

# Programování v UNIXu

(NSWI015)

verze: 12. prosince 2016

(c) 2016 – 2017 Vladimír Kotal, Jan Pechanec

(c) 2011 – 2015 Vladimír Kotal

(c) 2005 – 2011 Jan Pechanec

(c) 1999 – 2004 Martin Beran

SISAL MFF UK, Malostranské nám. 25, 118 00 Praha 1



# Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

# Obsah

- **úvod, vývoj UNIXu a C, programátorské nástroje**
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

## Literatura v češtině

1. Skočovský, L.: **Principy a problémy operačního systému UNIX**. Science, 1993
2. Skočovský, Luděk: **UNIX, POSIX, Plan9**. L. Skočovský, Brno, 1998
3. Jelen, Milan: **UNIX V - programování v systému**. Grada, Praha 1993
4. **Linux - Dokumentační projekt**. Computer Press, 1998;  
<http://www.cpress.cz/knihy/linux>
5. Herout, Pavel: **Učebnice jazyka C**. 2 díly. Kopp, České Budějovice, 2004 (4., respektive 2. přepracované vydání)

# Literatura - design a principy systému

1. Uresh Vahalia: **UNIX Internals: The New Frontiers**. Prentice Hall; 1st edition, 1995
2. Bach, Maurice J.: **The Design of the UNIX Operating System**. Prentice Hall, 1986
3. McKusick, M. K., Neville-Neil, G. V.: **The Design and Implementation of the FreeBSD Operating System**. Addison-Wesley, 2004
4. McDougall, R.; Mauro, J.: **Solaris Internals**. Prentice Hall; 2nd edition, 2006.
5. **Linux Documentation Project**. <http://tldp.org/>

# Literatura - programování

1. Stevens, W. R., Rago, S. A.: **Advanced Programming in UNIX(r) Environment**. Addison-Wesley, 2nd edition, 2005.
2. Rochkind, M. J.: **Advanced UNIX Programming**, Addison-Wesley; 2nd edition, 2004
3. Stevens, W. R., Fenner B., Rudoff, A. M.: **UNIX Network Programming, Vol. 1 – The Sockets Networking API**. Prentice Hall, 3rd edition, 2004
4. Butenhof, D. R.: **Programming with POSIX Threads**, Addison-Wesley; 1st edition, 1997
5. unixové specifikace, viz <http://www.unix.org>
6. manuálové stránky (zejm. sekce 2, 3)

## Literatura - historie UNIXu

- Peter Salus: **A Quarter Century of UNIX**, Addison-Wesley; 1st edition (1994)
  - Libes D., Ressler, S.: **Life With Unix: A Guide for Everyone**, Prentice Hall (1989)
  - **Open Sources: Voices from the Open Source Revolution**, kapitola **Twenty Years of Berkeley Unix From AT&T-Owned to Freely Redistributable**; O'Reilly (1999); on-line na webu:  
<http://oreilly.com/openbook/opensources/book/index.html>
- ... mnoho materiálů na webu; často však obsahující ne zcela přesné informace

# (Pre)historie UNIXu

- 1925 – **Bell Telephone Laboratories** – výzkum v komunikacích (např. 1947: transistor) v rámci AT&T
- 1965 – BTL s General Electric a MIT vývoj OS **Multics** (MULTIplexed Information and Computing System)
- 1969 – Bell Labs opouští projekt, **Ken Thompson** píše assembler, základní OS a systém souborů pro PDP-7
- 1970 – Multi-cs  $\Rightarrow$  Uni-cs  $\Rightarrow$  Uni-x
- 1971 – UNIX V1, a portován na PDP-11
- prosinec 1971 – první edice *UNIX Programmer's Manual*



# Historie UNIXu, pokračování

- únor 1973 – UNIX V3 obsahoval *cc* překladač (jazyk C byl vytvořen **Dennisem Ritchiem** pro potřeby UNIXu)
- říjen 1973 – UNIX byl představen veřejnosti článkem *The UNIX Timesharing System* na konferenci ACM
- listopad 1973 – **UNIX V4 přepsán do jazyka C**
- 1975 – UNIX V6 byl první verzí UNIXu běžně k dostání mimo BTL
- 1979 – UNIX V7, pro mnohé “the last true UNIX”, obsahoval *uucp*, Bourne shell; velikost kernelu byla pouze 40KB !!!
- 1979 – UNIX V7 portován na 32-bitový VAX-11
- 1980 – Microsoft přichází s XENIXem, který je založený na UNIXu V7

# Divergence UNIXu

- pol. 70. let – uvolňování UNIXu na univerzity: především **University of California v Berkeley**
- 1979 – z UNIX/32V (zmíněný port na VAX) poskytnutého do Berkeley se vyvíjí **BSD Unix (Berkeley Software Distribution)** verze 3.0; poslední verze 4.4 v roce 1993
- 1982 **AT&T**, vlastník BTL, může vstoupit na trh počítačů (zakázáno od roku 1956) a přichází s verzí *System III* (1982) až *V.4* (1988) – tzv. *SVR4*
- vznikají UNIX International, OSF (Open Software Foundation), X/OPEN, ...
- 1991 – Linus Torvalds zahájil vývoj OS Linux, verze jádra 1.0 byla dokončena v r. 1994

# Současné UNIXy

Hlavní komerční unixové systémy:

- Sun Microsystems: **SunOS** (není již dále vyvíjen), **Solaris**
- Apple: **Mac OS X**
- SGI: **IRIX**
- IBM: **AIX**
- HP: **HP-UX**, **Tru64 UNIX** (Compaq)
- SCO: **SCO Unix**
- Novell: **UNIXware**

Open source:

- **FreeBSD**, **NetBSD**, **OpenBSD**
- **Linux** distribuce

# Standardy UNIXu

- **SVID** (System V Interface Definition)
  - „fialová kniha“, kterou AT&T vydala poprvé v roce 1985
  - dnes ve verzi SVID3 (odpovídá SVR4)
- **POSIX** (Portable Operating System based on UNIX)
  - série standardů organizace IEEE značená P1003.xx, postupně je přejímá vrcholový nadnárodní orgán ISO
- **XPG** (X/Open Portability Guide)
  - doporučení konsorcia X/Open, které bylo založeno v r. 1984 předními výrobci platforem typu UNIX
- **Single UNIX Specification**
  - standard organizace The Open Group, vzniklé v roce 1996 sloučením X/Open a OSF
  - dnes Version 3 (**SUSv3**), předchozí Version 2 (**SUSv2**)
  - splnění je nutnou podmínkou pro užití obchodního názvu UNIX

# POSIX

- tvrzení “tento systém je POSIX kompatibilní” nedává žádnou konkrétní informaci
  - asi podporuje POSIX1990 a možná i něco dalšího (co?)
- dotyčný buď neví co je POSIX nebo si myslí, že to nevíte vy
- jediná rozumná reakce je otázka “jaký POSIX?”
- POSIX je **rodina standardů**
- prvním dokumentem je *IEEE Std POSIX1003.1-1988*, později po vzniku dalších rozšíření neformálně odkazovaný jako POSIX.1
- poslední verze POSIX.1 je *IEEE Std 1003.1, 2004 Edition*
  - obsahuje v sobě již i to, co dříve definoval POSIX.2 (Shell and Utilities) a různá, dříve samostatná rozšíření

# Jazyk C

- téměř celý UNIX je napsaný v C, pouze nejnižší strojově závislá část v assembleru ⇒ poměrně snadná přenositelnost
- navrhl Dennis Ritchie z Bell Laboratories v roce 1972.
- následník jazyka B od Kena Thomsona z Bell Laboratories.
- vytvořen jako prostředek pro přenos OS UNIX na jiné počítače – silná vazba na UNIX.
- varianty jazyka:
  - původní K&R C
  - standard ANSI/ISO C
- **úspěch jazyka C daleko přesáhl úspěch samotného UNIXu**

# Formáty dat

- pořadí bajtů – závisí na architektuře počítače

big endian: 0x11223344 = 

11	22	33	44	
addr +	0	1	2	3

little endian: 0x11223344 = 

44	33	22	11	
addr +	0	1	2	3

- řádky textových souborů končí v UNIXu znakem **LF** (nikoliv CRLF). Volání `putc('\n')` tedy píše pouze jeden znak.
- big endian – SPARC, MIPS, síťové pořadí bajtů
- little endian – Intel

# Deklarace a definice funkce

- K&R

- deklarace

- `návratový_typ identifikátor();`

- definice

- `návratový_typ identifikátor(par [,par...])`

- `typ par;...`

- `{ /* tělo funkce */ }`

- ANSI

- deklarace

- `návratový_typ identifikátor(typ par [,typ par...]);`

- definice

- `návratový_typ identifikátor(typ par [,typ par...])`

- `{ /* tělo funkce */ }`



# C style

- věc zdánlivě podřadná, přitom extrémně důležitá – úprava zdrojových kódů programu
- mnoho způsobů jak ano:

```
int
main(void)
{
    char c;
    int i = 0;

    printf("%d\n", i);
    return (0);
}
```

## C style (cont.)

- mnoho způsobů jak **NE** (tzv. assembler styl):

```
int main(void) {  
    int i = 0; char c;  
    printf("%d\n", i);  
    return (0);  
}
```

- nebo (schizofrenní styl):

```
int main(void) {  
    int i = 0; char c;  
    if (1)  
        printf("%d\n", i);i=2;  
    return (0);  
}
```

# Utility

<b>cc, c99<sup>*</sup>, gcc<sup>†</sup></b>	překladač C
<b>CC, g++<sup>†</sup></b>	překladač C++
<b>ld</b>	spojovací program (linker)
<b>ldd</b>	pro zjištění závislostí dynamického objektu
<b>cxref<sup>*</sup></b>	křížové odkazy ve zdrojových textech v C
<b>sccs<sup>*</sup>, rcs, cvs</b>	správa verzí zdrojového kódu
<b>make<sup>*</sup></b>	řízení překladač podle závislostí
<b>ar<sup>*</sup></b>	správa knihoven objektových modulů
<b>dbx, gdb<sup>†</sup></b>	debuggery
<b>prof, gprof<sup>†</sup></b>	profilery

\* SUSv3 † GNU

# Konvence pro jména souborů

- `*.c` jména zdrojových souborů programů v C
  - `*.cc` jména zdrojových souborů programů v C++
  - `*.h` jména hlavičkových souborů (headerů)
  - `*.o` přeložené moduly (object files)
  - `a.out` jméno spustitelného souboru (výsledek úspěšné kompilace)
- 
- `/usr/include` kořen stromu systémových headerů
  - `/usr/lib/lib*.a` statické knihovny objektových modulů
  - `/usr/lib/lib*.so` umístění dynamických sdílených knihoven objektových modulů

# Princip překladač

zdrojové moduly  
main.c

```
main()
{
  msg();
}
```

util.c

```
msg()
{
  puts();
}
```

překladač

objektové moduly  
main.o

```
main
msg ??
```

util.o

```
msg
puts ??
```

linker

program  
a.out

```
main
msg
msg
puts
puts
```

systemová  
knihovna

```
puts
```

# Překlad jednoho modulu (preprocesor)

main.c

```
#include <stdio.h>
#include "mydef.h"
main()
{
    puts(MSG);
}
```

/usr/include/stdio.h

```
int puts(char *s);
```

mydef.h

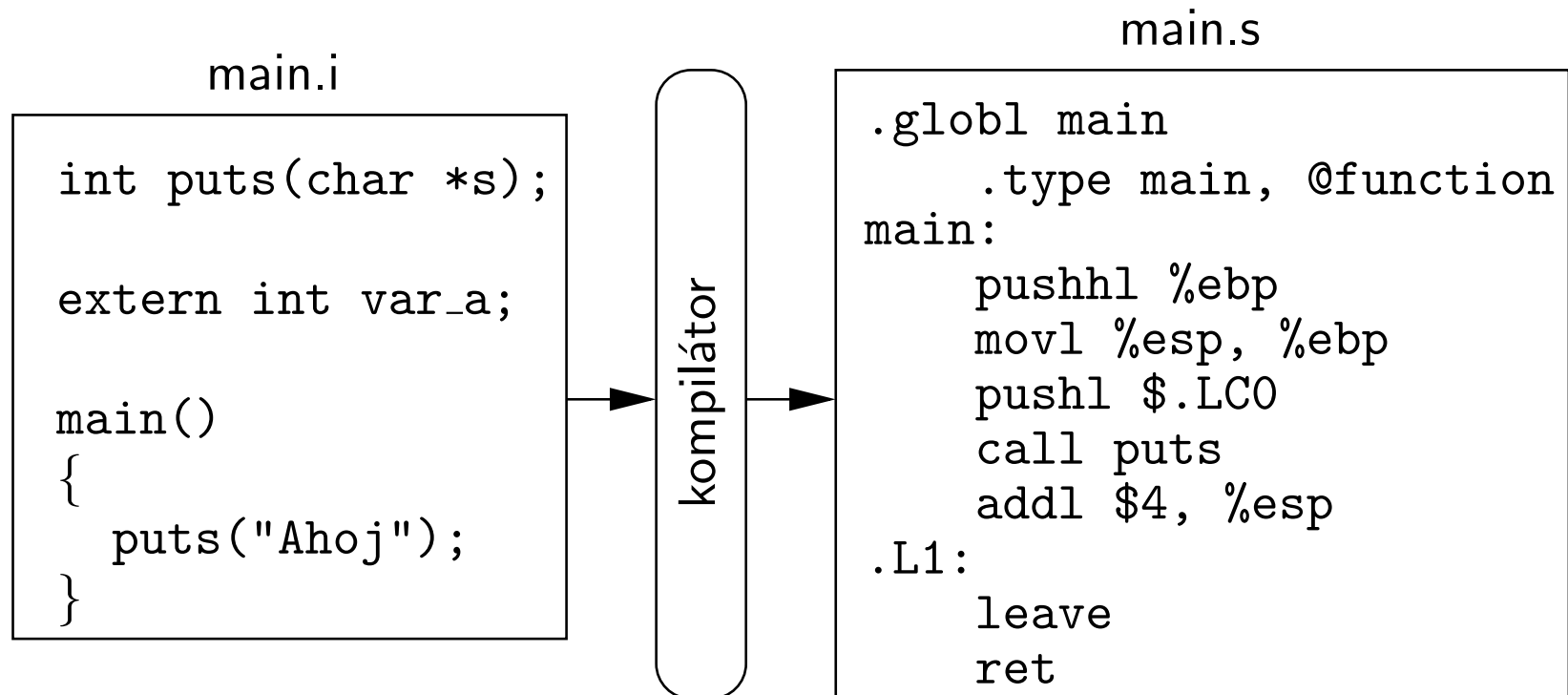
```
#define MSG "Ahoj"
extern int var_a;
```

preprocesor

main.i

```
int puts(char *s);
extern int var_a;
main()
{
    puts("Ahoj");
}
```

# Překlad jednoho modulu (kompilátor)



# Překlad jednoho modulu (assembler)

main.s

```
.globl main
.type main, @function
main:
    pushhl %ebp
    movl %esp, %ebp
    pushl $.LC0
    call puts
    addl $4, %esp
.L1:
    leave
    ret
```

assembler

main.o

```
457f 464c 0101 0001
0000 0000 0000 0000
0001 0003 0001 0000
0000 0000 0000 0000
00ec 0000 0000 0000
0034 0000 0000 0028
```



# Kompilátor

- volání:

`cc [options] soubor ...`

- nejdůležitější přepínače:

`-o soubor` jméno výsledného souboru

`-c` pouze překlad (nelinkovat)

`-E` pouze preprocesor (nepřekládat)

`-l` slinkuj s příslušnou knihovnou

`-Ljméno` přidej adresář pro hledání knihoven z `-l`

`-Olevel` nastavení úrovně optimalizace

`-g` překlad s ladicími informacemi

`-Djméno` definuj makro pro preprocesor

`-Iadresář` umístění `#include` souborů

# Předdefinovaná makra

`__FILE__`, `__LINE__`, `__DATE__`, `__TIME__`, `__cplusplus`, apod.

jsou standardní makra kompilátoru C/C++

`unix` vždy definováno v Unixu

`mips`, `i386`, `sparc` hardwarová architektura

`linux`, `sgi`, `sun`, `bsd` klon operačního systému

`_POSIX_SOURCE`, `_XOPEN_SOURCE`

překlad podle příslušné normy

pro překlad podle určité normy by před prvním `#include` měl být řádek s definicí následujícího makra. Pak načtěte `unistd.h`.

**UNIX 98** `#define _XOPEN_SOURCE 500`

**SUSv3** `#define _XOPEN_SOURCE 600`

**POSIX1990** `#define _POSIX_SOURCE`

# Link editor (linker)

- Volání:

```
ld [options] soubor ...
```

```
cc [options] soubor ...
```

- Nejdůležitější přepínače:

`-o soubor`      jméno výsledného souboru (default `a.out`)

`-llib`            linkuj s knihovnou `liblib.so` nebo `liblib.a`

`-Lpath`           cesta pro knihovny (`-llib`)

`-shared`          vytvořit sdílenou knihovnu

`-non_shared`     vytvořit statický program

# Řízení překladač a linkování (make)

- zdrojové texty

main.c

```
#include "util.h"
main()
{
    msg();
}
```

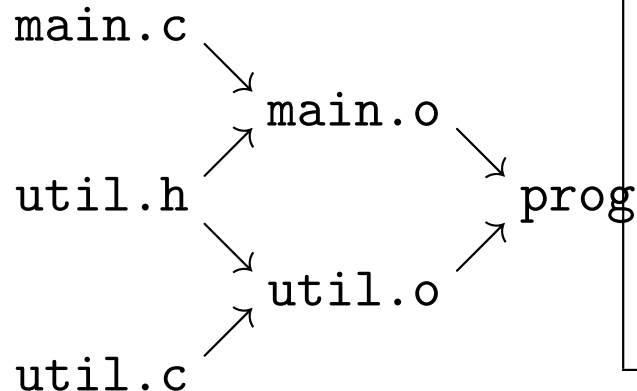
util.h

```
void msg();
```

util.c

```
#include "util.h"
msg()
{
    puts();
}
```

- závislosti



- soubor Makefile

```
prog : main.o util.o
      cc -o prog main.o util.o
main.o : main.c util.h
      cc -c main.c
util.o : util.c util.h
      cc -c util.c
```

# Syntaxe vstupního souboru (make)

- popis závislostí cíle: `targets : [files]`
- prováděné příkazy: `<Tab>command`
- komentář: `#comment`
- pokračovací řádek: `line-begin\  
line-continuation`

# Makra (make)

- definice makra:

```
name = string
```

- pokračování vkládá mezeru
- nedefinovaná makra jsou prázdná
- nezáleží na pořadí definic různých maker
- definice na příkazové řádce:

```
make target name=string
```

- vyvolání makra:

```
$name (pouze jednoznakové name),
```

```
${name} nebo $(name)
```

- systémové proměnné jsou přístupné jako makra

# Dynamický linker (loader)

- překlad vyžaduje všechny potřebné dynamické knihovny pro kontrolu dosažitelnosti použitých symbolů
- sestavení kompletního programu se ale provede až při spuštění. To je úkol pro dynamický linker (*run-time linker, loader*)
- seznam dynamických knihoven zjistí ze sekce `.dynamic`
- systém má nastaveno několik cest, kde se automaticky tyto knihovny hledají
- v sekci `.dynamic` je možné další cesty ke knihovnám přidat pomocí tagů `RUNPATH/RPATH`
- nalezené knihovny se připojí do paměťového procesu pomocí volání `mmap()` (bude později)

# API versus ABI

## API – Application Programming Interface

- rozhraní použité pouze ve zdrojovém kódu
- rozhraní **zdrojáku** vůči systému, knihovně či vlastnímu kódu, tj. např. `exit(1)`, `printf("hello\n")` nebo `my_function(1, 2)`
- ...aby se stejný **zdrojový kód** mohl přeložit na všech systémech podporující dané API

## ABI – Application Binary Interface

- low-level rozhraní **aplikace** vůči systému, knihovně či jiné části sama sebe
- ...aby se **objektový modul** mohl použít všude tam, kde je podporováno stejné ABI



# Debugger dbx

- Volání:

```
dbx [ options ] [ program [ core ] ]
```

- Nejběžnější příkazy:

<code>run [ <i>arglist</i> ]</code>	start programu
<code>where</code>	vypiš zásobník
<code>print <i>expr</i></code>	vypiš výraz
<code>set <i>var</i> = <i>expr</i></code>	změň hodnotu proměnné
<code>cont</code>	pokračování běhu programu
<code>next, step</code>	proved' řádku (bez/s vnořením do funkce)
<code>stop <i>condition</i></code>	nastavení breakpointu
<code>trace <i>condition</i></code>	nastavení tracepointu
<code>command <i>n</i></code>	akce na breakpointu (příkazy následují)
<code>help [ <i>name</i> ]</code>	nápověda
<code>quit</code>	ukončení debuggeru

# GNU debugger gdb

- Volání:

```
gdb [ options ] [ program [ core ] ]
```

- Nejběžnější příkazy:

<code>run [arglist]</code>	start programu
<code>bt</code>	vypiš zásobník
<code>print expr</code>	vypiš výraz
<code>set var = expr</code>	změň hodnotu proměnné
<code>cont</code>	pokračování běhu programu
<code>next, step</code>	proved' řádku (bez/s vnořením do funkce)
<code>break condition</code>	nastavení breakpointu
<code>help [name]</code>	nápověda
<code>quit</code>	ukončení debuggeru

# Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- **základní pojmy a konvence UNIXu a jeho API**
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

# Standardní hlavičkové soubory (ANSI C)

<code>stdlib.h</code>	...	základní makra a funkce
<code>errno.h</code>	...	ošetření chyb
<code>stdio.h</code>	...	vstup a výstup
<code>ctype.h</code>	...	práce se znaky
<code>string.h</code>	...	práce s řetězcí
<code>time.h</code>	...	práce s datem a časem
<code>math.h</code>	...	matematické funkce
<code>setjmp.h</code>	...	dlouhé skoky
<code>assert.h</code>	...	ladicí funkce
<code>stdarg.h</code>	...	práce s proměnným počtem parametrů
<code>limits.h</code>	...	implementačně závislé konstanty
<code>signal.h</code>	...	ošetření signálů

## Standardní hlavičkové soubory (2)

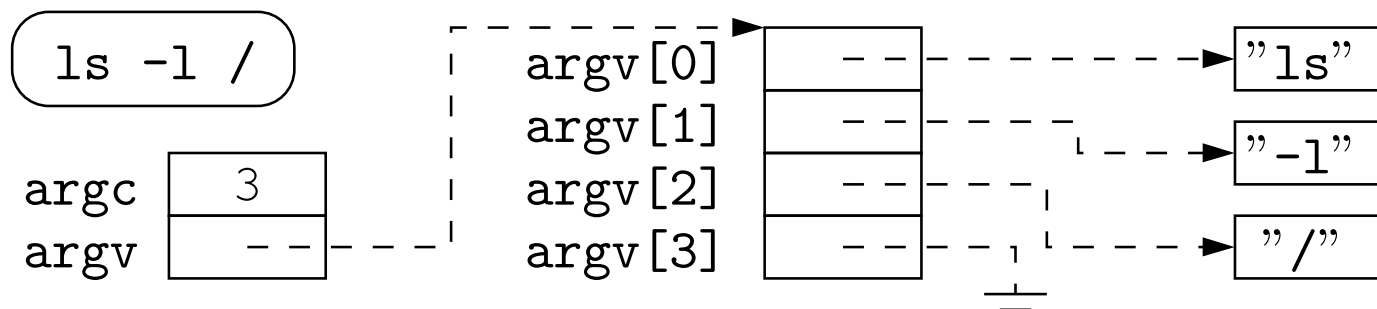
<code>unistd.h</code>	...	symbolické konstanty, typy a základní funkce
<code>sys/types.h</code>	...	datové typy
<code>fcntl.h</code>	...	řídící operace pro soubory
<code>sys/stat.h</code>	...	informace o souborech
<code>dirent.h</code>	...	procházení adresářů
<code>sys/wait.h</code>	...	čekání na synovské procesy
<code>sys/mman.h</code>	...	mapování paměti
<code>curses.h</code>	...	ovládání terminálu
<code>regex.h</code>	...	práce s regulárními výrazy

## Standardní hlavičkové soubory (3)

<code>semaphore.h</code>	...	semafory (POSIX)
<code>pthread.h</code>	...	vlákna (POSIX threads)
<code>sys/socket.h</code>	...	síťová komunikace
<code>arpa/inet.h</code>	...	manipulace se síťovými adresami
<code>sys/ipc.h</code>	...	společné deklarace pro System V IPC
<code>sys/shm.h</code>	...	sdílená paměť (System V)
<code>sys/msg.h</code>	...	fronty zpráv (System V)
<code>sys/sem.h</code>	...	semafory (System V)

# Funkce main()

- při spuštění programu je předáno řízení funkci main().
- `int main (int argc, char *argv []);`
  - `argc` ... počet argumentů příkazové řádky
  - `argv` ... pole argumentů
    - \* podle konvence je `argv[0]` jméno programu (bez cesty)
    - \* poslední prvek je `argv[argc] == NULL`
  - návrat z `main()` nebo volání `exit()` ukončí program
  - standardní návratové hodnoty `EXIT_SUCCESS (0)` a `EXIT_FAILURE (1)`



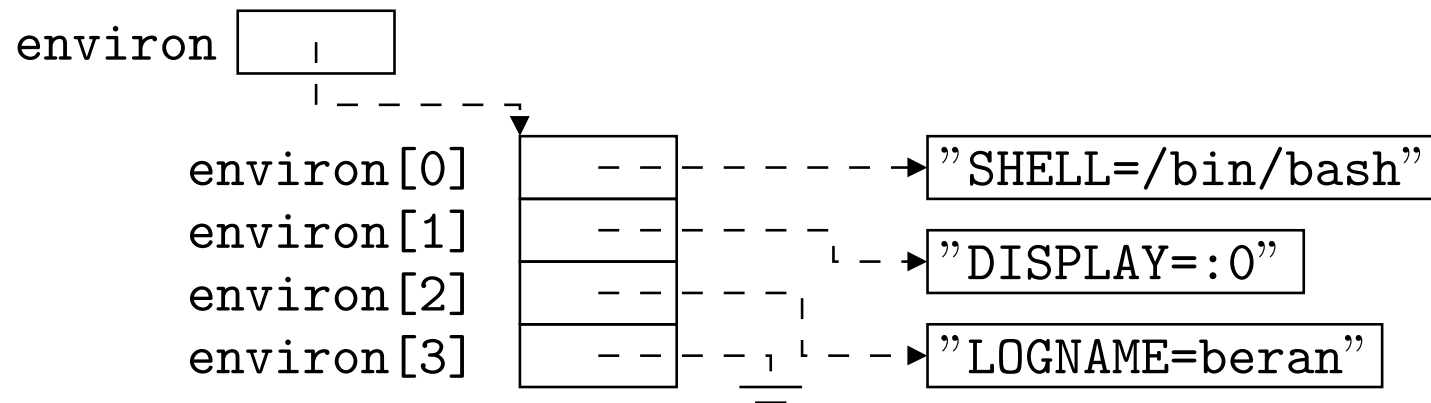
# Proměnné prostředí

- seznam všech proměnných prostředí (*environment variables*) se předává jako proměnná

```
extern char **environ;
```

- je to pole ukazatelů (ukončené NULL) na řetězce ve tvaru:

*proměnná=hodnota*





# Manipulace s proměnnými prostředí

- je možné přímo měnit proměnnou `environ`, SUSv3 to ale nedoporučuje
- `char *getenv (const char *name);`
  - vrátí hodnotu proměnné `name`
- `int putenv (char *string);`
  - vloží `string` ve tvaru *jméno=hodnota* do prostředí (přidá novou nebo modifikuje existující proměnnou)
- změny se přenášejí do synovských procesů
- změny v prostředí syna samozřejmě prostředí otce neovlivní
- existují i funkce `setenv()` a `unsetenv()`

# Zpracování argumentů programu

- obvyklý zápis v shellu: `program -přepínače argumenty`
- přepínače tvaru `-x` nebo `-x hodnota`, kde `x` je jedno písmeno nebo číslice, *hodnota* je libovolný řetězec
- několik přepínačů lze sloučit dohromady: `ls -lRa`
- argument `'--'` nebo první argument nezačínající `'-'` ukončuje přepínače, následující argumenty nejsou považovány za přepínače, i když začínají znakem `'-'`.
- tento tvar argumentů požaduje norma a lze je zpracovávat automaticky funkcí `getopt`.

## Zpracování přepínačů: getopt()

```
int getopt(int argc, char *const argv[],  
           const char *optstring);
```

```
extern char *optarg;
```

```
extern int optind, opterr, optopt;
```

- funkce dostane parametry z příkazového řádku, při každém volání zpracuje a vrátí další přepínač. Pokud má přepínač hodnotu, vrátí ji v `optarg`.
- když jsou vyčerpány všechny přepínače, vrátí `-1` a v `optind` je číslo prvního nezpracovaného argumentu.
- možné přepínače jsou zadány v `optstring`, když za znakem přepínače následuje `:`, má přepínač povinnou hodnotu.
- při chybě (neznámý přepínač, chybí hodnota) vrátí `?`, uloží znak přepínače do `optopt` a když `opterr` nebylo nastaveno na nulu, vypíše chybové hlášení.

## Příklad použití getopt()

```
struct {
    int a, b; char c[128];
} opts;
int opt; char *arg1;

while((opt = getopt(argc, argv, "abc:")) != -1)
    switch(opt) {
        case 'a': opts.a = 1; break;
        case 'b': opts.b = 1; break;
        case 'c': strcpy(opts.c, optarg); break;
        case '?': fprintf(stderr,
            "usage: %s [-ab] [-c Carg] arg1 arg2 ...\n",
            basename(argv[0])); break;
    }
arg1 = argv[optind];
```

# Dlouhý tvar přepínačů

- poprvé se objevilo v GNU knihovně `libiberty`:

`--jméno` nebo `--jméno=hodnota`

- argumenty se permutují tak, aby přepínače byly na začátku, např. `ls * -l` je totéž jako `ls -l *`, standardní chování lze docílit nastavením proměnné `POSIXLY_CORRECT`.
- zpracovávají se funkcí `getopt_long()`, která používá pole struktur popisujících jednotlivé přepínače:

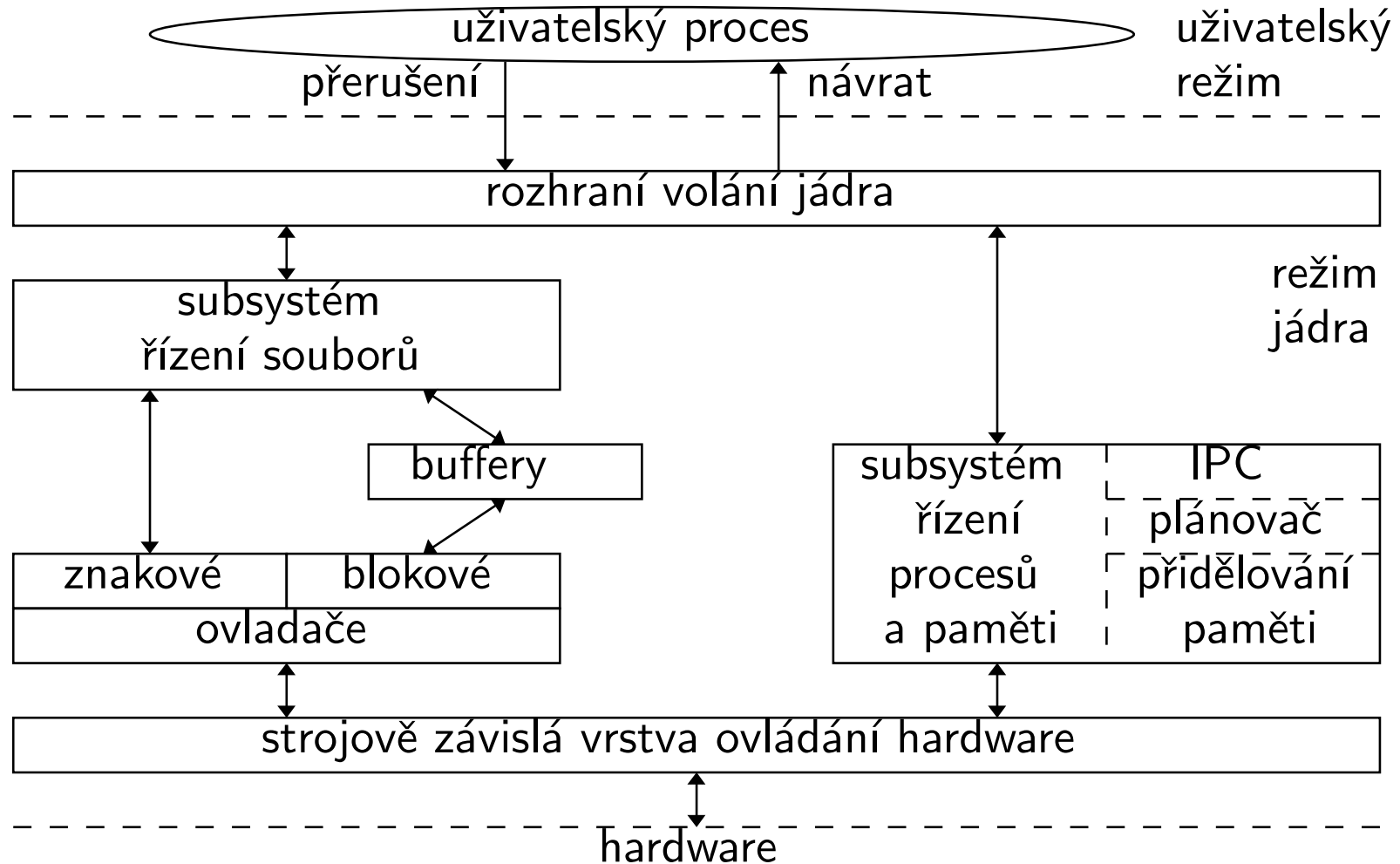
```
struct option {  
    const char *name; /* jméno přepínače */  
    int has_arg; /* hodnota: ano, ne, volitelně */  
    int *flag; /* když je NULL, funkce vrátí val, jinak vrátí 0  
                a dá val do *flag */  
    int val; /* návratová hodnota */  
};
```

## Dlouhé přepínače (pokračování)

```
int getopt_long(int argc, char * const argv [],
               const char *optstring,
               const struct option *longopts,
               int *longindex);
```

- `optstring` obsahuje jednopísmenné přepínače, `longopts` obsahuje adresu pole struktur pro dlouhé přepínače (poslední záznam pole obsahuje samé nuly)
- pokud funkce narazí na dlouhý přepínač, vrátí odpovídající `val` nebo nulu (pokud `flag` nebyl `NULL`), jinak je chování shodné s `getopt`.
- do `*longindex` (když není `NULL`) dá navíc index nalezeného přepínače v `longopts`.

# Struktura klasického OS UNIX



# Procesy, vlákna, programy

- **proces** je systémový objekt charakterizovaný svým kontextem, identifikovaný jednoznačným číslem (**process ID, PID**); jinými slovy „kód a data v paměti”
- **vlákno (thread)** je systémový objekt, který existuje uvnitř procesu a je charakterizován svým stavem. Všechna vlákna jednoho procesu sdílí stejný paměťový prostor kromě registrů procesoru a zásobníku; „linie výpočtu”, „to, co běží”
- **program** ... soubor přesně definovaného formátu obsahující instrukce, data a služební informace nutné ke spuštění; „spustitelný soubor na disku”
  - **paměť** se přiděluje **procesům**.
  - **procesory** se přidělují **vláknům**.
  - vlákna jednoho procesu mohou běžet na různých procesorech.

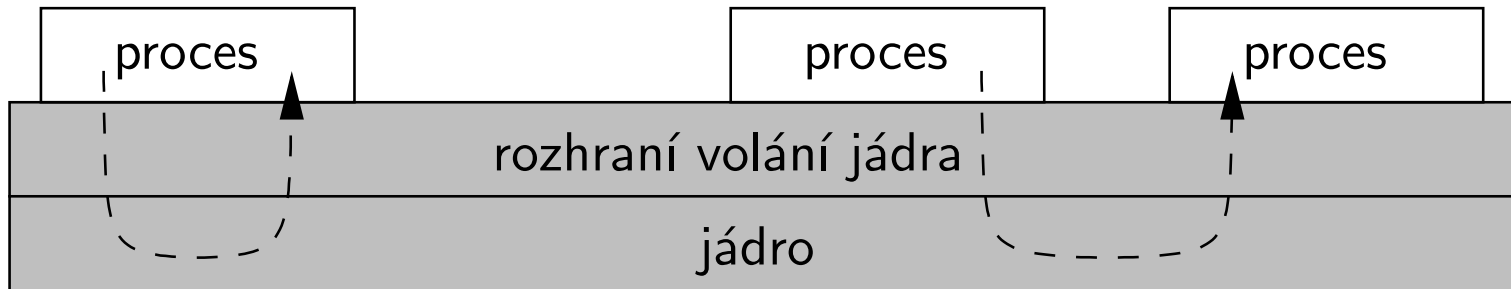


# Jádro, režimy, přerušení (klasický UNIX)

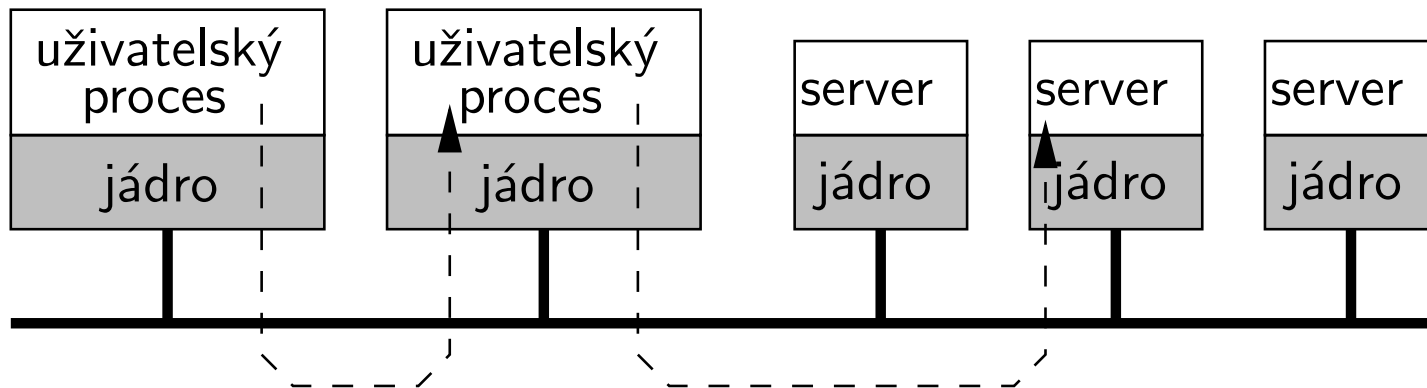
- procesy typicky běží v uživatelském režimu
- systémové volání způsobí přepnutí do režimu jádra
- proces má pro každý režim samostatný zásobník
- jádro je částí každého uživatelského procesu, není to samostatný proces (procesy)
- přepnutí na jiný proces se nazývá *přepnutí kontextu*
- obsluha přerušení se provádí v kontextu přerušeného procesu
- klasické jádro je nepreemptivní

# Volání služeb a komunikace mezi procesy

- UNIX



- distribuovaný OS



# Systémová volání, funkce

- v UNIXu se rozlišují **systémová volání** a **knihovní funkce**. Toto rozlišení dodržují i manuálové stránky: sekce **2** obsahuje systémová volání (*syscalls*), sekce **3** knihovní funkce (*library functions*).
  - knihovní funkce se vykonávají v uživatelském režimu, stejně jako ostatní kód programu.
  - systémová volání mají také tvar volání funkce. Příslušná funkce ale pouze zpracuje argumenty volání a předá řízení jádru pomocí instrukce synchronního přerušení. Po návratu z jádra funkce upraví výsledek a předá ho volajícímu.
- standardy tyto kategorie nerozlišují – z hlediska programátora je jedno, zda určitou funkci provede jádro nebo knihovna.

# Návratové hodnoty systémových volání

- celočíselná návratová hodnota (`int`, `pid_t`, `off_t`, apod.)
  - `>= 0` ... operace úspěšně provedena
  - `== -1` ... chyba
- návratová hodnota typu ukazatel
  - `!= NULL` ... operace úspěšně provedena
  - `== NULL` ... chyba
- po neúspěšném systémovém volání je kód chyby v globální proměnné `extern int errno`;
- úspěšné volání nemění hodnotu v `errno`! Je tedy třeba nejprve otestovat návratovou hodnotu a pak teprve `errno`.
- chybové hlášení podle hodnoty v `errno` vypíše funkce `void perror(const char *s);`
- textový popis chyby s daným číslem vrátí funkce `char *strerror(int errnum);`

## Skupina funkcí z `err(3)`

- pomocné funkce pro výpis chybových hlášení a případné ukončení programu
- místo `perror()` a `exit()` vám stačí jedna funkce
- `void err(int status, const char *fmt, ...);`
  - vypíše jméno programu, formátovaný řetězec, a chybu podle aktuální hodnoty `errno`
  - ukončí program s návratovou hodnotou ze `status`
- `void warn(const char *fmt, ...);`
  - stejné jako `err()`, ale program neukončí
- existují další podobné funkce, viz manuálová stránka
- funkce pocházejí ze 4.4BSD

# Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- **přístupová práva, periferní zařízení, systém souborů**
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

# Uživatelé a skupiny

```
beran:x:1205:106:Martin Beran:/home/beran:/bin/bash
```

**význam jednotlivých polí:** uživatelské jméno, zahašované heslo (dnes už v `/etc/shadow`), číslo uživatele (UID); superuživatel (root) má UID 0, číslo primární skupiny (GID), plné jméno, domovský adresář, login-shell

```
sisal*:106:forst,beran
```

**význam jednotlivých polí:** jméno skupiny, heslo pro přepnutí do skupiny, číslo skupiny (GID), seznam členů skupiny

# Name service switch

- dnešní systémy nejsou omezeny na používání `/etc/passwd` a `/etc/groups`
- systém používá *databáze* (`passwd`, `groups`, `protocols`, ...)
- data databází pocházejí ze *zdrojů* (soubory, DNS, NIS, LDAP, ...)
- soubor `nsswitch.conf` definuje jaké databáze používají jaké zdroje
- knihovní funkce toto samozřejmě musí explicitně podporovat
- je možné některé zdroje kombinovat, například uživatel se nejdříve může hledat v `/etc/passwd` a poté v NISu
- poprvé se objevilo v Solarisu, poté převzato dalšími systémy



# Získávání údajů o uživateli/skupinách

- `struct passwd *getpwnam(const char *name)`  
vrací strukturu reprezentující uživatele.
- `struct passwd *getpwuid(uid_t uid)`  
vrací strukturu reprezentující uživatele podle UID.
- `void setpwent(void)`
- `void endpwent(void)`
- `struct passwd *getpwent(void)`  
slouží k procházení položek v databázi uživatelů. **setpwent** nastaví pozici na začátek, **getpwent** získá další položku, **endpwent** dealokuje prostředky použité pro procházení seznamu.

