_DTR;  
ioctl(fd, TIOCMSET, &set_bits);
```

# Kapitola 8

## Práce se soubory

# i-uzel

- **i-uzel** (identifikační uzel, inode) je struktura na disku, která popisuje soubor.
- **metadata** souboru.
- Čtení atributů – služba `stat(2)`, příkaz `ls(1)`.

# Atributy i-uzlu

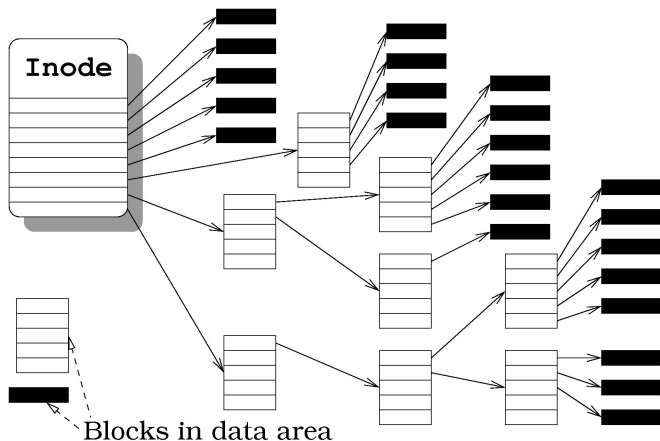
- Délka souboru
- Typ souboru
- UID a GID vlastníka
- Časy – přístupu, modifikace, a změny stavu.
- Přístupová práva
- Počet odkazů – klesne-li na nulu, je i-uzel uvolněn a jeho datové bloky také.
- Odkazy na datové bloky

# Odkazy na datové bloky

Tradiční přístup:

- 13 položek – odkazů na datové bloky.
- Položky 1-10 ukazují přímo na datové bloky.
- Položka 11 ukazuje na blok, kde jsou odkazy na datové bloky (první nepřímý odkaz).
- Položka 12 ukazuje na blok, kde jsou odkazy na bloky odkazů na datové bloky (druhý nepřímý odkaz)
- Položka 13 je třetí nepřímý odkaz.

# i-uzel a datové bloky



# i-uzel a datové bloky - vlastnosti

- Přímý přístup ke kterémukoli místu souboru.
- Díry v souborech - /var/log/lastlog, core.

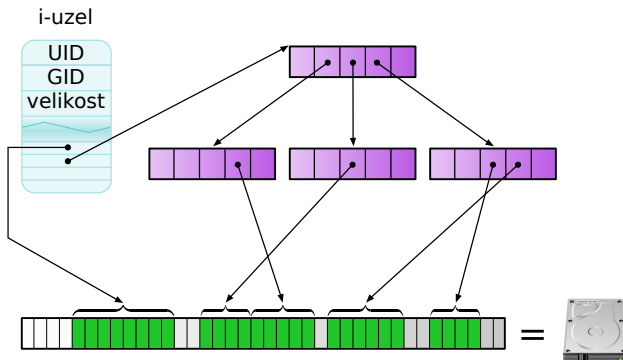


## Úkol:

Má-li souborový systém velikost bloku 1 KB a bloky jsou v i-uzlu indexovány 32-bitovým celým číslem bez znaménka, jaká je maximální teoretická velikost souboru?



# Extent-based souborové systémy



# Čtení atributů i-uzlu

## stat(2)

## Informace o i-uzlu

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
int stat(char *path, struct stat *st);
int lstat(char *path, struct stat *st);
int fstat(int fd, struct stat *st);
```

- Služba lstat(2) neprochází symbolické linky.

# Struktura stat

- `st_dev` - zařízení, na kterém se i-uzel nachází.
- `st_ino` - číslo i-uzlu.
- `st_mode` - typ souboru a přístupová práva.
- `st_nlink` - počet odkazů na i-uzel.
- `st_uid` - vlastník souboru.
- `st_gid` - skupina, které soubor patří.
- `st_rdev` - zde je uloženo hlavní a vedlejší číslo, jde-li o speciální soubor.
- `st_size` - velikost souboru.
- `st_blksize` - preferovaná velikost bloku pro I/O operace.
- `st_blocks` - počet bloků, odkazovaných z i-uzlu (viz soubory s děrami).
- `st_atime`, `st_ctime`, `st_mtime` - čas přístupu, změny stavu, změny obsahu souboru.

# Typy souborů

Typ souboru lze z položky `st_mode` získat těmito makry:

`S_ISREG()` - běžný soubor.

`S_ISDIR()` - adresář.

`S_ISCHR()` - znakový speciální soubor.

`S_ISBLK()` - blokový speciální soubor.

`S_ISFIFO()` - roura nebo pojmenovaná roura.

`S_ISLNK()` - symbolický link.

`S_ISSOCK()` - pojmenovaný socket.

# Přístupová práva souboru

- Přístupová práva lze z `st_mode` získat těmito maskami:
- `S_ISUID`, `S_ISGID`, `S_ISVTX` – set-uid bit, set-gid bit a sticky bit.
  - `S_IRUSR`, `S_IWUSR`, `S_IXUSR` – práva vlastníka souboru.
  - `S_IRGRP`, `S_IWGRP`, `S_IXGRP` – práva skupiny.
  - `S_IROTH`, `S_IWOTH`, `S_IXOTH` – práva ostatního světa.

# Ověření přístupových práv

## access(2)      Ověření přístupových práv

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

```
int access(char *path, int mode);
```

- Ověřuje proti **reálnému** UID/GID.
- Parametr mode - typ přístupu: F\_OK nebo log. součet R\_OK, W\_OK, X\_OK.
- **POZOR:** hrozí časová závislost a bezpečnostní problém!



### Úkol:

Ověřte, jak se služba access(2) chová, je-li argumentem symbolický link, resp. symbolický link ukazující do prázdna.

# Nově vytvářené soubory

- **Vlastník** – podle efektivního UID vytvářejícího procesu.
- Skupina – více možností:
- Podle efektivního GID procesu
  - SystemV
  - Linux
- Podle GID adresáře, ve kterém je soubor vytvářen.
  - BSD
  - Linux, SystemV se set-gid bitem adresáře
  - FIPS 151-1

# Nově vytvářené soubory

- Vlastník – podle efektivního UID vytvářejícího procesu.
- **Skupina** – více možností:
  - Podle efektivního GID procesu
    - SystemV
    - Linux
  - Podle GID adresáře, ve kterém je soubor vytvářen.
    - BSD
    - Linux, SystemV se set-gid bitem adresáře
    - FIPS 151-1



# Nově vytvářené soubory

- Vlastník – podle efektivního UID vytvářejícího procesu.
- Skupina – více možností:
- Podle **efektivního GID** procesu
  - SystemV
  - Linux
- Podle **GID adresáře**, ve kterém je soubor vytvářen.
  - BSD
  - Linux, SystemV se set-gid bitem adresáře
  - FIPS 151-1

# Nově vytvářené soubory

- Vlastník – podle efektivního UID vytvářejícího procesu.
- Skupina – více možností:
- Podle **efektivního GID** procesu
  - SystemV
  - Linux
- Podle **GID adresáře**, ve kterém je soubor vytvářen.
  - BSD
  - Linux, SystemV se set-gid bitem adresáře
  - FIPS 151-1

# Nově vytvářené soubory

**umask(2)****Maska přístupových práv**

```
#include <sys/stat.h>
```

```
int umask(int newmask);
```

- Bity, které jsou v masce nastaveny na 1, se u nově vytvářené souboru **nulují**.
- Vrací předchozí nastavení masky.

## Příklad:

Je-li umask rovno **022**  
a třetí parametr `open(2)` je roven **0776**,  
má výsledný soubor práva  $0776 \& \sim 022 = 0754$ .