## Testování přístupových práv

- uživatel je identifikován číslem uživatele (**UID**) a čísla skupin, do kterých patří (**primary GID, supplementary GIDs**).
- tuto identifikaci dědí každý proces daného uživatele.
- soubor  $S$  má vlastníka ( $UID_S$ ) a skupinového vlastníka ( $GID_S$ ).
- algoritmus testování přístupových práv pro proces  $P(UID_P, GID_P, SUPG)$  a soubor  $S(UID_S, GID_S)$ :

Jestliže

pak Proces má vůči Souboru

---

`if(UIDP == 0)`

... všechna práva

`else if(UIDP == UIDS)`

... práva vlastníka

`else if(GIDP == GIDS ||`

`GIDS ∈ SUPG)`

... práva člena skupiny

`else`

... práva ostatních

# Reálné a efektivní UID/GID

- u každého procesu se rozlišuje:
  - **reálné UID** (RUID) – skutečný vlastník procesu
  - **efektivní UID** (EUID) – uživatel, jehož práva proces používá
  - **uschované UID** (saved SUID) – původní efektivní UID
- podobně se rozlišuje reálné, efektivní a uschované GID procesu.
- obvykle platí `RUID==EUID && RGID==EGID`.
- **propůjčování práv** ... spuštění programu s nastaveným SUID (**set user ID**) bitem změní EUID a saved UID procesu na UID vlastníka programu, RUID se nezmění.
- podobně SGID bit ovlivňuje EGID procesu.
- **při kontrole přístupových práv se používají vždy EUID, EGID a supplementary GIDs**

# Identifikace vlastníka procesu

- `uid_t getuid(void)`  
vrací reálné user ID volajícího procesu.
- `uid_t geteuid(void)`  
vrací efektivní user ID volajícího procesu.
- `gid_t getgid(void)`  
vrací reálné group ID volajícího procesu.
- `gid_t getegid(void)`  
vrací efektivní group ID volajícího procesu.
- `int getgroups(int gidsz, gid_t glist [])`  
– do *glist* dá nejvýše *gidsz* supplementary group IDs volajícího procesu a vrátí počet všech GIDs procesu.

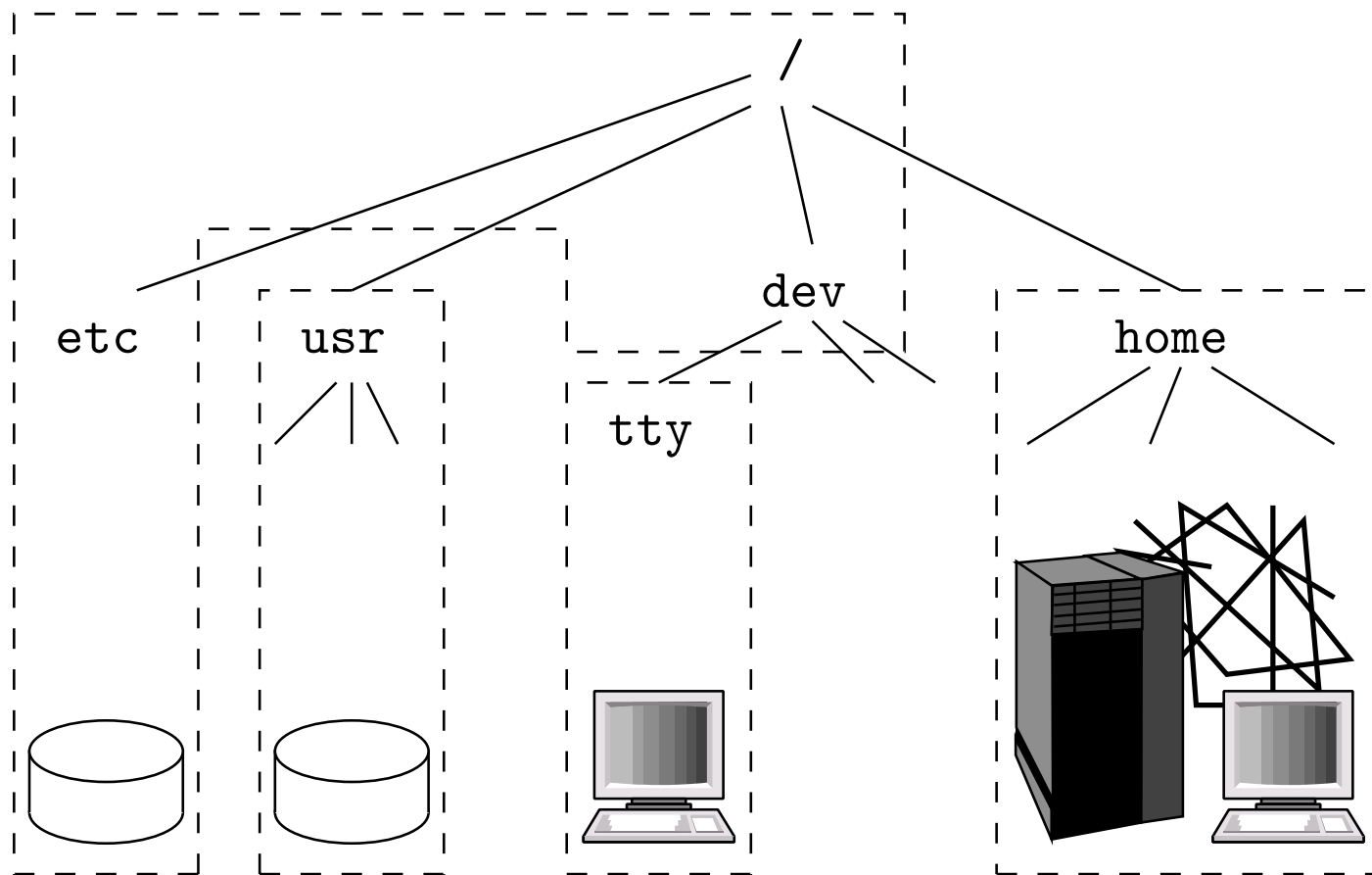
# Změna vlastníka procesu

- `int setuid(uid_t uid);`
  - v procesu s EUID == 0 nastaví RUID, EUID i saved-SUID na `uid`
  - pro ostatní procesy nastavuje jen EUID, a `uid` musí být buď rovné RUID nebo uschovanému SUID
- `int setgid(gid_t gid);`  
obdoba `setuid`, nastavuje group-IDs procesu.
- `int setgroups(int ngroups, gid_t *gidset)`  
nastavuje supplementary GIDs procesu, může být použito jen superuživatelským procesem.

# System souborů

- adresáře tvoří strom, spolu se soubory acyklický graf (na jeden soubor může existovat více odkazů).
- každý adresář navíc obsahuje odkaz na sebe '.' (tečka) a na nadřazený adresář '..' (dvě tečky).
- pomocí rozhraní systému souborů se přistupuje i k dalším entitám v systému:
  - periferní zařízení
  - pojmenované roury
  - sokety
  - procesy (/proc)
  - paměť (/dev/mem, /dev/kmem)
  - pseudosoubory (/dev/tty, /dev/fd/0,...)
- z pohledu jádra je každý obyčejný soubor pole bajtů.
- všechny (i síťové) disky jsou zapojeny do jednoho stromu.

# Jednotný hierarchický systém souborů



# Typická skladba adresářů

<code>/bin</code>	...	základní systémové příkazy
<code>/dev</code>	...	speciální soubory (zařízení, devices)
<code>/etc</code>	...	konfigurační adresář
<code>/lib</code>	...	základní systémové knihovny
<code>/tmp</code>	...	veřejný adresář pro dočasné soubory
<code>/home</code>	...	kořen domovských adresářů
<code>/var/adm</code>	...	administrativní soubory (ne na BSD)
<code>/usr/include</code>	...	hlavičkové soubory pro C
<code>/usr/local</code>	...	lokálně instalovaný software
<code>/usr/man</code>	...	manuálové stránky
<code>/var/spool</code>	...	spool (pošta, tisk,...)

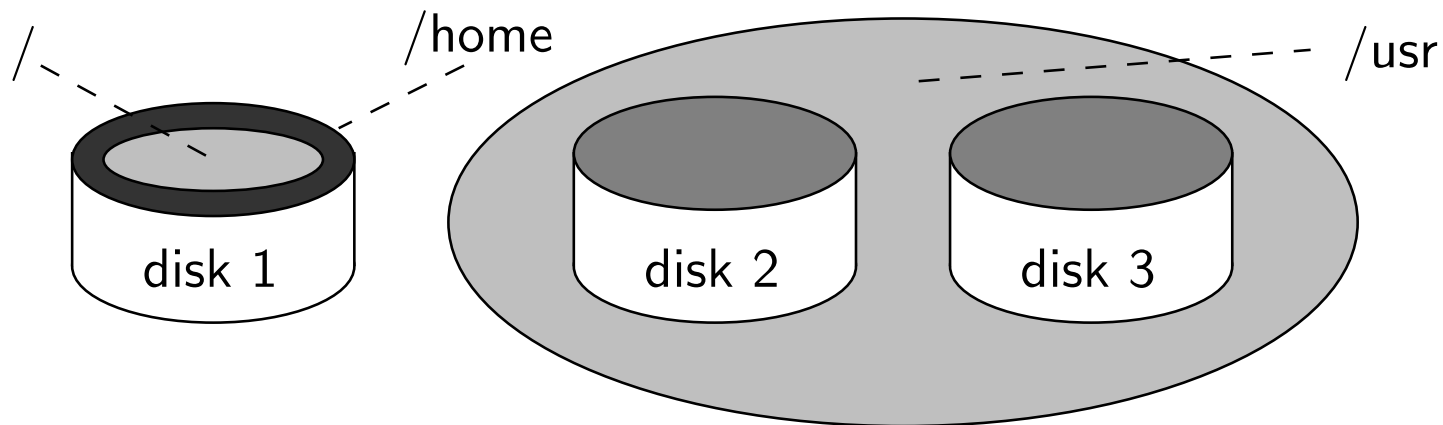


# Přístup k periferním zařízením

- adresář `/dev` obsahuje speciální soubory zařízení. Proces otevře speciální soubor systémovým voláním `open()` a dále komunikuje se zařízením pomocí volání `read()`, `write()`, `ioctl()`, apod.
- speciální soubory se dělí na
  - **znakové** ... data se přenáší přímo mezi procesem a ovladačem zařízení, např. sériové porty
  - **blokové** ... data prochází systémovou vyrovnávací pamětí (buffer cache) po blocích pevně dané velikosti, např. disky
- speciální soubor identifikuje zařízení dvěma čísly
  - **hlavní (major) číslo** ... číslo ovladače v jádru
  - **vedlejší (minor) číslo** ... číslo v rámci jednoho ovladače

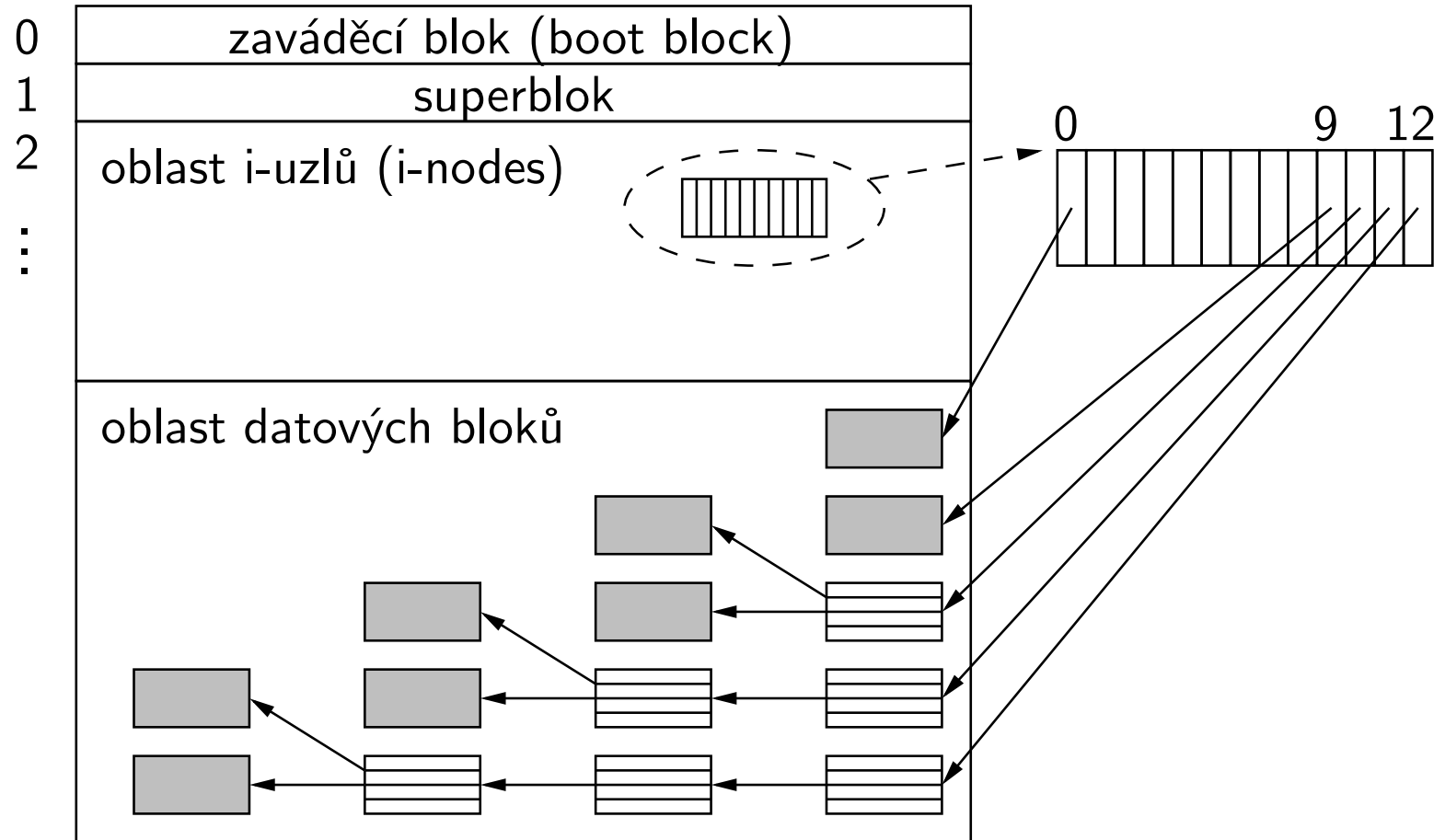
# Fyzické uložení systému souborů

- **systém souborů** (svazek, **filesystem**) lze vytvořit na:
  - **oddílu disku** (**partition**) – část disku, na jednom disku může být více oddílů
  - **logickém oddílu** (**logical volume**) – takto lze spojit více oddílů, které mohou být i na několika discích, do jednoho svazku.
- další možnosti: striping, mirroring, RAID



# Organizace systému souborů s5

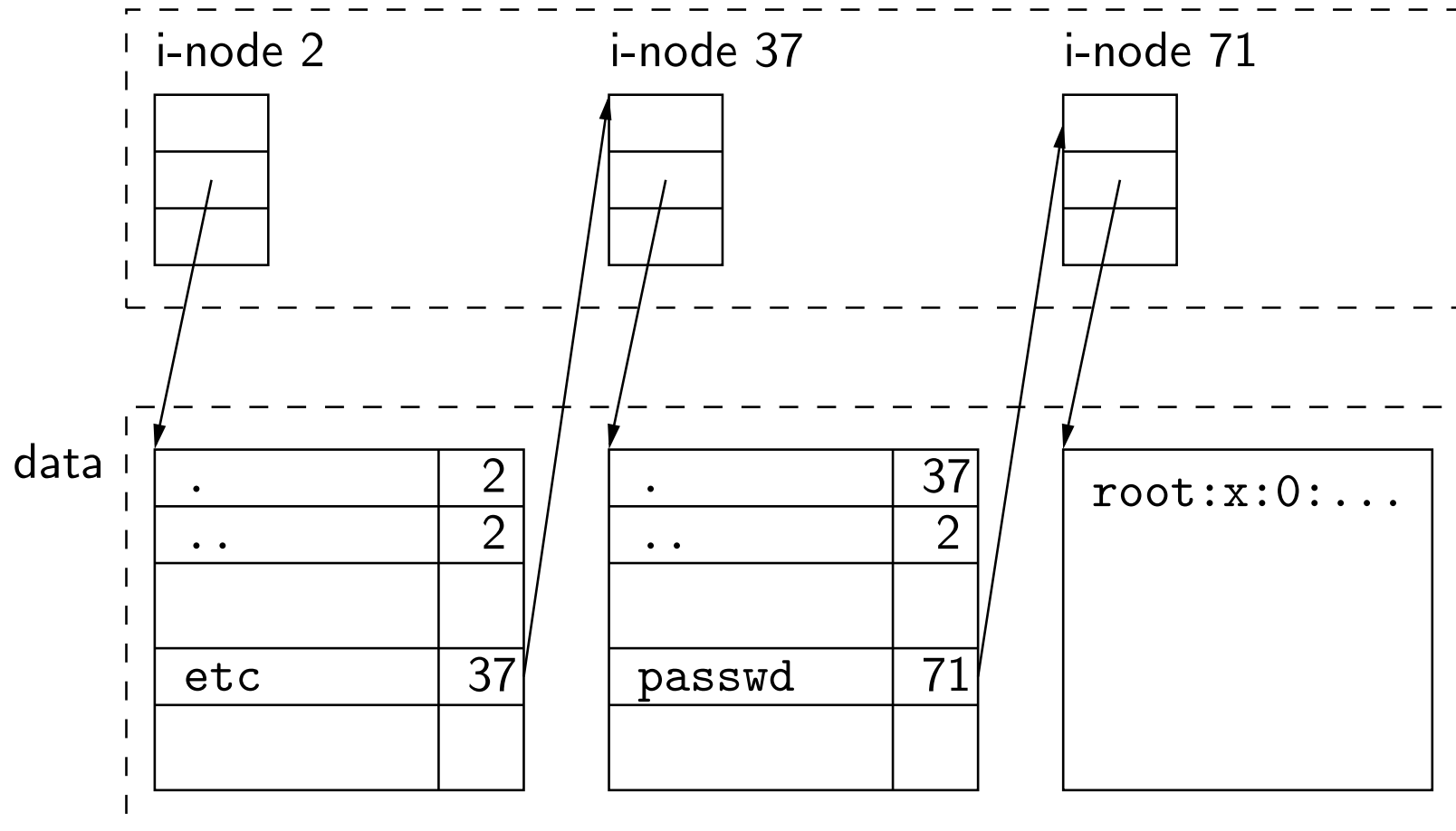
blok č.



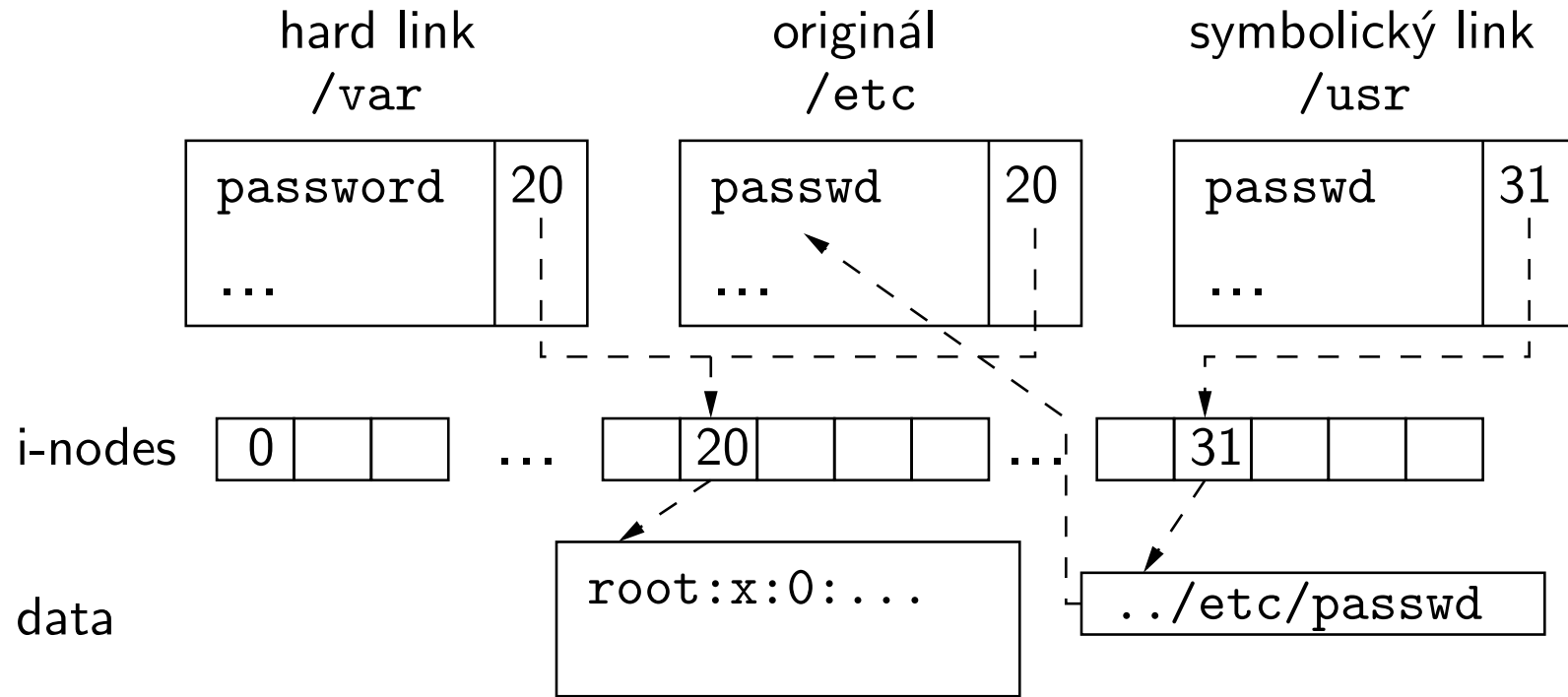
# Navigace v adresářové struktuře

i-nodes

/etc/passwd



# Linky



Hard linky lze vytvářet pouze v rámci jednoho (logického) filesystemu.

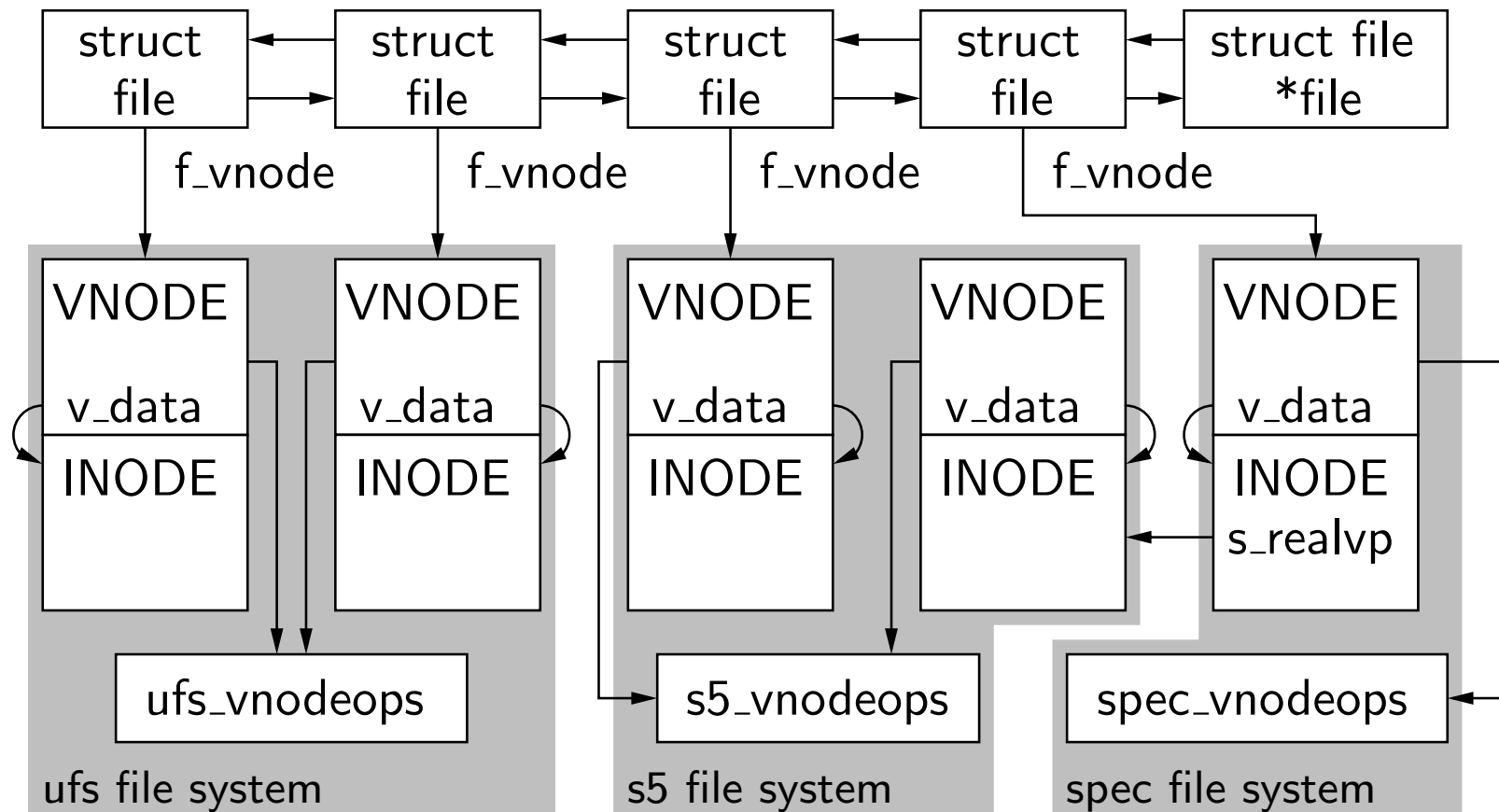
# Vylepšení systému souborů

- cíl: snížení fragmentace souborů, omezení pohybu hlav disku umístěním i-uzlů a datových bloků blíž k sobě
- UFS (Unix File System), původně Berkeley FFS (Fast File System)
- členění na skupiny cylindrů, každá skupina obsahuje
  - kopii superbloku
  - řídicí blok skupiny
  - tabulku i-uzlů
  - bitmapy volných i-uzlů a datových bloků
  - datové bloky
- bloky velikosti 4 až 8 kB, menší části do fragmentů bloků
- jména dlouhá 255 znaků

# Vývoj ve správě adresářových položek

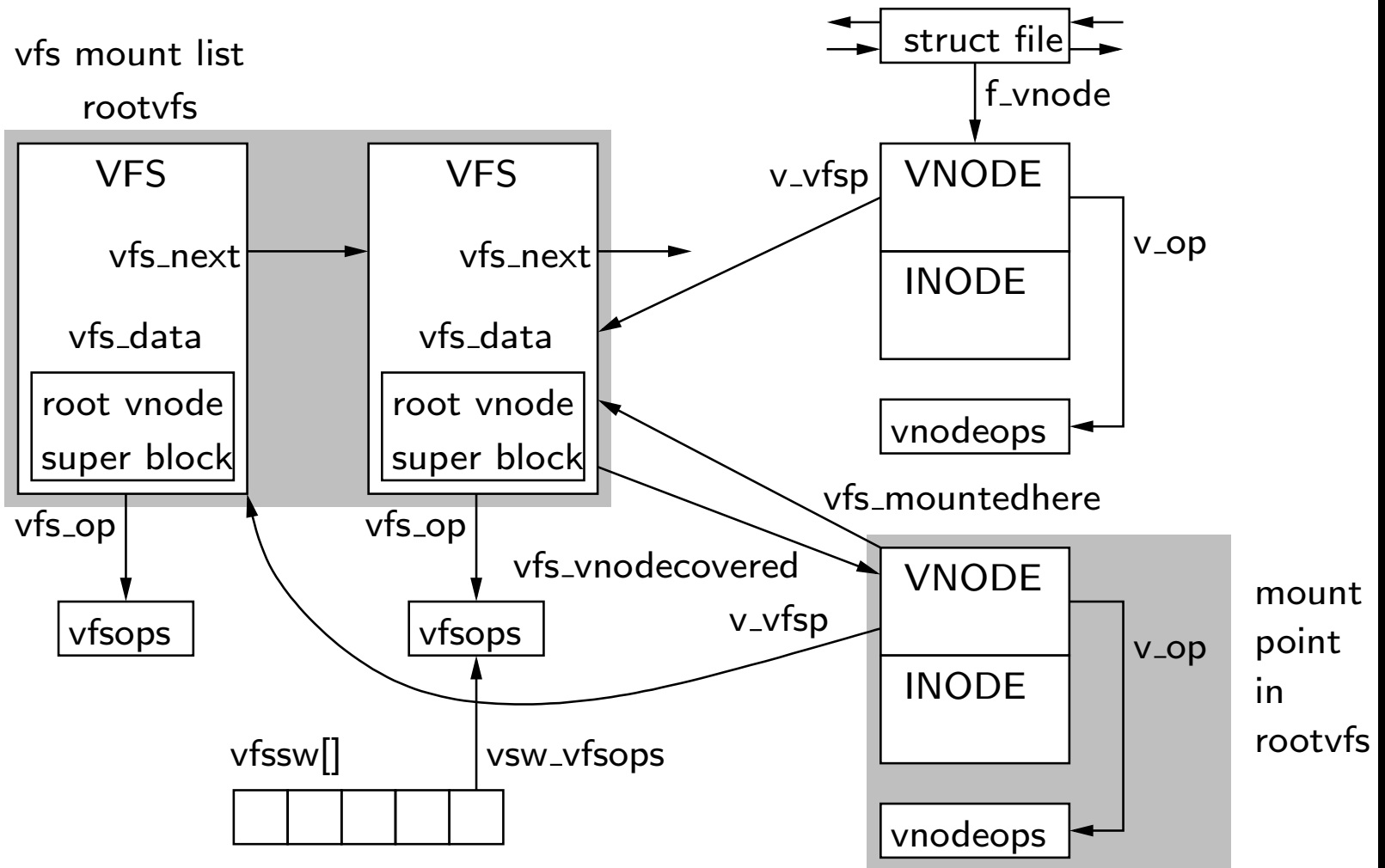
- maximální délka jména souboru 14 znaků nebyla dostačující
- FFS – délka až 255; každá položka zároveň obsahuje i její délku
- nové filesystemy používají pro vnitřní strukturu adresářů různé varianty B-stromů
  - výrazně zrychluje práci s adresáři obsahující velké množství souborů
  - XFS, JFS, ReiserFS, ...
- UFS2 zavádí zpětně kompatibilní tzv. *dirhash* pro zrychlení přístupu k adresářům s velkým počtem souborů

# Virtuální systém souborů (Virtual File System)

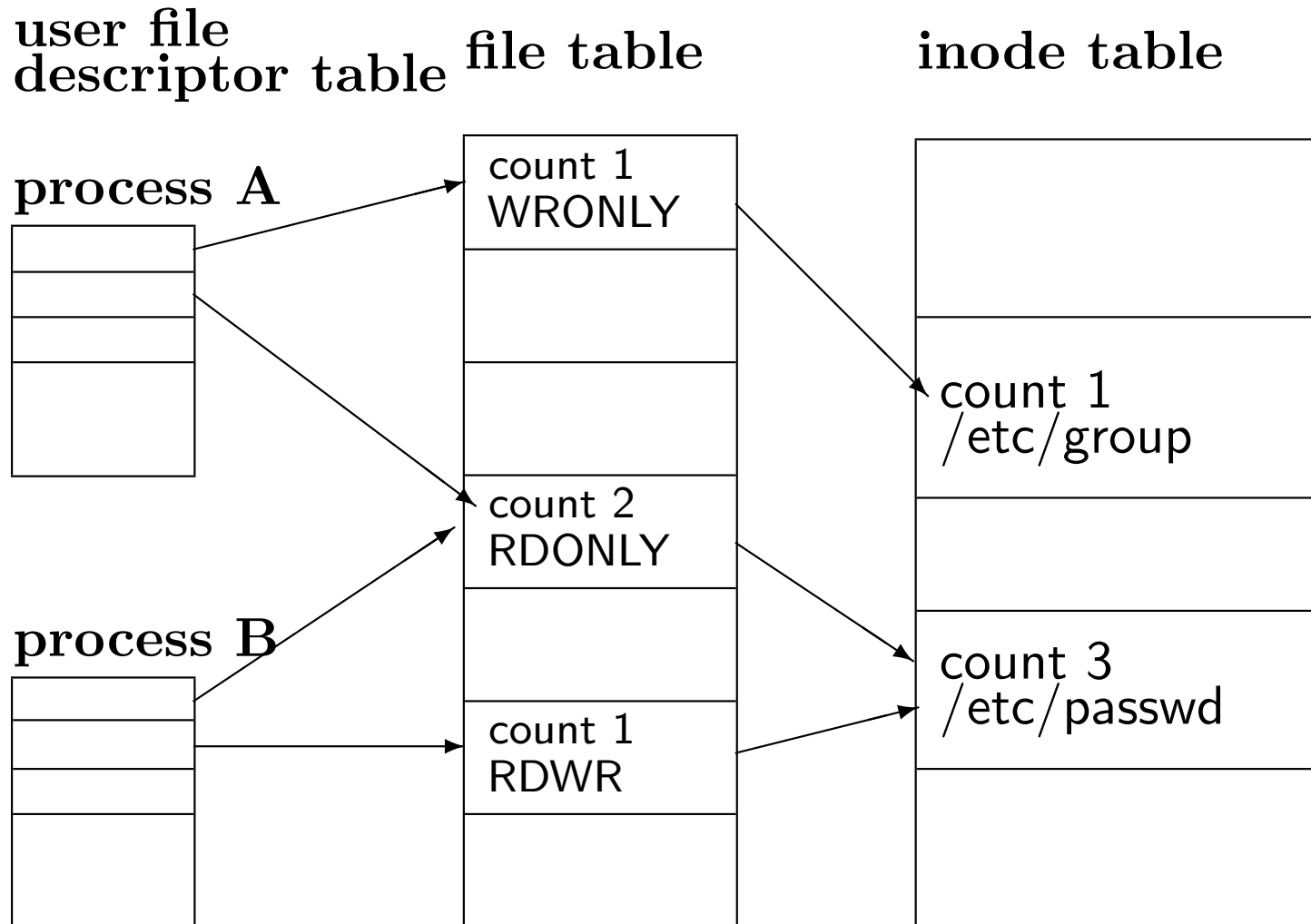




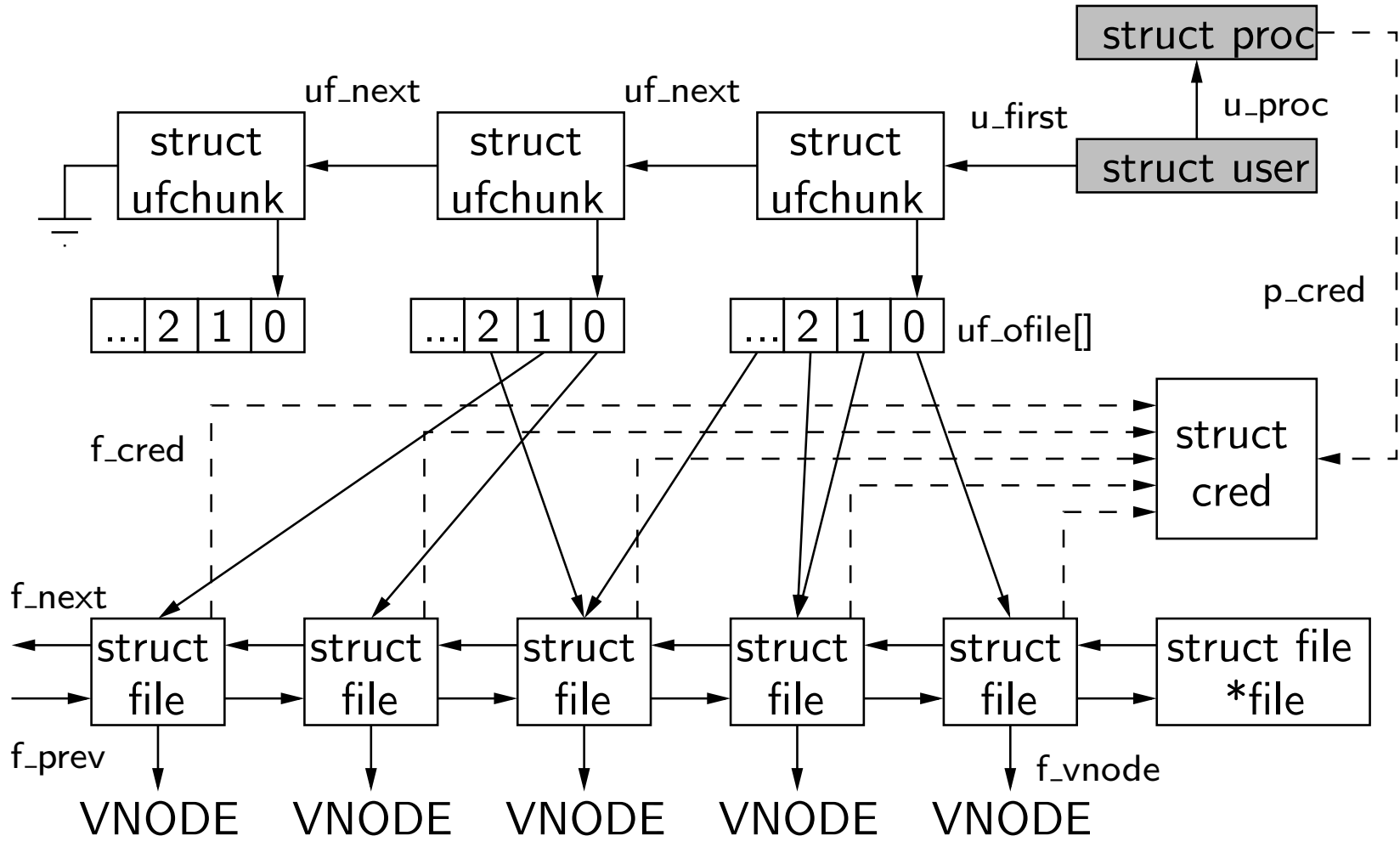
# Hierarchie souborových systémů



# Otevřené soubory z pohledu jádra I.



# Otevřené soubory z pohledu jádra II.



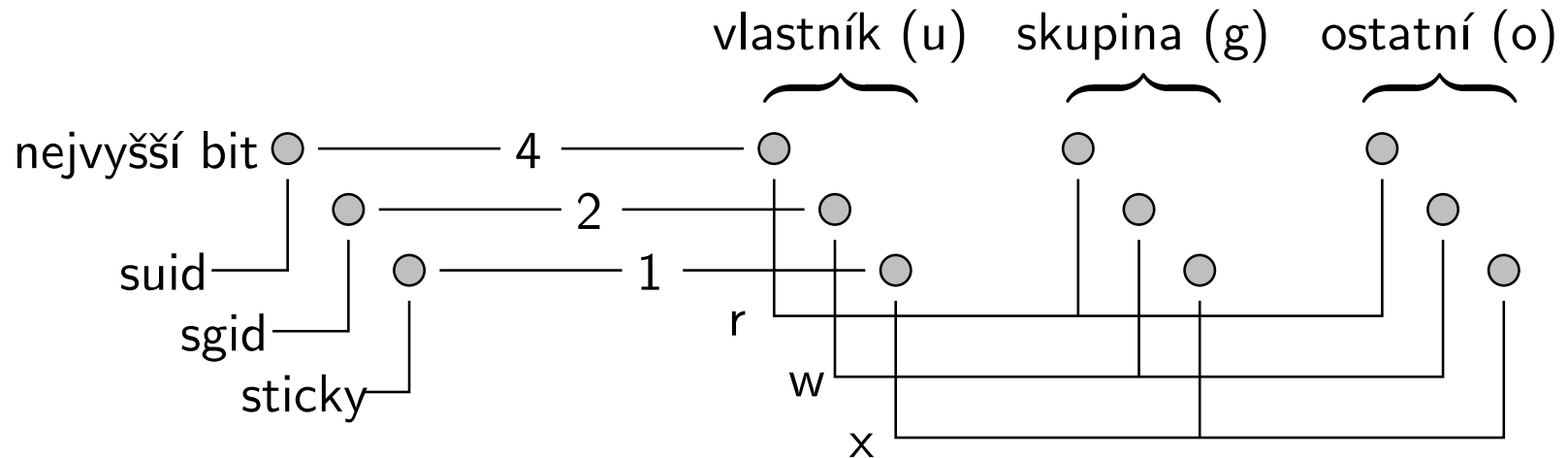
# Oprava konzistence souborového systému

- pokud není filesystem před zastavením systému korektně odpojen, mohou být data v nekonzistentním stavu.
- ke kontrole a opravě svazku slouží příkaz `fsck`. Postupně testuje možné nekonzistence:
  - vícenásobné odkazy na stejný blok
  - odkazy na bloky mimo rozsah datové oblasti systému souborů
  - špatný počet odkazů na i-uzly
  - nesprávná velikost souborů a adresářů
  - neplatný formát i-uzlů
  - bloky které nejsou obsazené ani volné
  - chybný obsah adresářů
  - neplatný obsah superbloku
- operace `fsck` je časově náročná.
- žurnálové (např. XFS v IRIXu, Ext3 v Linuxu) a transakční (ZFS) systémy souborů nepotřebují `fsck`.