# Změna vlastníka souboru

- **Právo měnit** – obvykle jen superuživatel (diskové kvóty).
- **POSIX.1** – volitelné: v době kompilace podle makra `_POSIX_CHOWN_RESTRICTED`, v době běhu pomocí `fpathconf(3)`, resp. `pathconf(3)`.

# Změna skupiny souboru

Skupinu může měnit i běžný proces, pokud jsou splněny zároveň tyto podmínky:

- Efektivní UID procesu je totožné s UID vlastníka souboru.
- Nemění se zároveň s GID také UID vlastníka souboru.
- Nové GID je totožné s efektivním GID procesu nebo s některým z dodatkových GID procesu.



## Poznámka

- Při změně vlastníka/skupiny se nulují set-UID/set-GID bity.
- BSD 4.4 a další – nulují set-UID/set-GID i při zápisu do souboru.

# Změna vlastníka a skupiny

## chown(2) Změna vlastníka/skupiny souboru

```
#include <sys/types.h>
#include <unistd.h>
```

```
int chown(char *path,uid_t owner,gid_t grp);
int lchown(char *path,uid_t owner,gid_t grp);
int fchown(int fd,uid_t owner,gid_t grp);
```

- Je-li owner nebo grp roven -1, neprovádí se změna tohoto údaje.
- lchown(2) je pouze v SVR4. V ostatních systémech mění chown(2) práva symbolického linku.

**Bezpečnost!**

# Změna přístupových práv souboru

**chmod(2)****Změna přístupových práv**

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
int chmod(char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

**Úkol:**

Jakým parametrem nastavíte práva rwsr-xr-- ?

# Změna velikosti souboru

## truncate(2)      Nastavení velikosti souboru

```
#include <sys/types.h>
#include <unistd.h>
```

```
int truncate(char *path, off_t length);
int ftruncate(int fd, off_t length);
```

- Některé systémy nedovolí zvětšit soubor.
- SVR4 implementuje navíc `fcntl(F_FREESP)` – vytvoření díry v již existujícím souboru.



# Změna velikosti souboru

## ? Úkol:

Napište program, který vytvoří soubor s dírou. Vyzkoušejte, které UN\*Xové programy (např. `cp(1)`, `tar/gtar(1)`, `pax(1)`, `cpio(1)`), `rsync(1)` umí takto vytvořený soubor zkopírovat včetně díry.

## ? Úkol:

Zjistěte, které ze tří časů evidovaných v `i`-uzlu se mění při volání `truncate(2)`.

# Pevné linky

## link(2)

## Vytvoření odkazu na i-uzel

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

```
int link(char *path, char *newpath);
```

- Vytvoří další jméno i-uzlu.
- Může skončit s chybou, pokud path a newpath nejsou na tomtéž svazku.

# Smazání souboru

## unlink(2)

## Zrušení odkazu na i-uzel

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

```
int unlink(char *path);
```

- Zruší odkaz na i-uzel (jméno souboru).
- Pokud je počet odkazů na i-uzel nulový, uvolní i-uzel a datové bloky souboru.
- **Pozor:** za odkaz se považuje také run-ime odkaz z tabulky otevřených souborů a podobně.

## Příklad: Anonymní dočasný soubor

```
fd = open("file", O_CREAT|O_RDWR|O_EXCL);  
unlink("file");
```



### Upozornění:

Jen příklad; nutno lépe zvolit jméno souboru!



### Dopředný odkaz pro odvážné ^\_~

Viz též [linkat\(2\)](#) a [O\\_TMPFILE](#).

# Smazání souboru nebo adresáře

## remove(3)

## Zrušení souboru/adresáře

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

```
int remove(char *path);
```

- Smaže soubor nebo adresář.
- Je součástí normy ANSI C.

# Přejmenování souboru

## rename(2) Přejmenování souboru/adresáře

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

```
int rename(char *oldpath, char *newpath);
```

- Atomické přejmenování/přesunutí souboru v rámci jednoho svazku.
- ANSI C definuje jen pro soubory.

### ? Úkol:

Jak funguje mv(1), není-li zdrojové a cílové jméno na tomtéž svazku?

# Časy souboru

## utime(2)

## Nastavení časů souboru

```
#include <sys/types.h>
#include <utime.h>

int utime(char *path, struct utimbuf *times);
struct utimbuf {
    time_t actime;
    time_t modtime;
};
```

- Je-li parametr times NULL, nastaví na aktuální čas.
- Nastavovat smí pouze vlastník souboru (nebo superuživatel).

# Časy souboru - vyšší přesnost

## utimes(2)

## Nastavení časů souboru

```
#include <sys/time.h>
```

```
int utimes(char *path, struct timeval tms[2]);  
struct timeval {  
    long tv_sec;  
    long tv_usec;  
};
```



# Časy souboru - ještě vyšší přesnost ^\_~

## utimensat(2)

## Nastavení časů souboru

```
#include <fcntl.h>
#include <sys/stat.h>

int utimensat(int dirfd, char *path,
              struct timespec times[2], int flags);
int futimens(int fd, struct timespec times[2]);
struct timespec {
    long tv_sec;
    long tv_nsec;
};
```

- tv\_nsec == `UTIME_NOW` - aktuální čas
- tv\_nsec == `UTIME_OMIT` - ignorovat

# Úkol: časy souborů



## Úkol:

Napište program, který nastaví délku zadaného souboru na nulu, ale zachová jeho čas posledního přístupu i modifikace.

# Symbolické linky

## Příklad: Symbolický link

```
$ ls -l /proc/self/fd/0  
lrwx-----. 1 kas staff 64 2009-12-07 12:44 \  
/proc/self/fd/0 -> /dev/pts/17
```

- Symbolický odkaz na soubor pomocí cesty.
- Relativní versus absolutní symbolické linky.
- Uloženo v datovém bloku souboru.
- **Přístupová práva** – obvykle nemají význam.

# Vytvoření symbolického linku

## symlink(2) Vytvoření symbolického linku

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

```
int symlink(char *sympath, char *path);
```

Vytvoří symbolický link path, obsahující řetězec sympath.

# Čtení obsahu symlinku

**readlink(2)****Čtení symbolického linku**

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

```
int readlink(char *path, char *buf, size_t sz);
```

- Přečte obsah symbolického linku.
- Provádí ekvivalent `open(2)`, `read(2)` a `close(2)`.
- Vrací délku symlinku.
- Obsah bufferu není ukončen nulovým znakem.

# Symbolické linky a přístup k souborům



## Služby, neprocházející symlinky

chown(2) (pokud v systému neexistuje `lchown(2)`),  
`lchown(2)`, `lstat(2)`, `readlink(2)`, `rename(2)` a  
`unlink(2)`.

## Úkol

Co bude výsledkem těchto tří příkazů na různých systémech?

```
$ touch ježek  
$ ln -s ježek tučňák  
$ ln tučňák ptakopysk
```

# Symbolické linky a přístup k souborům



## Služby, neprocházející symlinky

chown(2) (pokud v systému neexistuje `lchown(2)`),  
`lchown(2)`, `lstat(2)`, `readlink(2)`, `rename(2)` a  
`unlink(2)`.





## Úkol

Co bude výsledkem těchto tří příkazů na různých systémech?

```
$ touch ježek  
$ ln -s ježek tučňák  
$ ln tučňák ptakopysk
```

# Vytváření dočasných souborů

- Adresář /tmp, /var/tmp.
- Sticky bit
- Exkluzivita
- Bezpečnostní problém se symbolickými linky.
- Linux - O\_CREAT|O\_EXCL.
- O\_NOFOLLOW - , 



# Dočasný soubor z C

## mkstemp(3) Vytvoření dočasného souboru

```
#include <stdlib.h>
```

```
int mkstemp(char *template);
```

### Příklad: Použití mkstemp(3)

```
char *tmpfile = strdup("/tmp/mail.XXXXXX");  
int fd = mkstemp(tmpfile);
```

- Vytvoří dočasný soubor podle dané masky.
- Písmena X - na konci, aspoň 6.
- Vrátí deskriptor, do parametru zapíše skutečné jméno.
- **Nepoužívat:** mktemp(3), tmpnam(3), tempnam(3).