# Další způsoby zajištění konzistence filesystemu

- tradiční UFS – synchronní zápis metadat
  - aplikace vytvářející nový soubor čeká na inicializaci inode na disku; tyto operace pracují rychlostí disku a ne rychlostí CPU
  - asynchronní zápis ale častěji způsobí nekonzistenci metadat
- řešení problémů s nekonzistencí metadat na disku:
  - *journaling* – skupina na sobě závislých operací se nejdříve atomicky uloží do žurnálu; při problémech se pak žurnál může “přehrát”
  - bloky metadat se nejdříve zapíše do non-volatile paměti
  - *soft-updates* – sleduje závislosti mezi ukazateli na diskové struktuře a zapisuje data na disk metodou *write-back* tak, že data na disku jsou vždy konzistentní
  - *ZFS* je nový filesystem v Solarisu, který používá *copy-on-write* transakční model

# Přístupová práva



- **SGID** pro soubor bez práva spuštění pro skupinu v System V: kontrola zámeků při každém přístupu (**mandatory locking**)
- **sticky bit** pro adresáře: právo mazat a přejmenovávat soubory mají jen vlastníci souborů
- **SGID** pro adresář: nové soubory budou mít stejnou skupinu jako adresář (System V; u BSD systémů to funguje jinak, viz poznámky)

# API pro soubory

- před použitím musí proces každý soubor nejprve otevřít voláním `open()` nebo `creat()`.
- otevřené soubory jsou dostupné přes **deskriptory souborů** (file descriptors), číslované od 0, více deskriptorů může sdílet jedno **otevření souboru** (mód čtení/zápis, ukazatel pozice)
- standardní deskriptory:
  - 0 ... standardní vstup (jen pro čtení)
  - 1 ... standardní výstup (jen pro zápis)
  - 2 ... chybový výstup (jen pro zápis)
- čtení a zápis z/do souboru: `read()`, `write()`
- změna pozice: `lseek()`, zavření: `close()`, informace: `stat()`, řídicí funkce: `fcntl()`, práva: `chmod()`, ...

# Otevření souboru: `open()`

```
int open(const char *path, int oflag, ... );
```

- otevře soubor daný jménem (cestou) `path`, vrátí číslo jeho deskriptoru (použije první volné), `oflag` je OR-kombinace příznaků
  - `O_RDONLY/O_WRONLY/O_RDWR` ... otevřít pouze pro čtení / pouze pro zápis / pro čtení i zápis
  - `O_APPEND` ... připojování na konec
  - `O_CREAT` ... vytvořit, když neexistuje
  - `O_EXCL` ... chyba, když existuje (použití s `O_CREATE`)
  - `O_TRUNC` ... zrušit předchozí obsah (právo zápisu nutné)
  - ...
- při `O_CREAT` definuje třetí parametr `mode` přístupová práva



# Vytvoření souboru

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

- `open()` s příznakem `O_CREAT` vytvoří soubor, pokud ještě neexistuje. V zadané hodnotě přístupových práv se vynulují bity, které byly nastaveny pomocí funkce

```
mode_t umask(mode_t cmask);
```

- funkce je ekvivalentní volání

```
open(path, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

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

- vytvoří speciální soubor zařízení.

```
int mkfifo(const char *path, mode_t mode);
```

- vytvoří pojmenovanou rouru.

## Čtení a zápis souborů: `read()`, `write()`

```
ssize_t read(int filides, void *buf, size_t nbyte);
```

- z otevřeného souboru s číslem deskriptoru `filides` přečte od aktuální pozice max. `nbyte` bajtů dat a uloží je od adresy `buf`.
- vrací počet skutečně přečtených bajtů (`<= nbyte`), 0 znamená konec souboru.

```
ssize_t write(int filides, const void *buf, size_t nbyte);
```

- do otevřeného souboru s číslem deskriptoru `filides` zapíše na aktuální pozici max. `nbyte` bajtů dat uložených od adresy `buf`.
- vrací velikost skutečně zapsaných dat (`<= nbyte`).

## Uzavření souboru: `close()`

```
int close(int filides);
```

- uvolní deskriptor `filides`, pokud to byl poslední deskriptor, který odkazoval na otevření souboru, zavře soubor a uvolní záznam o otevření souboru.
- když je počet odkazů na soubor 0, jádro uvolní data souboru. Tedy i po zrušení všech odkazů (jmen) mohou se souborem pracovat procesy, které ho mají otevřený. Soubor se smaže, až když ho zavře poslední proces.
- když se zavře poslední deskriptor roury, všechna zbývající data v rouře se zruší.
- při skončení procesu se automaticky provede `close()` na všechny deskriptory.

# Příklad: kopírování souborů

```
#include <sys/types.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
    char buf[4096];
    int inf, outf;
    ssize_t ilen;

    inf = open(argv[1], O_RDONLY);
    outf = creat(argv[2], 0666);
    while ((ilen = read(inf, buf, 4096)) > 0)
        write(outf, buf, ilen);

    close(inf); close(outf);
    return (0);
}
```

# Práce s pojmenovanou rourou

- nemusí být možné vytvořit FIFO na síťovém filesystému (NFS, AFS)
- je nutné znát sémantiku otevírání pojmenované roury:
  - otevření roury pouze pro čtení se zablokuje do té doby, dokud se neobjeví zapisovatel (pokud již neexistuje)
  - otevření roury pouze pro zápis se zablokuje do té doby, dokud se neobjeví čtenář (pokud již neexistuje)
  - toto chování je možné ovlivnit flagem `O_NONBLOCK`
- sémantika čtení a zápisu je ještě o něco složitější, věnujte velkou pozornost poznámkám pod tímto slajdem
  - a je to stejné jako u nepojmenované roury

## Nastavení pozice: `lseek()`

```
off_t lseek(int filides, off_t offset, int whence);
```

- nastaví pozici pro čtení a zápis v otevřeném souboru daném číslem deskriptoru `filides` na hodnotu `offset`.
- podle hodnoty `whence` se `offset` počítá:
  - `SEEK_SET` ... od začátku souboru
  - `SEEK_CUR` ... od aktuální pozice
  - `SEEK_END` ... od konce souboru
- vrací výslednou pozici počítanou od začátku souboru.
- `lseek(filides, 0, SEEK_CUR)` pouze vrátí aktuální pozici.

## Změna velikosti: truncate()

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

```
int ftruncate(int fildes, off_t length);
```

- změní délku souboru zadaného cestou nebo číslem deskriptoru na požadovanou hodnotu.
- při zkrácení souboru zruší nadbytečná data.
- standard ponechává nespecifikované, jestli funguje prodloužení souboru (s vyplněním přidaného úseku nulami). Proto je lepší k prodloužení souboru použít:

```
char buf = '\0';
```

```
lseek(fildes, length - 1, SEEK_SET);
```

```
write(fildes, &buf, 1);
```

## Duplikace deskriptoru: dup(), dup2()

```
int dup(int fil-des);
```

- duplikuje deskriptor *fil-des* na první volný deskriptor, vrátí nový deskriptor, který odkazuje na stejné otevření souboru.
- ekvivalent `fcntl(fil-des, F_DUPFD, 0);`

```
int dup2(int fil-des, int fil-des2);
```

- duplikuje deskriptor *fil-des* do deskriptoru *fil-des2*.
- ekvivalent  
`close(fil-des2);`  
`fcntl(fil-des, F_DUPFD, fil-des2);`



## Příklad: implementace shellového přesměrování

- `$ program < in > out 2>> err`

```
close(0);
open("in", O_RDONLY);
close(1);
open("out", O_WRONLY | O_CREAT | O_TRUNC, 0666);
close(2);
open("err", O_WRONLY | O_CREAT | O_APPEND, 0666);
```

- `$ program > out 2>&1`

```
close(1);
open("out", O_WRONLY | O_CREAT | O_TRUNC, 0666);
close(2);
dup(1);
```

# Řídicí funkce souborů a zařízení: `fcntl()`, `ioctl()`

```
int fcntl(int fildev, int cmd, ...);
```

- slouží pro duplikaci deskriptorů, nastavování zámků, testování a nastavování různých příznaků souboru.

příklad: zavření standardního vstupu při spuštění programu  
(volání typu `exec`)

```
fcntl(0, F_SETFD, FD_CLOEXEC);
```

```
int ioctl(int fildev, int request, ... );
```

- rozhraní pro řídicí funkce periferních zařízení
- používá se jako univerzální rozhraní pro ovládání zařízení, každé zařízení definuje množinu příkazů, kterým rozumí.

## Informace o souboru: stat()

```
int stat(const char *path, struct stat *buf);  
int fstat(int fildes, struct stat *buf);
```

- pro soubor zadaný cestou, resp. číslem deskriptoru, vrátí strukturu obsahující informace o souboru, např.:
  - `st_ino` ... číslo i-uzlu
  - `st_dev` ... číslo zařízení obsahujícího soubor
  - `st_uid`, `st_gid` ... vlastník a skupina souboru
  - `st_mode` ... typ souboru a jeho přístupová práva
  - `st_size`, `st_blksize`, `st_blocks` ... velikost souboru v bajtech, preferovaná velikost bloku pro I/O a počet bloků
  - `st_atime`, `st_mtime`, `st_ctime` ... časy posledního přístupu, modifikace souboru a modifikace i-uzlu
  - `st_nlink` ... počet odkazů na soubor

## Informace o souboru (2)

- pro typ souboru jsou v `<sys/stat.h>` definovány konstanty `S_IFMT` (maska pro typ), `S_IFBLK` (blokový speciální), `S_IFCHR` (znakový speciální), `S_IFIFO` (FIFO), `S_IFREG` (obyčejný), `S_IFDIR` (adresář), `S_IFLNK` (symlink).
- typ lze testovat pomocí maker `S_ISBLK(m)`, `S_ISCHR(m)`, `S_ISFIFO(m)`, `S_ISREG(m)`, `S_ISDIR(m)`, `S_ISLNK(m)`.
- konstanty pro přístupová práva: `S_IRUSR` (čtení pro vlastníka), `S_IWGRP` (zápis pro skupinu), atd.

```
int lstat(const char *path, struct stat *buf);
```

- když je zkoumaný soubor symlink, `stat()` vrátí informace o souboru, na který ukazuje. Tato funkce vrací informace o symlinku.

# Nastavení časů souboru

```
int utime(const char *path, const struct utimbuf  
*times);
```

- nastaví čas poslední modifikace souboru a čas posledního přístupu k souboru.
- nelze změnit čas poslední modifikace i-uzlu.
- volající proces musí mít právo zápisu pro soubor.

# Test přístupových práv: `access()`

```
int access(const char *path, int amode);
```

- otestuje, zda volající proces má k souboru `path` práva daná OR-kombinací konstant v `amode`:
  - `R_OK` ... test práva na čtení
  - `W_OK` ... test práva na zápis
  - `X_OK` ... test práva na spuštění
  - `F_OK` ... test existence souboru
- na rozdíl od `stat()`, výsledek závisí na RUID a RGID procesu
- toto volání se nedá použít bezpečně, proto ho nikdy nepoužívejte procesu

# Nastavení přístupových práv

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

- změní přístupová práva souboru `path` na hodnotu `mode`.
- tuto službu může volat pouze vlastník souboru nebo superuživatel (root).

```
int chown(const char *path, uid_t owner, gid_t group);
```

- změní vlastníka a skupinu souboru `path`. Hodnota `-1` znamená zachovat vlastníka, resp. skupinu.
- měnit vlastníka může jen superuživatel, aby uživatelé nemohli obcházet nastavené quoty tím, že své soubory předají někomu jinému.
- běžný uživatel může měnit skupinu svých souborů a musí přitom patřit do cílové skupiny.

# Manipulace se jmény souborů

```
int link(const char *path1, const char *path2);
```

- vytvoří nový odkaz (položku adresáře) `path2` na soubor `path1`.  
Funguje pouze v rámci jednoho svazku.

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

- zruší odkaz na soubor. Po zrušení posledního odkazu na soubor a uzavření souboru všemi procesy je soubor smazán.

```
int rename(const char *old, const char *new);
```

- změní jméno souboru (přesně odkazu na soubor) z `old` na `new`.  
Funguje pouze v rámci jednoho svazku.



# Symbolické linky

```
int symlink(const char *path1, const char *path2);
```

- vytvoří symbolický link `path2` → `path1`.
- cíl symbolického linku může být i na jiném svazku, popřípadě nemusí vůbec existovat.

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

- do `buf` dá max. `bufsize` znaků z cesty, na kterou ukazuje `symlink path`.
- vrátí počet znaků uložených do `buf`.
- obsah `buf` není zakončen nulou (znakem `'\0'`)!

# Manipulace s adresáři

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

- vytvoří nový prázdný adresář, který bude obsahovat pouze položky '.' a '..'

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

- smaže adresář path. Adresář musí být prázdný.

```
DIR *opendir(const char *dirname);
```

```
struct dirent *readdir(DIR *dirp);
```

```
int closedir(DIR *dirp);
```

- slouží k sekvenčnímu procházení adresářů.
- struktura dirent obsahuje položky
  - d\_ino ... číslo i-uzlu
  - d\_name ... jméno souboru

## Příklad: procházení adresáře

```
int main(int argc, char *argv[])
{
    int i;
    DIR *d;
    struct dirent *de;
    for(i = 1; i < argc; i++) {
        d = opendir(argv[i]);
        while(de = readdir(d))
            printf("%s\n", de->d_name);
        closedir(d);
    }
    return (0);
}
```

# Aktuální adresář procesu

- každý proces má svůj aktuální (pracovní) adresář, vůči kterému jsou udávány relativní cesty k souborům. Počáteční nastavení pracovního adresáře se dědí od otce při vzniku procesu.

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