# Dočasný soubor ze shellu

## ✘ Špatně:

```
TMPFILE=/tmp/mujprogram.$$  
ls > $TMPFILE
```

## Správně:

```
TMPFILE='mktmp /tmp/mujprogram.XXXXXX'  
ls > $TMPFILE
```

# Dočasný soubor ze shellu

## ✘ Špatně:

```
TMPFILE=/tmp/mujprogram.$$  
ls > $TMPFILE
```


## ✓ Správně:

```
TMPFILE='mktmp /tmp/mujprogram.XXXXXX'  
ls > $TMPFILE
```

# Adresáře

- **Adresář** – je také soubor
- Obsahuje záznamy tvaru (*název, číslo i-uzlu*).
- Položka „.” – odkaz na sebe.
- Položka „..” – odkaz na nadřazený adresář.
- „..” v kořenovém adresáři ukazuje na sebe.
- **Implementace** – položky „.” a „..” jsou často implementovány na úrovni OS, nikoli nutně fyzicky na disku.
- **Soubor pod více jmény** – ne adresáře (nejasný význam „..” v adresáři).

# Adresáře - pokračování

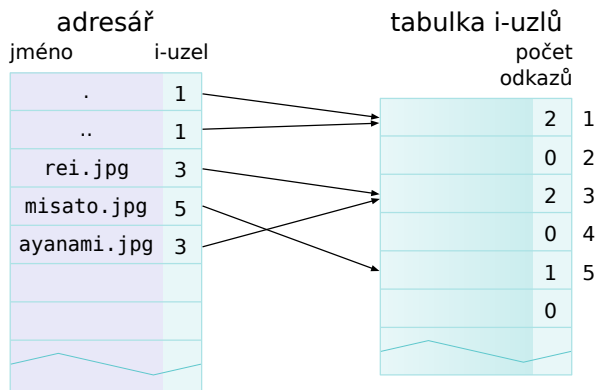
- **Délka jména** - záleží na FS. Původní UNIX: 14, dnes většinou aspoň 252.
- **Délka struktury** - pevná nebo proměnná.
- **Organizace adresáře** - seznam, pole, strom.
- Každý adresář má aspoň dva odkazy (odkud?).
- Sémantika konstrukce „//“.
- Viz též `path_resolution(7)` .



## BTRFS a `st_nlink`

<https://linux-btrfs.vger.kernel.narkive.com/oAoDX89D/btrfs-st-nlink-for-directories>

# Struktura adresáře



# Nový adresář

## mkdir(2)

## Vytvoření adresáře

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
int mkdir(char *path, mode_t mode);
```

- Vytvoří nový prázdný adresář.
- **Práva:** mode + umask(2).

# Zrušení adresáře

**rmdir(2)****Smazání adresáře**


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

```
int rmdir(char *path);
```

- Smaže **prázdný** adresář.
- S adresářem je možné nadále pracovat, má-li jej v této době některý proces otevřený.



# Čtení adresáře

- V některých systémech je možné adresář číst přímo pomocí služby `read(2)`.
- Linux: `O_DIRECTORY`. 
- POSIX.1 definuje přístup k adresáři pomocí následujícího rozhraní:

## `opendir(3)`

## Otevření adresáře

```
#include <sys/types.h>
#include <dirent.h>
```

```
DIR *opendir(char *path);
int closedir(DIR *dp);
```


# Čtení obsahu adresáře

## readdir(3)

## Čtení adresářové položky

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dp);
void rewinddir(DIR *dp);
struct dirent {
    ino_t d_ino;
    char d_name[NAME_MAX+1];
}
```

- POSIX.1 definuje pouze položku d\_name.
- Pořadí jmen souborů závisí na implementaci.
- Linux – služba jádra getdents64(2). 

# Úkol: vlastnosti čtení adresáře



## Úkol:

Napište program, který vypíše obsah adresáře pomocí výše uvedených funkcí. Je pořadí souborů pokaždé stejné? Je výpis setříděn? Jsou vypsány i soubory, začínající tečkou?

# Adresáře procesu

## getcwd(3) Jméno pracovního adresáře

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

```
char *getcwd(char *buf, size_t sz);
```

- Vrátí cestu k pracovnímu adresáři.
- Je-li sz příliš malé, skončí s chybou.
- Pozor na rozdíl mezi `pwd` a `/bin/pwd`.
- Linux – knihovná funkce nad `getcwd(2)`.



# Změna pracovního adresáře

## chdir(2) Změna pracovního adresáře

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

```
int chdir(char *path);
```

```
int fchdir(int fd);
```

- Změní pracovní adresář na zadaný adresář.
- Kontrola přístupových práv.
- Místo getcwd(3) a po čase chdir(2) zpět je lépe použít fchdir(2).
- Proč neexistuje cd(1)?

# Příběh cd(1)



POSIX.2 příkazy musí být exec(2)-ovatelné

```
$ find /neco -type d -exec cd {} \; -print
```

# Kořenový adresář procesu

## chroot(2)

## Kořenový adresář procesu

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

```
int chroot(char *path);
```

- Změní kořenový adresář procesu.
- Povoleno pouze superuživateli.



### Úkol:

Co všechno je nutné k tomu, aby proces mohl „uniknout“ z prostředí se změněným kořenovým adresářem?

# Relativní cesty lokálně



## Problém: pracovní adresář v knihovně

- Přístup sady funkcí (knihovna) k relativně adresovaným souborům
- Nelze použít pracovní adresář
- Hlavní kód může kdykoli volat `chdir(2)`
- Podobně uvnitř vláken



## \*at-slужby jádra

- Předává se adresářový deskriptor
- Relativní cesty vůči **tomuto adresáři**

### \*at(2)

### Adresářově-relativní cesty

```
#include <unistd.h>

int openat(int dirfd, path, flags, mode);
int mkdirat(int dirfd, path, mode);
int unlinkat(int dirfd, path, flags);
...
```

- Pro `dirfd == AT_FDCWD` se použije pracovní adresář procesu.
- `O_PATH` - deskriptor bez I/O operací

# Synchronizace disků

## sync(2)

## Synchronizování disků

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


```
void sync(void);
```

- Zařadí buffery které se mají ukládat na disk do fronty pro okamžitý zápis.
- **Nečeká** na dokončení zápisu.

# Synchronizace deskriptoru

## fsync(2), fdatasync(2)

```
#include <unistd.h>

int fdatasync(int fd);
int fsync(int fd);
int sync_file_range(int fd,
    off_t offset, off_t nbytes,
    unsigned int flags); 
```

- Zapiše všechny modifikované části souboru na disk.
- fdatasync(2) nezapisuje metadata souboru (čas modifikace, ...).

# Garance dat v POSIXu

- **Problém** – patří nadřazený adresář pod „metadata souboru“?
- **Problém** – jak atomicky přepsat soubor?
  - 0\_PONIES ^\_~



Čtení na dobrou noc ^\_~


Valerie Aurora: POSIX v. reality: A position on 0\_PONIES  
<http://lwn.net/Articles/351422/>


# Vytvoření speciálního souboru

## mknod(2)

## Vytvoření souboru

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int mknod(char *path, mode_t mode, dev_t dev);
int mkfifo(char *path, mode_t mode); 
```

- Vytvoří soubor daného jména.
- Parametr mode specifikuje přístupová práva a typ souboru (S\_IFREG, S\_IFCHR, S\_IFBLK nebo S\_IFIFO, viz stat(2)).
- Linux – nelze takto vytvořit adresář. 

# Access Control Lists

- Řízení přístupu pomocí GID – dostatečně silné, ale vyžaduje spoluúčast superuživatele.
- ACL – plné řízení přístupu vlastníkem souboru.
- Seznam položek tvaru `typ:hodnota:[r][w][x]`
- Implicitní položky – typ u, g, o s prázdnou hodnotou.
- Další položky – typ u a g s neprázdnou hodnotou.  
Je-li aspoň jedna takováto položka, je povinná další položka typu m – maska.



# Příklad: ACL

- `u::rwx,g::r-x,o::r--`
- `u::rwx,g::r-x,o::---,\`  
`u:bob:rwx,g:wheel:rw-,m::r-x`

# ACL - vlastnosti

- **Vyhodnocování** – hledá se shoda efektivního UID procesu, pokud se nenalezne, tak efektivní GID a doplňková GID, pokud se ani tady nenalezne, použije se položka o:. U nepovinných položek logický součin s maskou.
- **Omezení** – právě jedna položka od typu u::, g::, o::. Nejvýše jedna položka m::. Nejvýše jeden záznam pro každého uživatele a skupinu.
- **Korespondence s UNIXovými právy** – práva vlastníka souboru = položka u::, práva skupiny souboru = položka m::; není-li, pak g::.
- **Implicitní ACL** – u adresářů. Použije se pro nově vytvářené soubory.
- **Programy** – getfacl(1), setfacl(1), chacl(1). Též acl(5).



# Kapitola 9

## Komunikace mezi procesy

# Roura

- **Datový kanál** – zasílání proudu dat mezi procesy.
- **Implementace** – kruhový buffer velikosti PIPE\_BUF.
- **Čtecí konec, zápisový konec** (deskriptory).

# Nepojmenovaná roura

## pipe(2)

## Vytvoření roury

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

```
int pipe(int fd[2]);
```

- Vrátí dva deskriptory - fd[0] pro čtení a fd[1] pro zápis.
- Využití: zdědění deskriptorů přes fork(2).
- Komunikace mezi **příbuznými** procesy.
- Příklad: operátor „|“ v shellu.

# Pojmenovaná roura

- **Vznik** – službou jádra `mknod(2)`.
- **Otevření** – služba `open(2)` s příslušnou cestou.
- **Vlastnosti** – stejné jako u nepojmenované roury.
- I pro **nesouvisející** procesy.

# Vlastnosti roury

- **Zápis** až do velikosti PIPE\_BUF je atomický.
- **Otevření** (pojmenované) roury pro zápis se zablokuje do doby, než některý jiný proces otevře rouru pro čtení.
- **Čtení** z roury vrátí konec souboru (služba read(2) vrátí nulu), pokud žádný proces nemá otevřený zápisový konec roury a v bufferu nejsou žádná data.
- **Zápis** do roury způsobí zaslání SIGPIPE, nemá-li žádný proces rouru otevřenou pro čtení.

# Příklad použití roury - I.

```
#include <unistd.h>
...
int r, fd[2];
int buf[PIPE_BUF];
...
if (pipe(fd) == -1) {
    perror("pipe()");
    exit(1);
}
```

## Příklad použití roury - II.

```
switch (fork()) {
case -1:
    perror("fork()");
    exit(1);
case 0: /* Potomek */
    close(fd[0]);
    write(fd[1], "Manipulační svěrka\n", 19);
    exit(0);
default: /* Rodič */
    close(fd[1]);
    while ((r = read(fd[0], buf, PIPE_BUF)) > 0)
        write(1, buf, r);
    wait(NULL);
    exit(0);
}
```

# Signály

- **Signál** – asynchronní událost.
- **Reakce** – ignorovat, zachytit ovladačem (*handler*), implicitní akce.
- **Zachycení signálu** – proces začne vykonávat handler.
- **Ukončení handleru** – pokračování od místa přerušení.
- **Zaslání signálu procesem** – práva podle efektivního UID.
- **Zaslání signálu jádrem** – obvykle synchronní odpověď na akci procesu.



# Reakce na signál

## signal(2) Nastavení reakce na signál

```
#include <signal.h>
void (*signal(int sig,
              void (*hndlr)(int)))(int);
nebo jinak:
typedef void SigHandler(int);
SigHandler *signal(int sig, SigHandler *hndlr);
```

- Nainstaluje ovladač signálu.
- Vrátí jeho předešlou hodnotu.
- Speciální hodnoty handleru: `SIG_IGN` (ignore), `SIG_DFL` (default).
- Parametrem ovladače je číslo signálu.

# Zaslání signálu

`kill(2)`, `raise(2)`

Zaslání signálu

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int signo);
int raise(int signo);
```

`pid > 0` zaslán procesu s číslem `pid`.

`pid == 0` zaslán procesům ze stejné skupiny.

`pid < 0` zaslán procesům ze skupiny `abs(pid)`.

`pid == -1` nspecifikovaný výsledek (obvykle všem procesům).

`signo == 0` - jen testuje zaslání signálu (viz [EPERM](#) vs. [ESRCH](#)).

# Čekání na signál

pause(2)

Čekání na signál

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

```
int pause();
```

## Úkol:

Zjistěte, jakou hodnotu errno nastavuje služba jádra pause(2).

# Čekání na signál

pause(2)

Čekání na signál

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

```
int pause();
```

## ? Úkol:

Zjistěte, jakou hodnotu errno nastavuje služba jádra pause(2).

# Dostupné signály - I.

A - ANSI C

P - POSIX.1

J - POSIX.1, systém podporuje job control

S - System V Release 4

B - 4.3BSD

<b>Jméno</b>	<b>Popis</b>	<b>Std.</b>	<b>Akce</b>
SIGABRT	Abnormální ukončení	APSB	core
SIGALRM	Časovač	PSB	ukončení
SIGBUS	Hardwarová chyba	SB	core
SIGCHLD	Změna stavu potomka	JSB	ignorování
SIGCONT	Pokračování po STOP	JSB	znovuspuštění
SIGEMT	Hardwarová chyba	SB	core
SIGFPE	Chyba reálné aritmetiky	APSB	core

# Dostupné signály - II.

Jméno	Popis	Std.	Akce
<b>SIGHUP</b>	Zavěšení linky	PSB	ukončení
<b>SIGILL</b>	Neplatná instrukce	APSB	core
<b>SIGINFO</b>	Získání stavu z terminálu	B	ignorování
<b>SIGINT</b>	Přerušeni z terminálu	APSB	ukončení
<b>SIGIO</b>	Asynchronní I/O	SB	core
<b>SIGIOT</b>	Hardwarová chyba	SB	core
<b>SIGKILL</b>	Ukončení procesu	PSB	ukončení
<b>SIGPIPE</b>	Rouru nikdo nečte	PSB	ukončení
<b>SIGPOLL</b>	Sledovatelná událost	S	ukončení
<b>SIGPROF</b>	Profilovací časovač	SB	ukončení
<b>SIGPWR</b>	Výpadek napájení	S	ignorování
<b>SIGQUIT</b>	Znak Quit na terminálu	PSB	core
<b>SIGSEGV</b>	Chyba segmentace	APSB	core

# Dostupné signály - III.

Jméno	Popis	Std.	Akce
SIGSTOP	Pozastavení procesu	JSB	pozastavení
SIGSYS	Neplatná služba jádra	SB	core
SIGTERM	Výzva k ukončení	APSB	ukončení
SIGTRAP	Hardwarová chyba	SB	core
SIGTSTP	Znak Stop na terminálu	JSB	pozastavení
SIGTTIN	Pokus o čtení z terminálu	JSB	pozastavení
SIGTTOU	Pokus o zápis na terminál	JSB	pozastavení
SIGURG	Urgentní událost	SB	ignorování
SIGUSR1	Uživatelský signál 1	PSB	ukončení
SIGUSR2	Uživatelský signál 2	PSB	ukončení
SIGVTALRM	Virtuální časovač	SB	ukončení
SIGWINCH	Změna velikosti okna	SB	ignorování
SIGXCPU	Překročení strojového času	SB	core
SIGXFSZ	Překročení velikosti souboru	SB	core

# Vlastnosti signálů

- Z hlediska procesu – signál je v podstatě vnější (obvykle asynchronní) přerušení.
- Z hlediska CPU – zasílaný signál neodpovídá žádnému přerušení, některé generované signály odpovídají interním přerušením (exception) CPU.
- Nejsou atomické operace – příchod signálu mezi instalací ovladače a službou pause(2).
- Nespolehlivost – více vygenerovaných signálů může být doručeno jako jeden signál.