```
int fchdir(int fildes);
```

- nastaví nový pracovní adresář procesu.

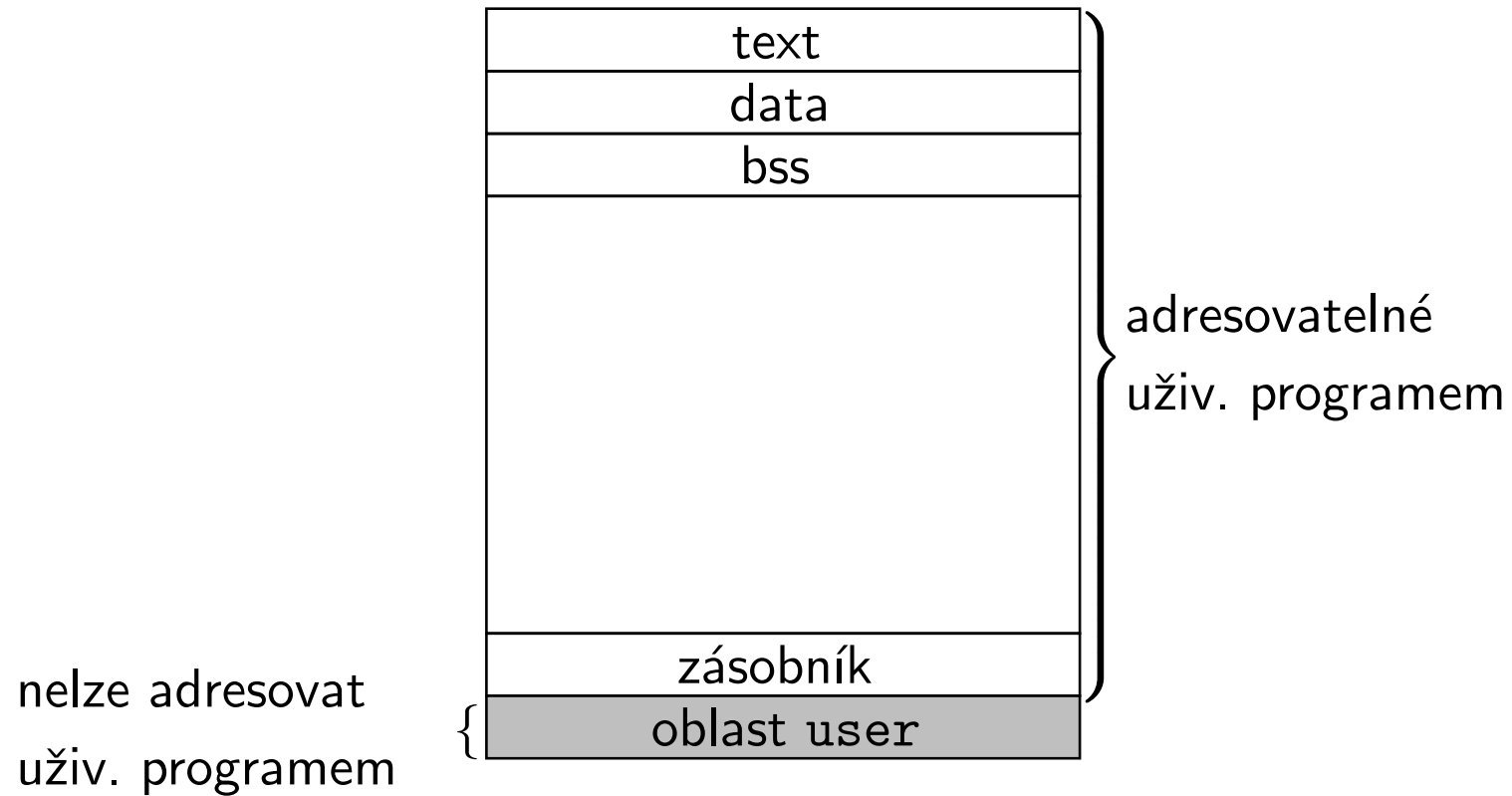
```
char *getcwd(char *buf, size_t size);
```

- uloží absolutní cestu k aktuálnímu adresáři do pole `buf`, jeho délka (`size`) musí být aspoň o 1 větší než délka cesty.

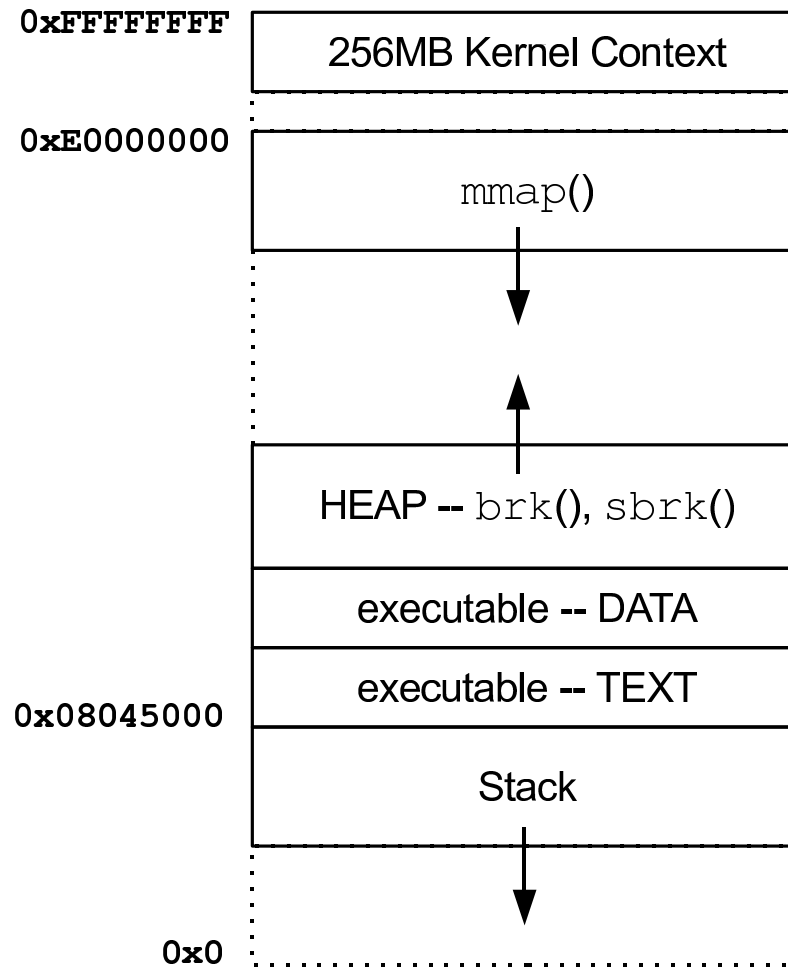
# Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- **manipulace s procesy, spouštění programů**
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

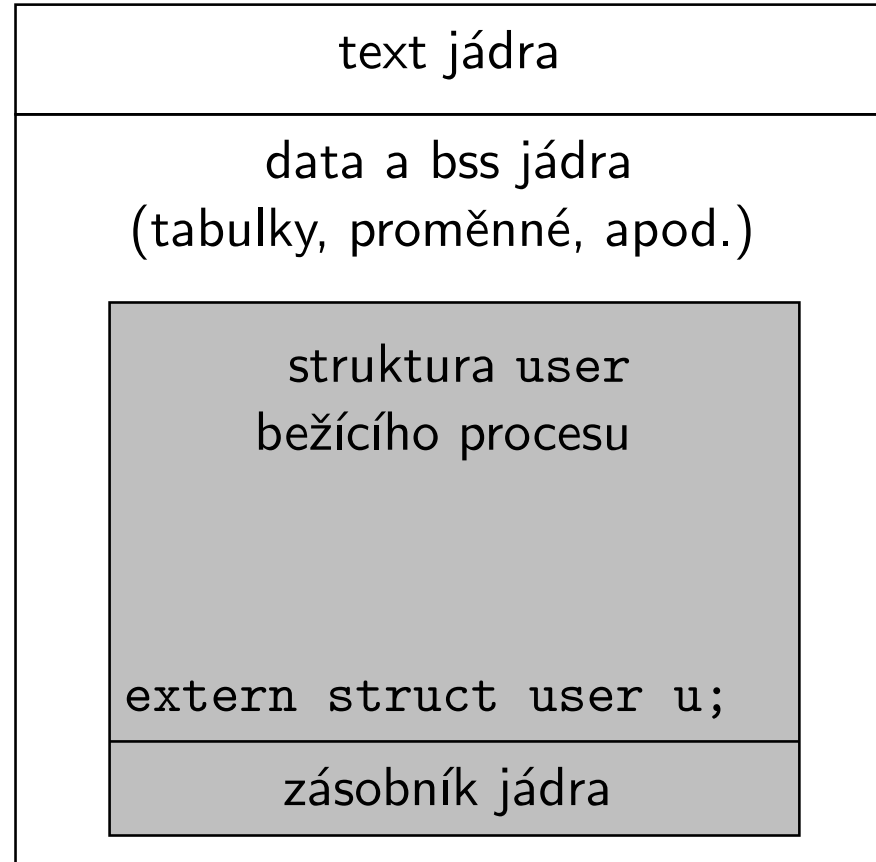
# Paměť procesu v uživatelském režimu



# Příklad: Solaris 11 x86 32-bit

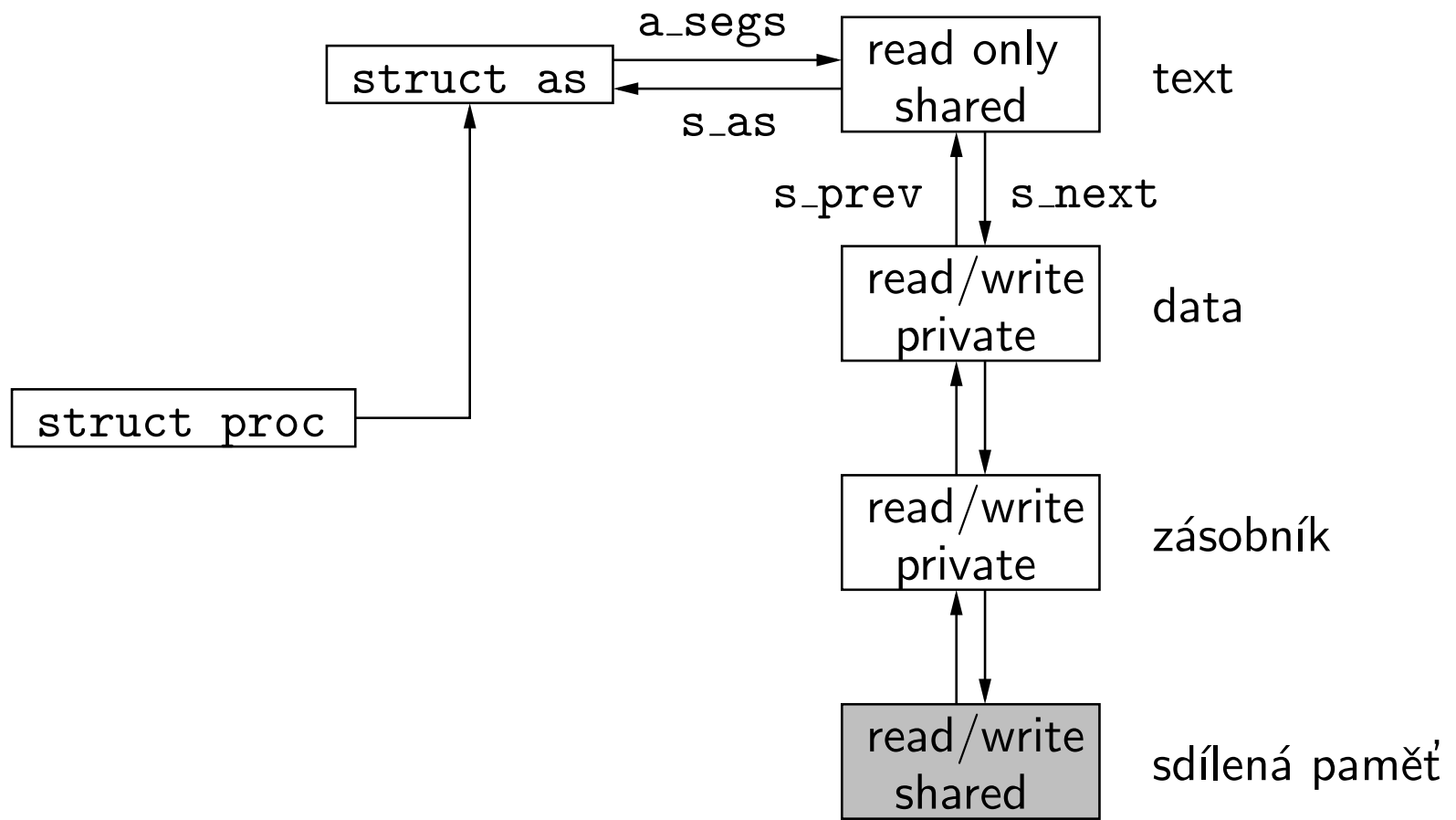


# Paměť procesu v režimu jádra

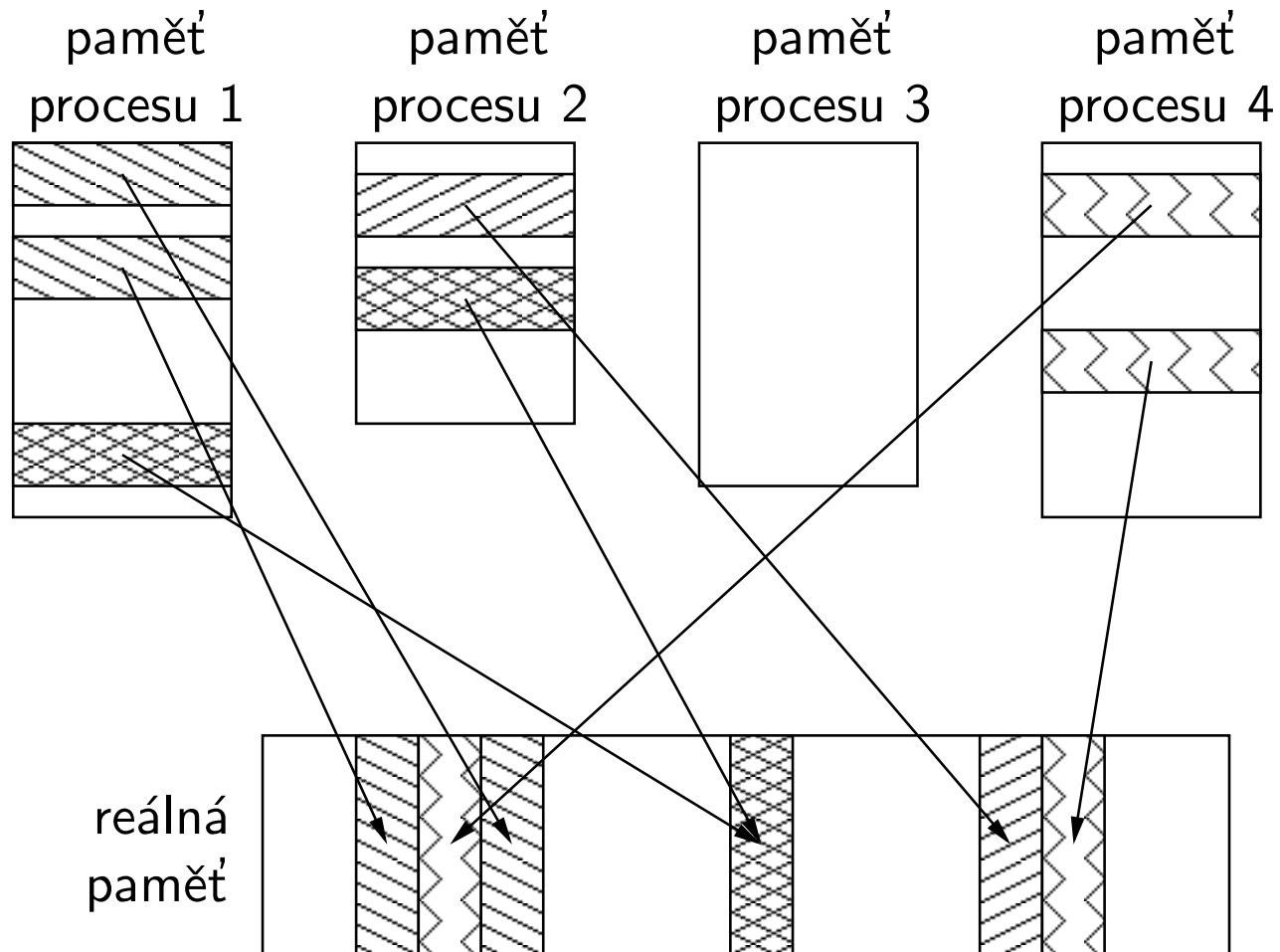




# Paměťové segmenty procesu



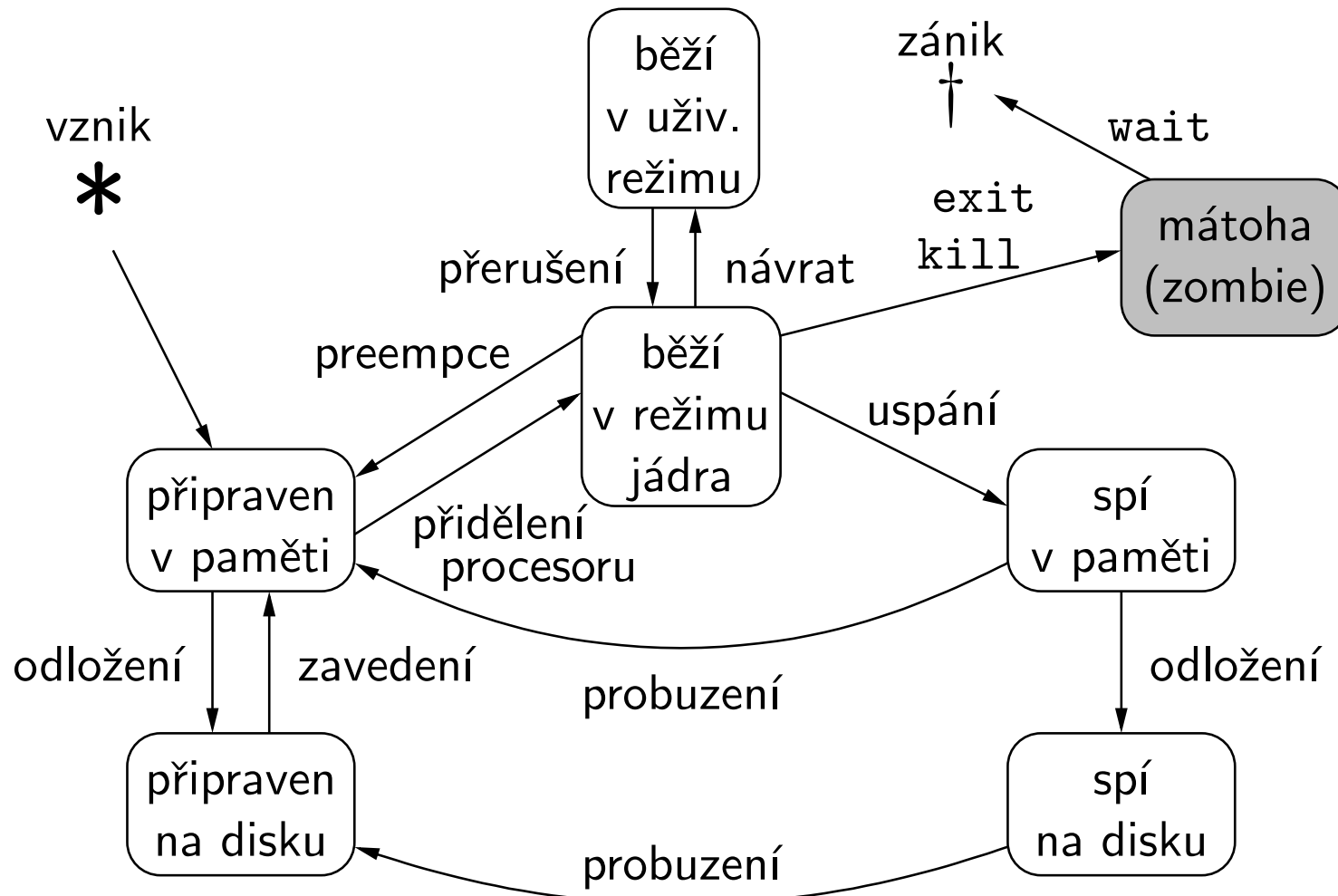
# Virtuální paměť



# Implementace virtuální paměti

- procesy v UNIXu používají k přístupu do paměti virtuální adresy, které na fyzické adresy převádí hardware ve spolupráci s jádrem systému.
- při nedostatku volné paměti se odkládají nepoužívané úseky paměti do odkládací oblasti (**swap**) na disk.
- před verzí SVR2 se procesem **swapper** (nyní **sched**) odkládaly celé procesy.
- od verze SVR2 se používá stránkování na žádost (**demand paging**) a **copy-on-write**. Stránky se alokují až při prvním použití a privátní stránky se kopírují při první modifikaci. Uvolňování a odkládání jednotlivých stránek provádí proces **pageout**, odkládání celých procesů nastupuje až při kritickém nedostatku paměti.

# Stavy procesu



# Plánování procesů

- *preemptivní plánování* – jestliže se proces nevzdá procesoru (neuspí se čekáním na nějakou událost), je mu odebrán procesor po uplynutí časového kvanta.
- procesy jsou zařazeny do front podle priority, procesor je přidělen vždy prvnímu připravenému procesu z fronty, která má nejvyšší prioritu.
- v SVR4 byly zavedeny prioritní třídy a podpora procesů reálného času (real-time) s garantovanou maximální dobou odezvy.
- na rozdíl od předchozích verzí znamená v SVR4 vyšší číslo vyšší prioritu.

# Prioritní třídy

- **systemová**
  - priorita 60 až 99
  - rezervována pro systémové procesy (pageout, sched, ...)
  - pevná priorita
- **real-time**
  - priorita 100 až 159
  - pevná priorita
  - pro každou hodnotu priority definováno časové kvantum
- **sdílení času (time-shared)**
  - priorita 0 až 59
  - proměnná dvousložková priorita, pevná uživatelská a proměnná systémová část – pokud proces hodně využívá procesor, je mu snižována priorita (a zvětšováno časové kvantum)

# Skupiny procesů, řízení terminálů

- každý proces patří do skupiny procesů, tzv. *process group*
- každá skupina může mít vedoucí proces, tzv. *group leader*
- každý proces může mít řídicí terminál (je to obvykle login terminál), tzv. *controlling terminal*
- speciální soubor `/dev/tty` je asociován s řídicím terminálem každého procesu
- každý terminál je asociován se skupinou procesů, tato skupina se nazývá řídicí skupina (*controlling group*)
- kontrola jobů (*job control*) je mechanismus, jak pozastavovat a probouzet skupiny procesů a řídit jejich přístup k terminálům
- *session* (relace) je kolekce skupin procesů vytvořená pro účely řízení jobů

# Identifikace procesu

```
pid_t getpid(void);
```

- vrací proces ID volajícího procesu.

```
pid_t getpgrp(void);
```

- vrací ID skupiny procesů, do které patří volající proces.

```
pid_t getppid(void);
```

- vrací proces ID rodiče.

```
pid_t getsid(pid_t pid);
```

- vrací group ID vedoucího procesu session (sezení, terminálové relace) pro proces *pid* (0 znamená pro volající proces)



# Nastavení skupiny/sezení

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

- nastaví process group ID procesu s *pid* na *pgid*.

```
pid_t setsid(void);
```

- vytvoří novou session, volající proces se stane vedoucím sezení i skupiny procesů.

# Vytvoření procesu: fork()

getpid() == 1234

```
switch(pid = fork()) {  
    case -1: /* Chyba */  
    case 0: /* Dítě */  
    default: /* Rodič */  
}
```

rodič (pokračuje)

dítě (nový proces)

getpid()==1234, pid==2345

```
switch(pid = fork()) {  
    case -1: /* Chyba */  
    case 0: /* Dítě */  
    default: /* Rodič */  
}
```

getpid()==2345, pid==0

```
switch(pid = fork()) {  
    case -1: /* Chyba */  
    case 0: /* Dítě */  
    default: /* Rodič */  
}
```

## Spuštění programu: `exec`

```
extern char **environ;
```

```
int execl(const char *path, const char *arg0, ... );
```

- spustí program definovaný celou cestou `path`, další argumenty se pak předají programu v parametrech `argc` a `argv` funkce `main`. Seznam argumentů je ukončen pomocí `(char *)0`, tj. `NULL`. `arg0` by měl obsahovat jméno programu (tj. ne celou cestu)
- úspěšné volání `execl` se nikdy nevrátí, protože spuštěný program zcela nahradí dosavadní adresový prostor procesu.
- program dědí proměnné prostředí, tj. obsah `environ`.
- `handlers` signálů se nahradí implicitní obsluhou
- zavřou se deskriptory souborů, které mají nastavený příznak `FD_CLOEXEC` (implicitně není nastaven).

## Varianty služby exec

```
int execl(const char *path, char *const argv []);
```

- obdoba `execl()`, ale argumenty jsou v poli `argv`, jehož poslední prvek je `NULL`.

```
int execlp(const char *path, const char *arg0, ... ,  
           char *const envp []);
```

- obdoba `execl()`, ale místo `environ` se použije `envp`.

```
int execve(const char *path, char *const argv [],  
           char *const envp []);
```

- obdoba `execv()`, ale místo `environ` se použije `envp`.

```
int execlp(const char *file, const char *arg0, ...);
```

```
int execvp(const char *file, char *const argv []);
```

- obdoby `execl()` a `execv()`, ale pro hledání spustitelného souboru se použije proměnná `PATH`.

# Formát spustitelného souboru

- **Common Object File Format (COFF)** – starší System V
- **Extensible Linking Format (ELF)** – nový v SVR4
- často se mluví o **a.out** formátu, protože tak se jmenuje (pokud není použit přepínač `-o`) výstup linkeru.

- Formát ELF:

hlavička souboru
tabulka programových hlaviček
sekce 1
⋮
sekce N
tabulka hlaviček sekcí

# Ukončení procesu

```
void exit(int status);
```

- ukončí proces s návratovým kódem `status`.
- nikdy se nevrátí na instrukci následující za voláním.

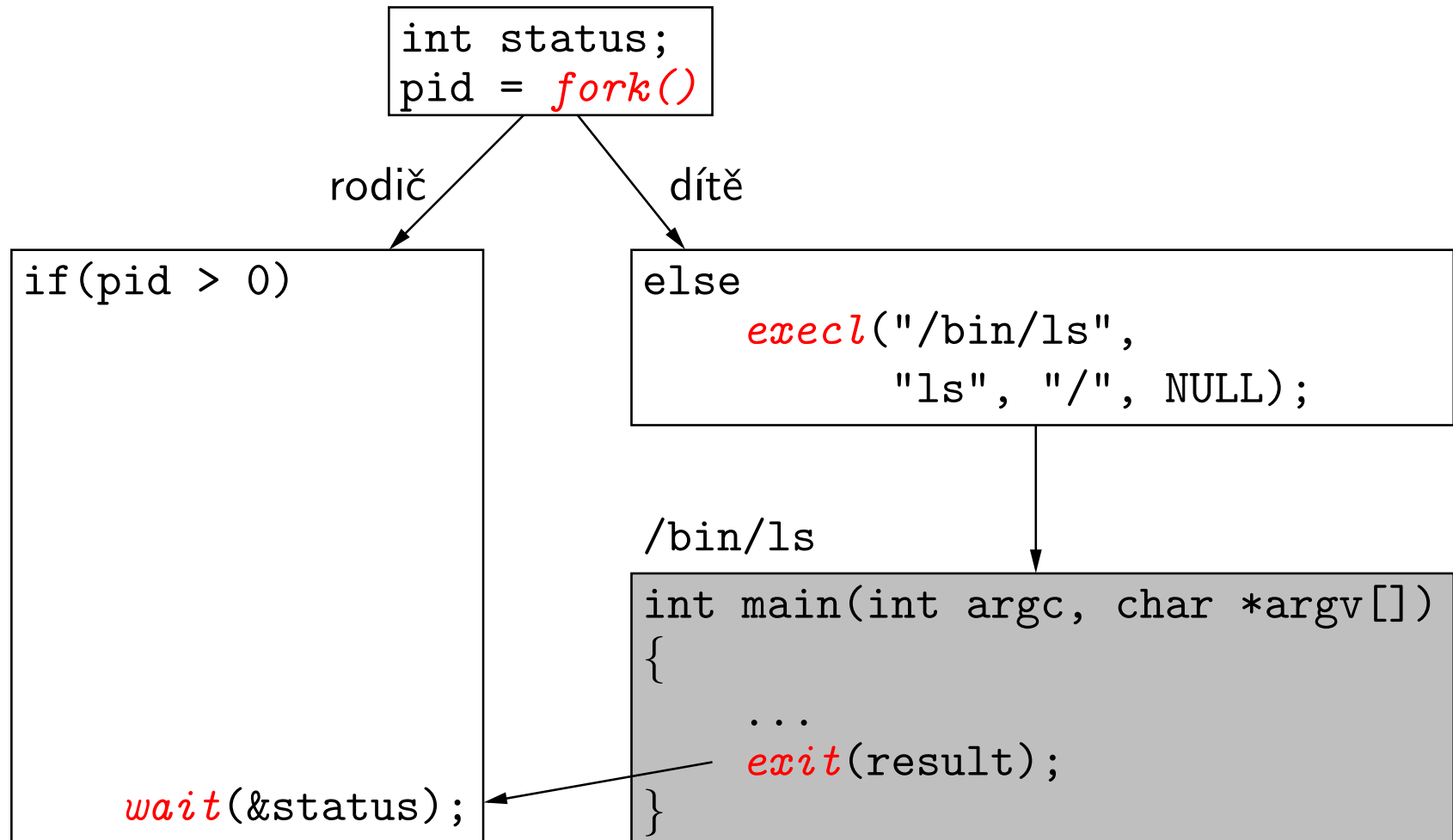
```
pid_t wait(int *stat_loc);
```

- počká, až skončí některý synovský proces, vrátí jeho PID a do `stat_loc` uloží návratový kód, který lze dále testovat:
  - `WIFEXITED(stat_val)` ... proces volal `exit()`
  - `WEXITSTATUS(stat_val)` ... argument `exit()`
  - `WIFSIGNALED(stat_val)` ... proces dostal signál
  - `WTERMSIG(stat_val)` ... číslo signálu
  - `WIFSTOPPED(stat_val)` ... proces pozastaven
  - `WSTOPSIG(stat_val)` ... číslo signálu