# Signály: vylepšené rozhraní

- Vygenerování signálu - v okamžiku volání `kill(2)`.
- Doručení signálu (*delivery*) - vykonání reakce na signál.
- Čekající signál (*pending*) - stav mezi vygenerováním a doručením.
- Blokování signálu - odložení doručení. Signál zůstává ve stavu *pending* dokud proces nezruší blokování nebo nenastaví reakci na ignorování.
- Restartování služeb jádra - místo `EINTR` (přerušitelné služby).

# Množiny signálů

- **Množina signálů** – nový datový typ. Slouží ke změně reakcí na více signálů jednou (atomickou) službou jádra.

## sigsetops(3) Operace nad množinou signálů

```
#include <signal.h>
```

```
int sigemptyset(sigset_t *set);  
int sigfillset(sigset_t *set);  
int sigaddset(sigset_t *set, int signo);  
int sigdelset(sigset_t *set, int signo);  
int sigismember(sigset_t *set, int signo);
```

# Reakce na signál

## sigaction(2)

## Změna reakce na signál

```
#include <signal.h>

int sigaction(int sig, struct sigaction *sa,
              struct sigaction *old);
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
}
```

`sa_handler` může být i SIG\_IGN nebo SIG\_DFL.

`sa_mask` - blokované během provádění handleru.

# Příznaky struktury sigaction

**SA\_NOCLDSTOP** – pro SIGCHLD: ne při pozastavení, jen při ukončení.

**SA\_ONESHOT** (nebo **SA\_RESETHAND**) – jednorázová instalace ovladače. Pak zpět na SIG\_DFL.

**SA\_ONSTACK** – použít alternativní zásobník (viz `sigaltstack(2)`).

**SA\_NOCLDWAIT** – pro SIGCHLD: proces nečeká na potomky a potomci nevytvářejí zombie.

**SA\_NODEFER** (nebo **SA\_NOMASK**) – během provádění ovladače není zablokováno doručení stejného signálu.

**SA\_RESTART** – restartuj případnou přerušitelnou službu jádra namísto chyby EINTR.

**SA\_INFO** – `sa_sigaction` místo `sa_handler`

# Struktura siginfo\_t

```
int      si_signo;
int      si_errno;
int      si_code;          /* Důvod vzniku */
int      si_trapno;       /* HW výjimka */
pid_t    si_pid;
uid_t    si_uid;          /* reálné UID */
int      si_status;       /* retval */
clock_t  si_utime;
clock_t  si_stime;
union sigval si_value;    /* Signal value */
int      si_int;          /* POSIX.1b signal */
void     *si_ptr;         /* POSIX.1b signal */
int      si_overrun;      /* Timer overrun count; */
int      si_timerid;      /* Timer ID; POSIX.1b */
[...]
```

# Zasílání dodatečných dat

## sigqueue(2)

## Zaslání signálu s daty

```
#include <signal.h>

int sigqueue(pid_t pid, int sig,
             union sigval value);
union sigval {
    int    sival_int;
    void *sival_ptr;
};
```

Viz dále POSIX.1b signály.

# Zablokování signálu

## sigprocmask(2)

## Blokování signálů

```
#include <signal.h>
```

```
int sigprocmask(int how, sigset_t *set,  
                sigset_t *old);
```

Hodnota parametru how:

**SIG\_BLOCK** – sjednocení původní množiny a set.

**SIG\_UNBLOCK** – průnik původní množiny a doplňku set.

**SIG\_SETMASK** – nastavení na set.

Jsou-li odblokovány čekající signály, je aspoň jeden doručen před návratem ze sigprocmask(2).

# Dotaz na čekající signály

## sigpending(2)      Dotaz na čekající signály

```
#include <signal.h>
```

```
int sigpending(sigset_t *set);
```

Do množiny set uloží signály, které v daném okamžiku čekají na doručení.



# Čekání na signál

## sigsuspend(2)

## Čekání na signál

```
#include <signal.h>
```

```
int sigsuspend(sigset_t *set);
```

Dočasně nahradí masku blokových signálů za set a zablokuje proces, dokud nepříjde některý z nezablokovaných signálů.

# Synchronní příjem signálu

## sigwait(2)

## Čekání na signál

```
#include <signal.h>
```

```
int sigwait(sigset_t *set, int *sig);  
int sigwaitinfo(sigset_t *set,  
                siginfo_t *info);  
int sigtimedwait(sigset_t *set,  
                 siginfo_t *info, struct timespec *timeout);
```

Zablokuje proces, dokud nepřijde některý ze zadaných signálů. Při příchodu signálu vyplní číslo signálu do sig a **nevolá** handler. Nelze použít na synchronní signály (SIGSEGV, SIGFPE, ...).



## See Also

sigald(2) - čtení signálů z filedeskriptoru



# Real-time signály (POSIX.1b)

- Fronta signálů
- Garantované pořadí doručení
- SIGRTMIN - SIGRTMAX
- `struct siginfo.si_value`
- `_POSIX_SIGQUEUE_MAX`
- `RLIMIT_SIGPENDING`

# Diskuse rozhraní signálů



## Čtení na dobrou noc ^\_~

Neil Brown:

Ghosts of UNIX past, part 3: Unfixable designs

<http://lwn.net/Articles/414618/>

Viz dále `signal(7)`.

# I/O multiplexing

## Příklad: Kopírování dat mezi deskriptory

```
while (!done) {
    if ((n = read(fd1, buf, bufsiz)) <= 0)
        break;

    if (write(fd2, buf, n) < 0)
        error_message("write to fd2");

    if ((n = read(fd2, buf, bufsiz)) <= 0)
        break;

    if (write(fd1, buf, n) < 0)
        error_message("write to fd1");
}
```

# I/O multiplexing

- Jak řešit blokuující I/O operace?
  - polling
  - asynchronní I/O
  - vlákna
  - selektivní čekání (událostně řízené programy)

## Rozhraní pro multiplexing

`select(2)` - původně z BSD

`poll(2)` - System V

`kqueue(2)` - FreeBSD

`/dev/poll` - Solaris

`/dev/epoll`, `eventfd(2)` - Linux

# I/O multiplexing


- Jak řešit blokující I/O operace?
  - polling
  - asynchronní I/O
  - vlákna
  - selektivní čekání (událostně řízené programy)




## Rozhraní pro multiplexing

`select(2)` - původně z BSD

`poll(2)` - System V

`kqueue(2)` - FreeBSD 

`/dev/poll` - Solaris 

`/dev/epoll`, `eventfd(2)` - Linux 

# Množiny deskriptorů

## FD\_\*(3)

```
#include <sys/select.h>

fd_set set;
FD_CLR(int fd, fd_set *pset);
FD_SET(int fd, fd_set *pset);
FD_ISSET(int fd, fd_set *pset);
FD_ZERO(fd_set *pset);
```



# Selektivní čekání

## select(2)

```
#include <sys/select.h>
struct timeval {
    long tv_sec; /* sekundy */
    long tv_usec; /* mikrosekundy */
};
int select(int n, fd_set *rdset, fd_set *wrset,
           fd_set *exset, struct timeval *timeout);
```

- Parametr n - horní limit velikosti množiny deskriptorů.
- Vrací ty deskriptory, kde nastala očekávaná událost.
- Návratová hodnota: počet připravených deskriptorů.