```
pid_t waitpid(pid_t pid, int *stat_loc, int opts);
```

- čekání na jeden proces.

# Příklad: spuštění programu a čekání



## Roura: `pipe()`

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

- vytvoří rouru a dva deskriptory
  - `filides[0]` ... čtení z roury
  - `filides[1]` ... zápis do roury
- roura zajišťuje synchronizaci čtení a zápisu:
  - zapisující proces se zablokuje, když je roura plná,
  - čtoucí proces se zablokuje, když je roura prázdná.
- čtoucí proces přečte konec souboru (tj. `read()` vrátí 0), pokud jsou uzavřeny všechny kopie `filides[1]`.
- pojmenovaná roura (vytvořená voláním `mkfifo()`) funguje stejně, ale má přidělené jméno v systému souborů a mohou ji tedy používat libovolné procesy.



# Příklad: roura mezi dvěma procesy

shell: `ls / | more`

```
int pd[2];  
pipe(pd);  
switch(fork()) {
```

producent (dítě)

```
case 0:  
    close(1);  
    dup(pd[1]);  
    close(pd[0]);  
    close(pd[1]);  
    execl("/bin/ls", "ls",  
          "/", NULL);
```

konzument (rodič)

```
default:  
    close(0);  
    dup(pd[0]);  
    close(pd[0]);  
    close(pd[1]);  
    execl("/bin/more",  
          "more", NULL);
```



## Sdílená paměť – úvod

- pípky a soubory jako metody meziprocesové komunikace vyžadují systémová volání
- výhoda: procesy nemohou poškodit adresový prostor jiného procesu
- nevýhoda: velká režie pro systémová volání, typicky **read**, **write**
- sdílená paměť je namapování části paměti do adresového prostoru více procesů
- odstranění nevýhody, ztráta dosavadní výhody
- synchronizace přístupu do sdílené paměti
  - System V semaforey
  - POSIX semaforey bez nutnosti systémového volání v běžném případě

# Mapování souborů do paměti (1)

```
void *mmap(void *addr, size_t len, int prot, int flags,  
           int fildes, off_t off);
```

- do paměťového prostoru procesu od adresy `addr` (0 ... adresu přidělí jádro) namapuje úsek délky `len` začínající na pozici `off` souboru reprezentovaného deskriptorem `fildes`.
- vrací adresu namapovaného úseku nebo `MAP_FAILED`.
- v `prot` je OR-kombinace `PROT_READ` (lze číst), `PROT_WRITE` (lze zapisovat), `PROT_EXEC` (lze spouštět), nebo `PROT_NONE` (nelze k datům přistupovat).
- ve `flags` je OR-kombinace `MAP_PRIVATE` (změny jsou privátní pro proces, neukládají se do souboru), `MAP_SHARED` (změny se ukládají do souboru), `MAP_FIXED` (jádro nezmění `addr`).

## Mapování souborů do paměti (2)

```
int msync(void *addr, size_t len, int flags);
```

- zapíše změněné stránky v úseku len bajtů od adresy addr do souboru. Hodnota flags je OR-kombinace
  - MS\_ASYNC ... asynchronní zápis
  - MS\_SYNC ... synchronní zápis
  - MS\_INVALIDATE ... zrušit namapovaná data, která se liší od obsahu souboru

```
int munmap(void *addr, size_t len);
```

- zapíše změny, zruší mapování v délce len od adresy addr.

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

- změní přístupová práva k namapovanému úseku souboru. Hodnoty prot jsou stejné jako u mmap().

## Příklad: mapování souborů do paměti

```
int main(int argc, char *argv[])
{
    int fd, fsz; char *addr, *p1, *p2, c;

    fd = open(argv[1], O_RDWR);
    fsz = lseek(fd, 0, SEEK_END);
    p1 = addr = mmap(0, fsz, PROT_READ|PROT_WRITE,
                     MAP_SHARED, fd, 0);

    p2 = p1 + fsz - 1;
    while(p1<p2) {
        c = *p1; *p1++ = *p2; *p2-- = c;
    }
    munmap(addr, fsz);
    close(fd);
    return (0);
}
```

# Dynamický přístup ke knihovnám

```
void *dlopen(const char *file, int mode);
```

- zpřístupní knihovnu v souboru *file*, vrátí **handle** nebo NULL.
- v *mode* je OR-kombinace RTLD\_NOW (okamžité relopace), RTLD\_LAZY (odložené relopace), RTLD\_GLOBAL (symboly budou globálně dostupné), RTLD\_LOCAL (nebudou globálně dostupné).

```
void *dlsym(void *handle, const char *name);
```

- vrátí adresu symbolu zadaného jména z knihovny.

```
int dlclose(void *handle);
```

- ukončí přístup ke knihovně.

```
char *dlerror(void);
```

- vrátí textový popis chyby při práci s knihovnami.

## Příklad: zpřístupnění knihovny

```
char *err;
void *handle;
double y, x = 1.3;
double (*fun)(double);
char *libname = "libm.so", *fn_name = "sin";

if ((handle = dlopen(libname, RTLD_NOW)) == NULL) {
    fprintf(stderr, "%s\n", dlerror()); exit(1);
}

fun = dlsym(handle, fn_name);
if ((err = dlerror()) != NULL)
    fprintf(stderr, "%s\n", err); exit(1);

y = fun(x);
dlclose(handle);
```

# Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- **signály**
- synchronizace a komunikace procesů
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času



# Signály

- informují proces o výskytu určité události
- na uživatelské úrovni zpřístupňují mechanismy přerušení
- kategorie signálů:
  - **chybové události** generované běžícím procesem, např. pokus o přístup mimo přidělenou oblast paměti (**SIGSEGV**)
  - **asynchronní události** vznikající mimo proces, např. signál od jiného procesu, vypršení nastaveného času (**SIGALRM**), odpojení terminálu (**SIGHUP**), stisk **Ctrl-C** (**SIGINT**)
- nejjednodušší mechanismus pro komunikaci mezi procesy – nesou pouze informaci o tom, že nastala nějaká událost.
- většinou se zpracovávají asynchronně – příchod signálu přeruší běh procesu a vyvolá se obslužná funkce, tzv. *handler signálu*

# Poslání signálu

```
int kill(pid_t pid, int sig);
```

- pošle signál s číslem `sig` procesu (nebo skupině procesů) podle hodnoty `pid`:
  - `> 0` ... procesu s číslem `pid`
  - `== 0` ... všem procesům ve stejné skupině
  - `== -1` ... všem procesům, kromě systémových
  - `< -1` ... procesům ve skupině `abs(pid)`
- `sig == 0` znamená, že se pouze zkontroluje oprávnění poslat signál, ale žádný signál se nepošle.
- právo procesu poslat signál jinému procesu závisí na UID obou procesů.

# Ošetření signálů

- pokud proces neřekne jinak, provede se v závislosti na konkrétním signálu implicitní akce, tj. bud':
    - ukončení procesu (**exit**)
    - ukončení procesu plus coredump (**core**)
    - ignorování signálu (**ignore**)
    - pozastavení procesu (**stop**)
    - pokračování pozastaveného procesu (**continue**)
  - proces také může nastavit ignorování signálu
  - nebo signál ošetření uživatelsky definovanou funkcí (**handler**), po návratu z handleru proces pokračuje od místa přerušení
- signály SIGKILL a SIGSTOP vždy vyvolají implicitní akci (zrušení, resp. pozastavení).

# Přehled signálů (1)

signály je možné logicky rozdělit do několika skupin...

## detekované chyby:

SIGBUS	přístup k nedef. části paměťového objektu (core)
SIGFPE	chyba aritmetiky v pohyblivé čárce (core)
SIGILL	nepovolená instrukce (core)
SIGPIPE	zápis do roury, kterou nikdo nečte (exit)
SIGSEGV	použití nepovolené adresy v paměti (core)
SIGSYS	chybné systémové volání (core)
SIGXCPU	překročení časového limitu CPU (core)
SIGXFSZ	překročení limitu velikosti souboru (core)

## Přehled signálů (2)

generované uživatelem nebo aplikací:

SIGABRT	ukončení procesu (core)
SIGHUP	odpojení terminálu (exit)
SIGINT	stisk speciální klávesy Ctrl-C (exit)
SIGKILL	zrušení procesu (exit, <b>nelze ošetřit ani ignorovat</b> )
SIGQUIT	stisk speciální klávesy Ctrl-\ (core)
SIGTERM	zrušení procesu (exit)
SIGUSR1	uživatelsky definovaný signál 1 (exit)
SIGUSR2	uživatelsky definovaný signál 2 (exit)

## Přehled signálů (3)

### job control:

SIGCHLD	změna stavu synovského procesu (ignore)
SIGCONT	pokračování pozastaveného procesu (continue)
SIGSTOP	pozastavení (stop, <b>nelze ošetřit ani ignorovat</b> )
SIGTSTP	pozastavení z terminálu Ctrl-Z (stop)
SIGTTIN	čtení z terminálu procesem na pozadí (stop)
SIGTTOU	zápis na terminál procesem na pozadí (stop)

- součástí nepovinného POSIX rozšíření, existují pouze když v `<unistd.h>` je definováno makro `_POSIX_JOB_CONTROL`

## Přehled signálů (4)

### časovače:

SIGALRM pláňované časové přerušení (exit)

SIGPROF vypršení profilujícího časovače (exit)

SIGVTALRM vypršení virtuálního časovače (exit)

### různé:

SIGPOLL testovatelná událost (exit)

SIGTRAP ladicí přerušení (core)

SIGURG urgentní událost na soketu (ignore)

## Nastavení obsluhy signálů

```
int sigaction(int sig, const struct sigaction *act,  
              struct sigaction *oact);
```

- nastaví obsluhu signálu `sig` podle `act` a vrátí předchozí nastavení v `oact`.
- obsah struktury `sigaction`:
  - `void (*sa_handler)(int) ... SIG_DFL, SIG_IGN, nebo adresa handleru`
  - `sigset_t sa_mask ... signály blokované v handleru, navíc je blokován signál sig`
  - `int sa_flags ... SA_RESETHAND (při vstupu do handleru nastavit SIG_DFL), SA_RESTART (restartovat přerušená systémová volání), SA_NODEFER (neblokovat signál sig během obsluhy)`



## Příklad: časově omezený vstup

```
void handler(int sig)
{ write(2, " !!! TIMEOUT !!! \n", 17); }

int main(void)
{
    char buf[1024]; struct sigaction act; int sz;
    act.sa_handler = handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGALRM, &act, NULL);
    alarm(5);
    sz = read(0, buf, 1024);
    alarm(0);
    if (sz > 0)
        write(1, buf, sz);
    return (0);
}
```

# Blokování signálů

- blokované signály budou procesu doručeny a zpracovány až po odblokování.

```
int sigprocmask(int how, const sigset_t *set,  
               sigset_t *oset);
```

- nastaví masku blokováných signálů a vrátí starou masku.
- *how* – SIG\_BLOCK pro přidání signálů co se mají blokovat, pro odebrání SIG\_UNBLOCK, pro kompletní změnu masky SIG\_SETMASK
- pro manipulaci s maskou signálů slouží funkce: sigaddset(), sigdelset(), sigemptyset(), sigfillset(), sigismember()

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

- vrátí čekající zablokované signály.

## Příklad: blokování signálů

```
sigset_t sigs, osigs; struct sigaction sa;
sigfillset(&sigs); sigprocmask(SIG_BLOCK, &sigs, &osigs);
switch(cpid = fork()) {
    case -1: /* Chyba */
        sigprocmask(SIG_SETMASK, &osigs, NULL);
        ...
    case 0: /* Synovský proces */
        sa.sa_handler = h_cld; sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        sigaction(SIGINT, &sa, NULL);
        sigprocmask(SIG_SETMASK, &osigs, NULL);
        ...
    default: /* Rodičovský proces */
        sigprocmask(SIG_SETMASK, &osigs, NULL);
        ...
}
```

# Čekání na signál

```
int pause(void);
```

- pozastaví volající proces do příchodu (neblokovaného) signálu

```
int sigsuspend(const sigset_t *sigmask);
```

- jako `pause()`, ale navíc po dobu čekání masku blokováných signálů změní na `sigmask`

```
int sigwait(const sigset_t *set, int *sig);
```

- čeká na příchod signálu z množiny `set` (tyto signály musí být předtím zablokované), číslo signálu vrátí v `sig`. Vrací 0 nebo číslo chyby.
- nevolá se handler signálu (to ale není v normě jednoznačně definováno)

# Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- **synchronizace a komunikace procesů**
- síťová komunikace
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času

# Problém: konflikt při sdílení dat

- máme strukturu `struct { int a, b; } shared;`
- ```
for( ; ; ) {  
    /* neatomická operace */  
    a = shared.a; b = shared.b;  
    if (a != b) printf("NEKONZISTENTNÍ STAV");  
    /* neatomická operace */  
    shared.a = val; shared.b = val;  
}
```
- jestliže tento cyklus spustíme ve dvou různých procesech (nebo vláknech), které obě sdílejí stejnou strukturu `shared` a mají různé hodnoty `val`, bude docházet ke konfliktům.
- příčina: operace na zvýrazněných řádcích nejsou atomické.

# Scénář konfliktu

Procesy **A** (val==1) a **B** (val==2)

1. počáteční stav struktury
2. proces **A** zapíše do položky a
3. proces **B** zapíše do položky a
4. proces **B** zapíše do položky b
5. proces **A** zapíše do položky b
6. struktura je v nekonzistentním stavu a jeden z procesů to zjistí.

|    | a | b |
|----|---|---|
| 1. | ? | ? |
| 2. | 1 | ? |
| 3. | 2 | ? |
| 4. | 2 | 2 |
| 5. | 2 | 1 |

## Řešení: vzájemné vyloučení procesů

- je potřeba zajistit atomicitu operací nad strukturou, tzn. jeden proces provádí modifikaci a dokud neuvede strukturu do konzistentního stavu, druhý proces s ní nemůže manipulovat.

Procesy **A** (val==1) a **B** (val==2)

|                                        | a | b |
|----------------------------------------|---|---|
| 1. počáteční stav struktury            | ? | ? |
| 2. proces <b>A</b> zapíše do položky a | 1 | ? |
| 3. proces <b>B</b> musí čekat          | 1 | ? |
| 4. proces <b>A</b> zapíše do položky b | 1 | 1 |
| 5. proces <b>B</b> zapíše do položky a | 2 | 1 |
| 6. proces <b>B</b> zapíše do položky b | 2 | 2 |
| 7. Struktura je v konzistentním stavu. |   |   |



# Problém: konflikt zapisovatelů a čtenářů

- několik běžících procesů zapisuje protokol o své činnosti do společného log-souboru. Nový záznam je připojen vždy na konec souboru.
- pokud zápis záznamu není proveden atomickou operací, může dojít k promíchání více současně zapisovaných záznamů.
- zapisovat smí vždy pouze jeden proces.
- další procesy čtou data z log-souboru.
- při přečtení právě zapisovaného záznamu obdržíme nesprávná (neúplná) data.
- během operace zápisu ze souboru nelze číst. Když nikdo nezapisuje, může více procesů číst současně.

## Řešení: zamykání souborů

- zapisující proces zamkne soubor pro zápis. Ostatní procesy (zapisovatelé i čtenáři) se souborem nemohou pracovat a musí čekat na odemčení zámku.
- čtoucí proces zamkne soubor pro čtení. Zapisovatelé musí čekat na odemčení zámku, ale ostatní čtenáři mohou také zamknout soubor pro čtení a číst data.
- v jednom okamžiku může být na souboru aktivní nejvýše jeden zámeček pro zápis nebo libovolně mnoho zámečků pro čtení, ale ne oba typy zámečků současně.
- z důvodu efektivity by každý proces měl držet zámeček co nejkratší dobu a pokud možno nezamykat celý soubor, ale jen úsek, se kterým pracuje. Preferované je pasivní čekání, aktivní čekání je vhodné jen na velmi krátkou dobu.

# Synchronizační mechanismy

- teoretické řešení – algoritmy vzájemného vyloučení (Dekker 1965, Peterson 1981)
- zákaz přerušování (na 1 CPU stroji), speciální *test-and-set* instrukce
- **lock-soubory**
- nástroje poskytované operačním systémem
  - **semafony** (součást System V IPC)
  - **zámky pro soubory** (`fcntl()`, `flock()`)
  - synchronizace vláken: **mutexy** (ohraničují kritické sekce, pouze jedno vlákno může držet mutex), **podmínkové proměnné** (zablokují vlákno, dokud jiné vlákno nesignalizuje změnu podmínky), **read-write zámky** (sdílené a exkluzivní zámky, podobně jako pro soubory)

# Lock-soubory

- pro každý sdílený zdroj existuje dohodnuté jméno souboru. Zamčení zdroje se provede vytvořením souboru, odemčení smazáním souboru. Každý proces musí otestovat, zda soubor existuje, a pokud ano, tak počkat.

```
void lock(char *lockfile) {  
    while( (fd = open(lockfile,  
                      O_RDWR|O_CREAT|O_EXCL, 0600)) == -1)  
        sleep(1); /* Čekáme ve smyčce na odemčení */  
    close(fd);  
}
```

```
void unlock(char *lockfile) {  
    unlink(lockfile);  
}
```

## Zamykání souborů: `fcntl()`

```
int fcntl(int fildes, int cmd, ...);
```

- k nastavení zámků pro soubor `fil`des se používá `cmd`:
  - `F_GETLK` ... vezme popis zámku z třetího argumentu a nahradí ho popisem existujícího zámku, který s ním koliduje
  - `F_SETLK` ... nastaví nebo zruší zámeček popsaný třetím argumentem, pokud nelze zámeček nastavit, ihned vrátí `-1`
  - `F_SETLKW` ... jako `F_SETLK`, ale uspí proces, dokud není možné nastavit zámeček (`W` znamená “wait”)
- třetí argument obsahuje popis zámku a je typu ukazatel na `struct flock`

# Zamykání souborů: struct flock

- `l_type` ... typ zámku
  - `F_RDLCK` ... sdílený zámeček (pro čtení), více procesů
  - `F_WRLCK` ... exkluzivní zámeček (pro zápis), jeden proces
  - `F_UNLCK` ... odemčení
- `l_whence` ... jako u `lseek()`, tj. `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
- `l_start` ... začátek zamykaného úseku vzhledem k `l_whence`
- `l_len` ... délka úseku v bajtech, 0 znamená do konce souboru
- `l_pid` ... PID procesu držícího zámeček, používá se jen pro `F_GETLK` při návratu

# Deadlock (aka uváznutí)

- máme dva sdílené zdroje `res1` a `res2` chráněné zámky `lck1` a `lck2`. Procesy `p1` a `p2` chtějí každý výlučný přístup k oběma zdrojům.

p1

```
lock(lck1); /* OK */  
lock(lck2); /* Čeká na p2 */
```

p2

```
lock(lck2); /* OK */  
lock(lck1); /* Čeká na p1 */
```

Deadlock

```
use(res1, res2);  
unlock(lck2);  
unlock(lck1);
```

```
use(res1, res2);  
unlock(lck2);  
unlock(lck1);
```

- pozor na pořadí zamykání!

# System V IPC

- **IPC** je zkratka pro *Inter-Process Communication*
- komunikace mezi procesy **v rámci jednoho systému**, tj. nezahrnuje síťovou komunikaci
- **semafony** ... použití pro synchronizaci procesů
- **sdílená paměť** ... předávání dat mezi procesy, přináší podobné problémy jako sdílení souborů, k řešení lze použít semafony
- **fronty zpráv** ... spojují komunikaci (zpráva nese data) se synchronizací (čekání procesu na příchod zprávy)
- prostředky IPC mají podobně jako soubory definovaná **přístupová práva** (pro čtení a zápis) pro vlastníka, skupinu a ostatní.



# Semaforey

- zavedl je E. Dijkstra
- semafor  $s$  je datová struktura obsahující
  - celé nezáporné číslo  $i$  (volná kapacita)
  - frontu procesů  $q$ , které čekají na uvolnění
- operace nad semaforem:

**init( $s, n$ )**

vyprázdnit  $s.q$ ;  $s.i = n$

**P( $s$ )**

if( $s.i > 0$ )  $s.i--$  else

uspat volající proces a zařadit do  $s.q$

**V( $s$ )**

if( $s.q$  prázdná)  $s.i++$  else

odstranit jeden proces z  $s.q$  a probudit ho

# Vzájemné vyloučení pomocí semaforů

- jeden proces inicializuje semafor

```
sem s;  
init(s, 1);
```

- kritická sekce se doplní o operace nad semaforem

```
...
```

```
P(s);
```

```
kritická sekce;
```

```
V(s);
```

```
...
```

## API pro semaforey

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

- vrátí identifikátor pole obsahujícího `nsems` semaforů asociovaný s klíčem `key` (klíč `IPC_PRIVATE` ... privátní semaforey, při každém použití vrátí jiný identifikátor). `semflg` je OR-kombinace přístupových práv a konstant `IPC_CREAT` (vytvořit, pokud neexistuje), `IPC_EXCL` (chyba, pokud existuje).