## Příklad: select(2)

```
static int done=0;
int dualcopy(int ttyfd, int modemfd)
{
    fd_set rfdset, wfdset, xfdset;
    char mbuffer[BUFSIZE], tbuffer[BUFSIZE];
    char *mptr, *tptr;
    int mlen=0, tlen=0;
    int nfdset, i;

    int maxfd=1+(ttyfd>modemfd?ttyfd:modemfd);
    while (!done) {
```

## Příklad: select(2)

```
FD_ZERO(&rfd);
FD_ZERO(&wfd);
FD_ZERO(&xfd);
FD_SET(modemfd, &xfd);
FD_SET(ttyfd, &xfd);
if (rlen) FD_SET(ttyfd, &wfd);
    else FD_SET(modemfd, &rfd);
if (tlen) FD_SET(modemfd, &wfd);
    else FD_SET(ttyfd, &rfd);
nfd = select(maxfd, &rfd, &wfd,
            &xfd, NULL);
```

## Příklad: select(2)

```
if (nfds == -1)
    switch(errno) {
    case EBADF:
        printf("invalid fd!\n");
        return -1;
    case EINTR:
        /* Dostali jsme signál. */
        continue;
    case EINVAL:
        printf("Internal error!\n");
        return -3;
    default:
        printf("Invalid errno=%d\n",
            errno);
        return -4;
    }
```

## Příklad: select(2)

```
if (nfds == 0)
    /* Toto se stane jen při timeoutu. */
    continue;
if (FD_ISSET(ttyfd, &xfds)) {
    printf("Exception on tty\n");
    return 0;
}
if (FD_ISSET(modemfd, &xfds)) {
    printf("Exception on modem\n");
    return 0;
}
```

## Příklad: select(2)

```
if (FD_ISSET(ttyfd, &rfdsets)) {
    tlen=read(ttyfd, tbuffer, BUFSIZE);
    tptr=tbuffer;
}
if (FD_ISSET(modemfd, &rfdsets)) {
    mlen=read(modemfd, mbuffer, BUFSIZE);
    mptr=mbuffer;
}
```

## Příklad: select(2)

```
if (FD_ISSET(ttyfd, &wfds)) {
    i=write(ttyfd, mptr, mlen);
    mptr+=i; mlen-=i;
}
if (FD_ISSET(modemfd, &wfds)) {
    i=write(modemfd, tptr, tlen);
    tptr+=i; tlen-=i;
}
}
/* NOTREACHED */ return 0;
}
```

# poll(2) - System V I/O multiplexing

## poll(2)

```
#include <stropts.h>
#include <poll.h>
int poll(struct pollfd fdarray[],
         unsigned long nfd, int timeout_ms);
struct pollfd {
    int fd;
    short events;
    short revents;
};
```

- Samostatné vstupní a výstupní parametry (events, revents).



## poll(2) - vstupní parametry

Parametr events a revents je logický součet některých z následujících hodnot:

**POLLIN** - data jiné než vysoké priority mohou být čtena bez blokování.

**POLLRDNORM** - data normální priority (priorita 0) mohou být čtena bez blokování.

**POLLRDBAND** - data nenulové priority mohou být čtena bez blokování.

**POLLPRI** - data s vysokou prioritou mohou být čtena bez blokování.

**POLLOUT** - data normální priority mohou být zapsána bez blokování.

**POLLWRNORM** - totéž jako POLLOUT.

**POLLWRBAND** - data s vysokou prioritou ( $> 0$ ) mohou být zapsána bez blokování.

## poll(2) - výstupní parametry

Následující typy událostí jsou v revents vraceny vždy bez ohledu na nastavení v events:

**POLLERR** - došlo k chybě na příslušném deskriptoru.

**POLLHUP** - došlo k zavěšení linky.

**POLLNVAL** - deskriptor tohoto čísla není otevřen.

# Kapitola 10

## Pokročilé I/O operace

# Zamykání souborů

- Zamykání pro čtení nebo pro zápis – ve skutečnosti sdílený a výlučný zámek.
- Nepovinné a povinné zamykání – *advisory/mandatory locking*.
  - Nepovinné – jen vzhledem k dalším zámkům.
  - Povinné – i vzhledem k I/O operacím.

## Povinné zamykání

- Původně jen v SVR3 a SVR4.
- POSIX.1 – jen nepovinné.
- Nastavení: set-gid bit na souboru, který není přístupný pro provádění pro skupinu.

# Zamykání souborů

- Zamykání pro čtení nebo pro zápis – ve skutečnosti sdílený a výlučný zámek.
- Nepovinné a povinné zamykání – *advisory/mandatory locking*.
  - Nepovinné – jen vzhledem k dalším zámkům.
  - Povinné – i vzhledem k I/O operacím.



## Povinné zamykání

- Původně jen v SVR3 a SVR4.
- POSIX.1 – jen nepovinné.
- Nastavení: set-gid bit na souboru, který není přístupný pro provádění pro skupinu.

# Vlastnosti zámků

- Zámek přísluší souboru (i-uzlu): přes víc procesů, přes případné hard linky.
- Zámek přetrvá přes volání `exec(2)`.
- **Metody zamykání** – `fcntl(2)` (POSIX.1, SysV, 4.4BSD), `lockf(2)` (SysV), `flock(2)` (BSD).

## Úkol:

Zjistěte, jak se chovají zámky při duplikování deskriptoru pomocí `dup(2)`, a při uzavření některého z takto získaných deskriptorů.

# Vlastnosti zámků

- Zámek přísluší souboru (i-uzlu): přes víc procesů, přes případné hard linky.
- Zámek přetrvá přes volání `exec(2)`.
- Metody zamykání – `fcntl(2)` (POSIX.1, SysV, 4.4BSD), `lockf(2)` (SysV), `flock(2)` (BSD).



## Úkol:

Zjistěte, jak se chovají zámký při duplikování deskriptoru pomocí `dup(2)`, a při uzavření některého z takto získaných deskriptorů.

# Zamykání přes `fcntl(2)`

## `fcntl(2)`

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, long arg);
struct flock {
    short l_type;
    off_t l_start;
    short l_whence;
    off_t l_len;
    pid_t l_pid;
};
```



## Význam příkazů fcntl(2)

- F\_GETLK** - zjistí, jestli je vytvoření zámku blokováno nějakým jiným zámkem. Pokud takový zámek existuje, je struktura flock naplněna popisem tohoto zámku. Pokud neexistuje, je l\_type změněno na F\_UNLCK.
- F\_SETLK** - nastaví nebo zruší zámek. Nemá-li možno zámek vytvořit, vrátí -1 s errno rovno EACCESS nebo EAGAIN.
- F\_SETLKW** - blokující verze F\_SETLK. Pokusí se nastavit zámek. Pokud to není možné, zablokuje se do doby, než bude možné zámek vytvořit nebo než přijde signál.

# Parametry `fcntl(2)`

`cmd` `F_GETLK`, `F_SETLK` nebo `F_SETLKW`

`arg` je ukazatel na strukturu `lock`.

`l_type` - typ zámku:

- `F_RDLCK` - zámek pro čtení,
- `F_WRLCK` - zámek pro zápis,
- `F_UNLCK` - pro zrušení zámku.

`l_start` - offset prvního bajtu zamykaného regionu.

`l_whence` - jedno z `SEEK_SET`, `SEEK_CUR` nebo `SEEK_END` (viz `lseek(2)`).

`l_len` - délka zamykaného regionu. Je-li nulové, značí zámek od `l_start` do konce souboru.

`l_pid` - PID procesu, který drží zámek (vrací `F_GETLK`).

# Scatter-gather I/O

- Čtení do nespojitelného datového prostoru
- Zápis z nespojitelného datového prostoru
- Jedna služba jádra – ušetří se přepnutí do jádra a zpět (nebo kopírování dat do souvislého bufferu).
- Moderní hardware – umí scatter/gather přímo.

# Roztroušené čtení

**readv(2)****scatter read**

```
#include <sys/types.h>
#include <sys/uio.h>

ssize_t readv(int fd, struct iovec iov[],
              int iovcount);
struct iovec {
    void *iov_base;
    size_t iov_len;
};
```

Přečte ze vstupního deskriptoru data do bufferů popsaných v poli struktur `iovec`. Vrací celkový počet přečtených bajtů.

# Sesbíraný zápis

**writev(2)****gather write**

```
#include <sys/types.h>
#include <sys/uio.h>
```

```
ssize_t writev(int fd, struct iovec iov[],
               int iovcount);
```

Zapíše na výstupní deskriptor obsah bufferů popsaných v poli `iov[]`. Vrací celkový počet zapsaných bajtů.

# Memory-mapped I/O

- Mapování (části) souboru do paměti
- Sdílená paměť mezi programy
- Urychlení I/O - ušetří se kopírování dat.
- Alokace paměti - namapování /dev/zero.
- Nevýhoda - zneplatnění TLB při změně mapování.

# Mapování souboru do paměti

## mmap(2)

## Mapování souboru do paměti

```
#include <unistd.h>
#include <sys/mman.h>

#ifdef _POSIX_MAPPED_FILES
void *mmap(void *start, size_t length,
           int prot, int flags, int fd,
           off_t offset);
#endif
```

Parametr prot:

**PROT\_EXEC** - stránky mohou být prováděny.

**PROT\_READ** - stránky jsou přístupné pro čtení.

**PROT\_WRITE** - do stránek lze zapisovat.

**PROT\_NONE** - ke stránkám nelze přistupovat.

## Přepínače pro mmap(2)

**MAP\_FIXED** - zakazuje jádru zvolit jinou adresu pro mapování než start. Parametr start pak musí být zarovnan na velikost stránky (viz `sysconf(2)`). Používání této volby se nedoporučuje z důvodu přenositelnosti.

**MAP\_SHARED** - zápis do mapované oblasti se projeví v namapovaném souboru i v paměti dalších procesů, které si tento úsek souboru namapovaly.

**MAP\_PRIVATE** - mapovaná oblast je copy-on-write kopií obsahu mapovaného souboru. Změny provedené procesem se neprojeví jinde.

Služba `mmap(2)` vrací v případě úspěchu ukazatel na první bajt namapovaného regionu.



# Odmapování souboru

## munmap(2)

## Odmapování souboru

```
#include <unistd.h>
#include <sys/mman.h>

#ifdef _POSIX_MAPPED_FILES
int munmap(void *start, size_t length);
#endif
```

Další přístup k odmapované části je neplatný (způsobí zaslání signálu SIGBUS nebo SIGSEGV).

# Synchronizace paměti

## msync(2)

## Synchronizace regionu

```
#include <unistd.h>
#include <sys/mman.h>

#ifdef _POSIX_MAPPED_FILES
#ifdef _POSIX_SYNCHRONIZED_IO
int msync(const void *start, size_t length,
           int flags);
#endif
#endif
```

## Parametry msync (2)

`MS_SYNC` – služba počká na dokončení zápisu na disk.

`MS_ASYNC` – služba nastartuje zápis, ale skončí bez čekání na dokončení diskových operací.

`MS_INVALIDATE` – zruší platnost namapovaných stránek tohoto souboru, takže stránky jsou případně znovu načteny z diskové kopie.

Právě jedno z `MS_SYNC` a `MS_ASYNC` musí být nastaveno.

# Přístupová práva paměťového regionu

## mprotect(2)      Nastavení přístupu k regionu

```
#include <sys/mman.h>
```

```
int mprotect(const void *addr, size_t len,  
             int prot);
```

- Parametr prot stejný jako u mmap(2).
- Parametr addr musí být zarovnan na velikost stránky.
- POSIX.4 (1b) říká, že mprotect(2) může být použito pouze na regiony získané pomocí mmap(2).

# Zákaz swapování

## mlock(2), munlock(2)

```
#include <sys/mman.h>
```

```
int mlock(const void *addr, size_t len);  
int munlock(const void *addr, size_t len);
```

- Nedědí se přes fork(2) a exec(2).
- Využití - real-time aplikace, kryptografické aplikace.

# Zamčení celé virtuální paměti

```
mlockall(2), munlockall(2)
```

```
#include <sys/mman.h>
```

```
int mlockall(int flags);
```

```
int munlockall();
```

**MCL\_CURRENT** - jen ty stránky, které jsou momentálně namapovány.

**MCL\_FUTURE** - i regiony mapované v budoucnu budou zamčeny.

# Vlastnosti `mlock(2)`, `mlockall(2)`

Vícenásobné zamčení těžé stránky se zvláště nepočítá. K odemčení stačí jediné `munlockall(2)` nebo `munlock(2)`.



## Úkol:

Jak předalokovat dostatečně velký prostor na zásobníku?

# Kapitola 11

## Vlákna



# Vícevláknové aplikace

- Vlákno - thread - *light-weight process*.
- Kontext činnosti procesoru - podobně jako procesy.
- Vlákna jednoho procesu
  - stejná VM
  - jiný zásobník
  - stejné deskriptory
  - stejný pracovní adresář
  - a další sdílené atributy
- Vlákna v UNIXu - IEEE POSIX 1003.1c (POSIX threads).

# Vlákna: pro a proti

- Proč vlákna?
  - využití více procesorů
  - paralelizace diskových operací
- Kdy ne vlákna – tam kde lze použít událostně řízené programování (GUI aplikace) nebo samostatné procesy (síťové servery).

Čtení na dobrou noc ^\_~

John Ousterhout: Why Threads Are A Bad Idea (for most purposes).

# Vlákna: pro a proti

- Proč vlákna?
  - využití více procesorů
  - paralelizace diskových operací
- Kdy ne vlákna – tam kde lze použít událostně řízené programování (GUI aplikace) nebo samostatné procesy (síťové servery).



Čtení na dobrou noc ^\_~

John Ousterhout: Why Threads Are A Bad Idea (for most purposes).

# Kontexty jádra versus vlákna

- **1:N** (user-level threads) – například balík pthreads.
- **1:1** (kernel-level threads) – LinuxThreads a NPTL.
- **M:N** – kombinace obojího – např. scheduler activations ve FreeBSD; IRIX.

# Vytvoření vlákna

## pthread\_create(3)

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

```
int pthread_create(pthread_t *thread,  
                  pthread_attr_t *attr,  
                  void (*start_routine)(void *),  
                  void *arg);
```

Vytvoření vlákna. Identifikace vlákna je uložena do thread. Parametr attr určuje další vlastnosti vlákna. Nastavuje se následujícími funkcemi: pthread\_attr\_init(3), pthread\_attr\_destroy(3), pthread\_attr\_setdetachstate(3), pthread\_attr\_setschedparam(3) a další.

# Ukončení vlákna

## pthread\_exit(3)

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

```
void pthread_exit(void *retval);
```

Ukončí vlákno, zavolá registrované funkce pro dobu ukončení a uvolní lokální data vlákna. Viz též `pthread_cleanup_push(3)` a další funkce `pthread_cleanup_*(3)`.

# Čekání na ukončení vlákna

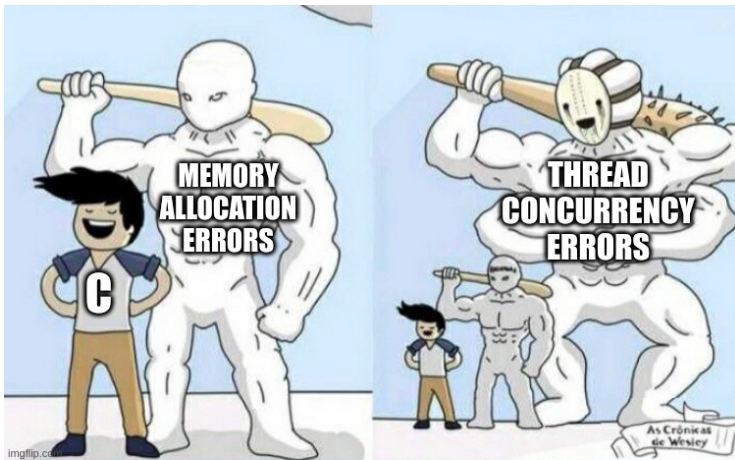
## pthread\_join(3)

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

```
int pthread_join(pthread_t tid, void **retval);
```

Počká na ukončení vlákna a získá návratovou hodnotu.

# Synchronizace vláken





# Synchronizace vláken

- **Mutex** - vzájemné vyloučení vláken.
- **Stavy** - odemčený zámek/zamčený zámek.
- **Vzájemné vyloučení** - v jednu chvíli může držet zámek zamčený nejvýše jedno vlákno.

# Vlákna a mutexy

## pthread\_mutex\_\*(2)

```
pthread_mutex_t fmutex =
    PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t rmutex =
    PTHREAD_RECURSIVE_MUTEX_INITIALIZER;
pthread_mutex_t emutex =
    PTHREAD_ERRORCHECK_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mtx,
    pthread_mutexattr_t *attr);

int pthread_mutex_lock(pthread_mutex_t *mtx);
int pthread_mutex_trylock(pthread_mutex_t *mtx);
int pthread_mutex_unlock(pthread_mutex_t *mtx);
int pthread_mutex_destroy(pthread_mutex_t *mtx);
```

# Podmínkové proměnné

- **Podmínková proměnná** - hlášení o události jinému vláknou.
- **Strany komunikace** - vlákno čeká na podmínku, vlákno signalizuje podmínku.
- Podmínková proměnná má asociovaný zámek.
- **Čekání na podmínku** - atomické odemčení mutexu a zablokování vlákna. Mutex musí být předem zamčený. Při ukončování funkce se mutex opět zamče.

# Podmínkové proměnné - použití

## pthread\_cond\_\*(3)

```
pthread_cond_t c = PTHREAD_COND_INITIALIZER;  
int pthread_cond_wait(pthread_cond_t *c,  
    pthread_mutex_t *mutex);  
int pthread_cond_timedwait(pthread_cond_t *c,  
    pthread_mutex_t *mutex,  
    struct timespec *abstime);  
int pthread_cond_signal(pthread_cond_t *c);  
int pthread_cond_broadcast(pthread_cond_t *c);  
int pthread_cond_destroy(pthread_cond_t *c);
```

# Soukromá data vlákna

- **Globální proměnná** – ale v každém vlákně s jinou hodnotou.
- **Důvod použití** – není nutno předávat jako argument do všech funkcí.
- **Klíč** – konkrétní kus dat, v každém vlákně s jinou hodnotou.
- **Destruktor** – při ukončení vlákna se volá pro jeho nenulové klíče.

# Soukromá data vlákna - použití

## pthread\_\*specific(3)

```
pthread_key_t list_key;
extern void* cleanup_list(void*);
pthread_key_create(&list_key, cleanup_list);

int* p_num = (int *)malloc(sizeof(int));
(*p_num) = 4; /* Nejaka hodnota */
pthread_setspecific(list_key, (void *)p_num);

/* Nekde uplne jinde */
int* p_keyval = (int*)
    pthread_getspecific(list_key);

/* a nakonec */
pthread_key_delete(list_key);
```

# Vlákna - další vlastnosti

- **Zrušení vlákna** - `pthread_cancel(3)`. Vlákno může být zrušitelné jen v některých bodech.
- **Odpojení vlákna** - `pthread_detach(3)`. Není pak možno/nutno vlákno připojovat přes `pthread_join(3)`.
- **Jednorázová inicializace** - `pthread_once(3)`. Zavolání pouze při prvním použití.
- **Identifikace vlákna** - `pthread_self(3)`.

# Vlákna a signály

- Zaslání signálu - `pthread_kill(3)`.
- Cekání na signál - `sigwait(3)`.
- Synchronní signál - doručen vláknu které signál vygenerovalo.
- Asynchronní signál - doručen některému vláknu, které signál neblokuje.
- Maska blokováných signálů - pro každé vlákno zvlášť - viz `pthread_sigmask(3)`.



# Vlákna a soubory

- Ukazovátka pozice – globální pro strukturu file.
- Problematický přístup z více vláken.

## pread(3)

## Čtení na dané pozici

```
ssize_t pread(int fd, void *buf,  
              size_t count, off_t offset);
```

## pwrite(3)

## Zápis na danou pozici

```
ssize_t pwrite(int fd, void *buf,  
               size_t count, off_t offset);
```

- Ukazovátka pozice se zde nemění.

# Kapitola 12

## SystemV/POSIX IPC

# SystemV IPC

- Prostředky pro komunikaci mezi procesy
  - Semaforey
  - Fronty zpráv
  - Sdílená paměť
- Pojem BSD IPC = sockety
- Existují nezávisle na procesech
- Mají přístupová práva
- POSIX IPC: evoluce SystemV IPC

# SystemV IPC

- Prostředky pro komunikaci mezi procesy
  - Semaforey
  - Fronty zpráv
  - Sdílená paměť
- Pojem **BSD IPC** = sockety
  - Existují nezávisle na procesech
  - Mají přístupová práva
  - POSIX IPC: evoluce SystemV IPC

# SystemV IPC

- Prostředky pro komunikaci mezi procesy
  - Semaforey
  - Fronty zpráv
  - Sdílená paměť
- Pojem BSD IPC = sockety
- Existují nezávisle na procesech
  - Mají přístupová práva
  - POSIX IPC: evoluce SystemV IPC

# SystemV IPC

- Prostředky pro komunikaci mezi procesy
  - Semaforey
  - Fronty zpráv
  - Sdílená paměť
- Pojem BSD IPC = sockety
- Existují nezávisle na procesech
- Mají přístupová práva
- POSIX IPC: evoluce SystemV IPC

# SystemV IPC

- Prostředky pro komunikaci mezi procesy
  - Semaforey
  - Fronty zpráv
  - Sdílená paměť
- Pojem BSD IPC = sockety
- Existují nezávisle na procesech
- Mají přístupová práva
- **POSIX IPC**: evoluce SystemV IPC

# Jak identifikovat IPC?

**ftok(3)****Klíč pro IPC**

```
#include <sys/types.h>  
#include <sys/ipc.h>
```

```
key_t ftok(const char *path, int proj_id);
```

- POSIX varianty mají identifikaci cestou k souboru



# SystemV semaforey

- **Semafor**: nezáporné celé číslo
- Nesmí klesnout pod nulu
- Blok semaforů

# SystemV semaforey

- Semafor: nezáporné celé číslo
- Nesmí klesnout pod nulu
- Blok semaforů

# SystemV semaforey

- Semafor: nezáporné celé číslo
- Nesmí klesnout pod nulu
- **Blok** semaforů

# Získání bloku semaforů

**sem\*(2)****Vytvoření bloku semaforů**

```
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
int semget(key_t key, int nsems, int flags);
```

- **key** - IPC\_PRIVATE nebo ftok(3).
- **flags** - IPC\_CREAT, IPC\_EXCL

# Operace se semaforem

## semop(2)

## Operace se semaforey

```
int semop(key_t key,
          struct sembuf *sops, size_t nsops);
int semtimedop(key_t key,
              struct sembuf *sops, size_t nsops,
              struct timespec *timeout);
struct sembuf {
    unsigned short sem_num;
    short sem_op;
    short sem_flg; // SEM_UNDO, IPC_NOWAIT
}
```