```
int semctl(int semid, int semnum, int cmd, ...);
```

- řídicí funkce, volitelný čtvrtý parametr `arg` je typu `union semun`.

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

- zobecněné operace P a V.

## API pro semaforey: `semctl()`

- `semnum` ... číslo semaforu v poli
- možné hodnoty `cmd`:
  - `GETVAL` ... vrátí hodnotu semaforu
  - `SETVAL` ... **nastaví semafor na hodnotu `arg.val`**
  - `GETPID` ... PID procesu, který provedl poslední operaci
  - `GETNCNT` ... počet procesů čekajících na větší hodnotu
  - `GETZCNT` ... počet procesů čekajících na nulu
  - `GETALL` ... uloží hodnoty všech semaforů do pole `arg.array`
  - `SETALL` ... nastaví všechny semaforey podle `arg.array`
  - `IPC_STAT` ... do `arg.buf` dá počet semaforů, přístupová práva a časy posledních `semctl()` a `semop()`
  - `IPC_SET` ... nastaví přístupová práva
  - `IPC_RMID` ... zruší pole semaforů

## API pro semaforey: semop()

- operace se provádí atomicky (tj. buď se povede pro všechny semaforey, nebo pro žádný) na `nsops` semaforech podle pole `sops` struktur `struct sembuf`, která obsahuje:
  - `sem_num` ... číslo semaforu
  - `sem_op` ... operace
    - \* `P(sem_num, abs(sem_op))` pro `sem_op < 0`
    - \* `V(sem_num, sem_op)` pro `sem_op > 0`
    - \* čekání na nulovou hodnotu semaforu pro `sem_op == 0`
  - `sem_flg` ... OR-kombinace
    - \* `IPC_NOWAIT` ... když nelze operaci hned provést, nečeká a vrátí chybu
    - \* `SEM_UNDO` ... při ukončení procesu vrátit operace se semaforem

# Vytváření prostředků IPC

- jeden proces prostředek vytvoří, ostatní se k němu připojí.
- po skončení používání je třeba prostředek IPC zrušit.
- funkce `semget()`, `shmget()` a `msgget()` mají jako první parametr klíč identifikující prostředek IPC. Skupina procesů, která chce komunikovat, se musí domluvit na společném klíči. Různé skupiny komunikujících procesů by měly mít různé klíče.

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

- vrátí klíč odvozený ze zadaného jména souboru `path` a čísla `id`. Pro stejné `id` a libovolnou cestu odkazující na stejný soubor vrátí stejný klíč. Pro různá `id` nebo různé soubory na stejném svazku vrátí různé klíče.

## Další prostředky IPC

- POSIX a SUSv3 definují ještě další prostředky komunikace mezi procesy:
  - **signály** ... pro uživatelské účely lze využít signály SIGUSR1 a SIGUSR2
  - **POSIXová sdílená paměť** přístupná pomocí shm\_open() a mmap()
  - **POSIXové semaforey** ... sem\_open(), sem\_post(), sem\_wait(), ...
  - **POSIXové fronty zpráv** ... mq\_open(), mq\_send(), mq\_receive(), ...
- Z BSD pochází **sokety (sockets)** umožňující komunikaci v doménách AF\_UNIX (komunikace v rámci jednoho počítače) a AF\_INET (komunikace na jednom počítači nebo po síti).

# Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- **síťová komunikace**
- vlákna, synchronizace vláken
- ??? - bude definováno později, podle toho kolik zbyde času



# Síťová komunikace

**UUCP (UNIX-to-UNIX Copy Program)** – první aplikace pro komunikaci UNIXových systémů propojených přímo nebo přes modemy, součást Version 7 UNIX (1978)

**sokety (sockets)** – zavedeny ve 4.1aBSD (1982); soket je jeden konec obousměrného komunikačního kanálu vytvořeného mezi dvěma procesy buď lokálně na jednom počítači, nebo s využitím síťového spojení

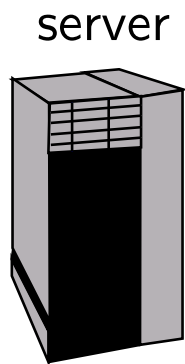
**TLI (Transport Layer Interface)** – SVR3 (1987); knihovna zajišťující síťovou komunikaci na úrovni 4. vrstvy referenčního modelu ISO OSI

**RPC (Remote Procedure Call)** – SunOS (1984); protokol pro přístup ke službám na vzdáleném stroji, data přenášena ve tvaru XDR (External Data Representation)

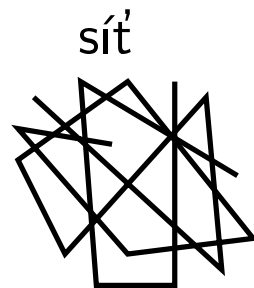
# TCP/IP

- protokoly
  - **IP (Internet Protocol)** – přístupný jen pro uživatele root
  - **TCP (Transmission Control Protocol)** – streamový, spojovaný, spolehlivý
  - **UDP (User Datagram Protocol)** – datagramový, nespojovaný, nespolehlivý
- **IP adresa** – 4 bajty (IPv4) / 16 bajtů (IPv6), definuje síťové rozhraní, nikoliv počítač
- **port** – 2 bajty, rozlišení v rámci 1 IP adresy, porty s číslem menším než 1024 jsou rezervované (jejich použití vyžaduje práva uživatele root)
- **DNS (Domain Name System)** – převod mezi symbolickými jmény a numerickými IP adresami

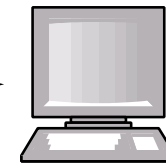
# Spojované služby (TCP), sekvenční obsluha



server



sít'



klient

```
fd = socket()
```

```
bind(fd)
```

```
listen(fd)
```

```
fd2 = accept(fd)
```

```
read(fd2); write(fd2)
```

```
close(fd2)
```

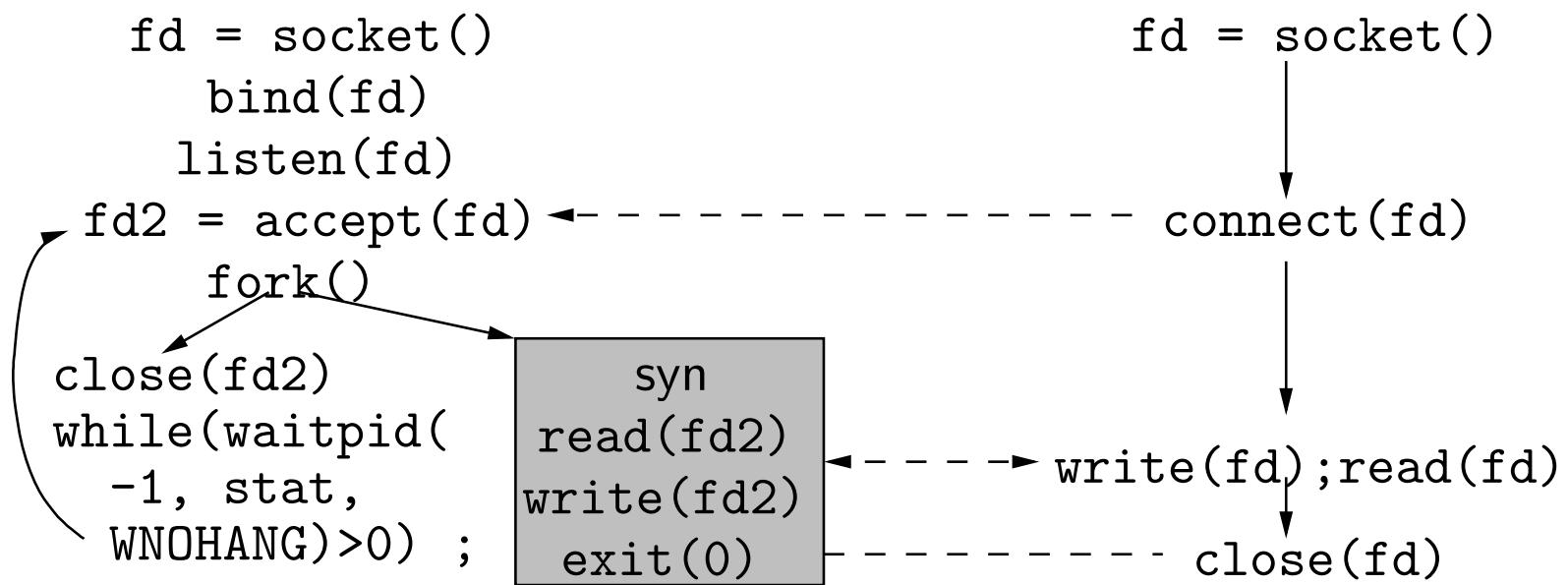
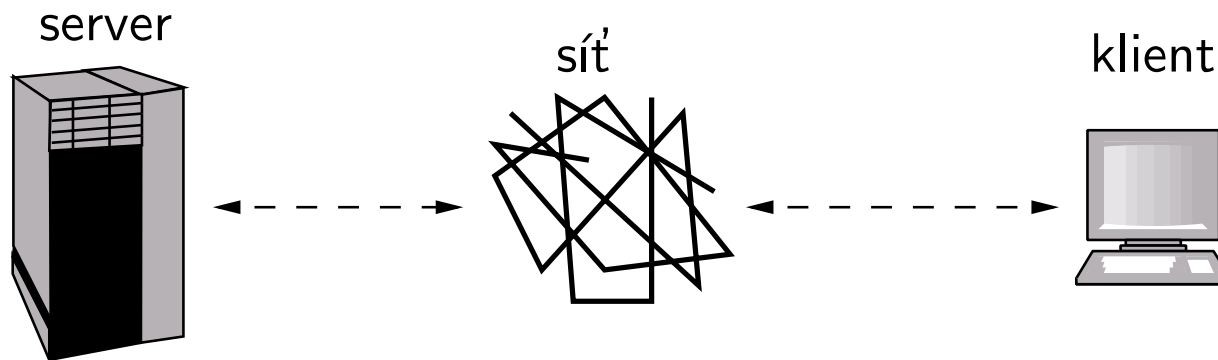
```
fd = socket()
```

```
connect(fd)
```

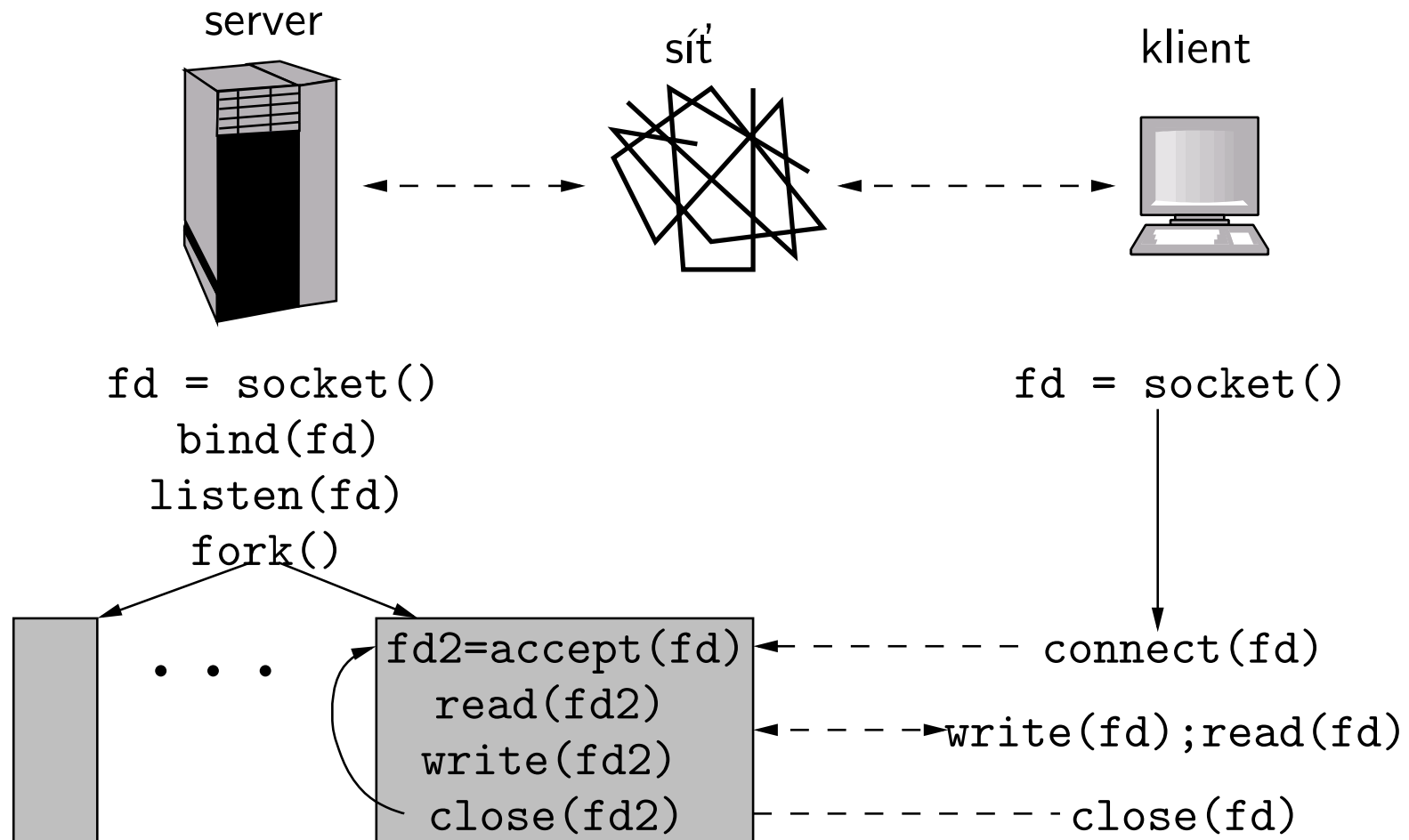
```
write(fd); read(fd)
```

```
close(fd)
```

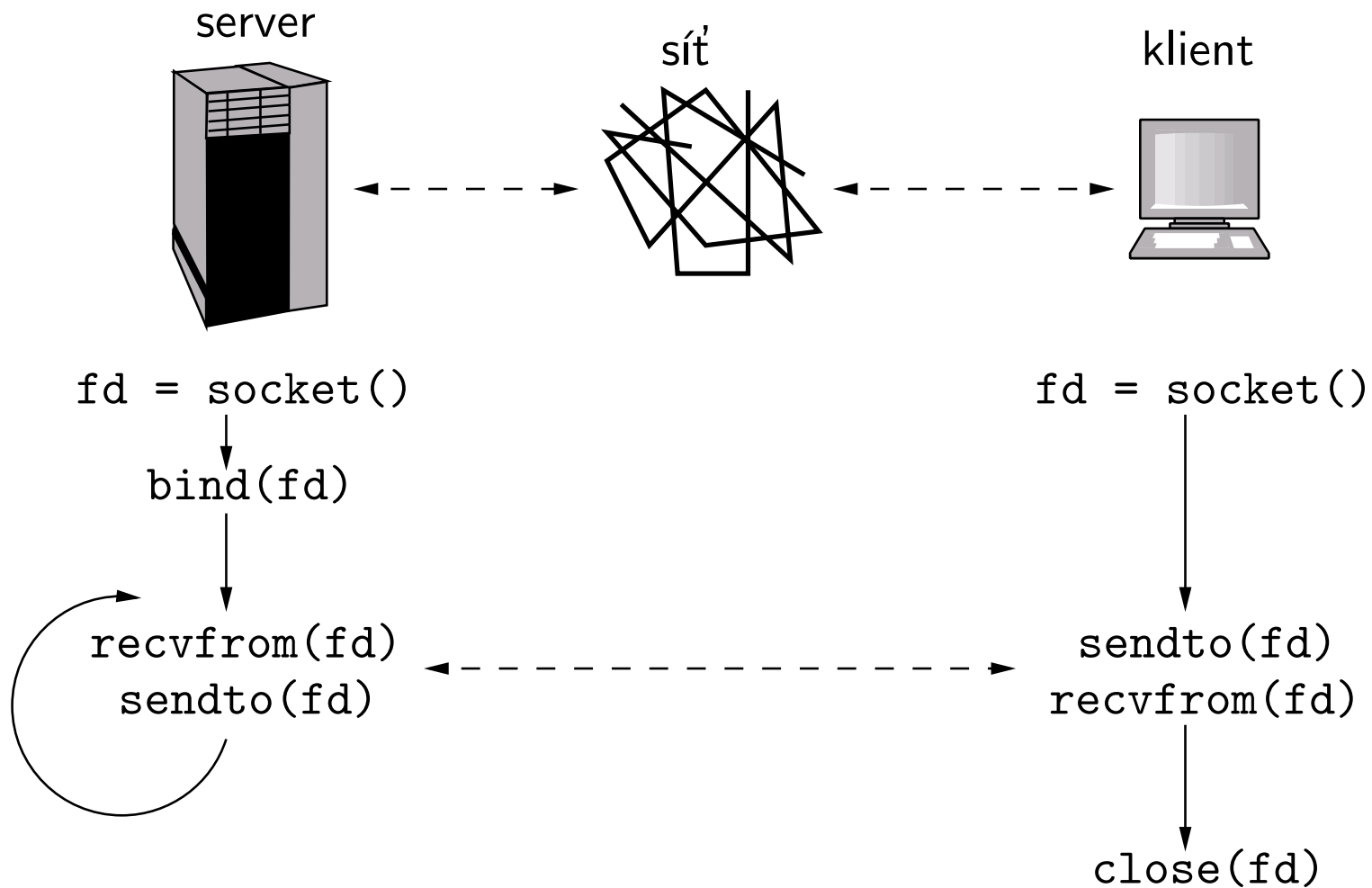
# Spojované služby (TCP), paralelní obsluha



# Spojované služby, paralelní accept()



# Datagramové služby (UDP)



## Vytvoření socketu: `socket()`

```
int socket(int domain, int type, int protocol);
```

- vytvoří socket a vrátí jeho deskriptor.
- `domain` – „kde se bude komunikovat“:
  - `AF_UNIX` ... lokální komunikace, adresa je jméno souboru
  - `AF_INET`, `AF_INET6` ... síťová komunikace, adresa je dvojice (IP adresa, port)
- `type`:
  - `SOCK_STREAM` ... spojovaná spolehlivá služba, poskytuje obousměrný sekvenční proud dat
  - `SOCK_DGRAM` ... nespojovaná nespolehlivá služba, přenos datagramů
- `protocol`: 0 (default pro daný `type`) nebo platné číslo protokolu (např. 6 = TCP, 17 = UDP)

## Pojmenování socketu: bind()

```
int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);
```

- přiřadí socketu zadanému deskriptorem `socket` adresu
- obecná `struct sockaddr`, nepoužívá se pro vlastní zadávání adres:
  - `sa_family_t sa_family` ... doména
  - `char sa_data[]` ... adresa
- pro `AF_INET` se používá `struct sockaddr_in`:
  - `sa_family_t sin_family` ... doména (`AF_INET`)
  - `in_port_t sin_port` ... číslo portu (16 bitů)
  - `struct in_addr sin_addr` ... IP adresa (32 bitů)
  - `unsigned char sin_zero[8]` ... výplň (*padding*)



## Struktura pro IPv4 adresy

- každá adresní rodina má svoji strukturu a svůj hlavičkový soubor
- použitou strukturu pak ve volání `socket` přetypujete na `struct sockaddr`

```
#include <netinet/in.h>
struct sockaddr_in in; /* IPv4 */

bzero(&in, sizeof (in));
in.sin_family = AF_INET;
in.sin_port = htons(2222);
in.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(s, (struct sockaddr *)&in, sizeof (in)) == -1) ..
```

# Čekání na spojení: `listen()`

```
int listen(int socket, int backlog);
```

- označí soket zadaný deskriptorem `socket` jako akceptující spojení a systém na soketu začne poslouchat.
- maximálně `backlog` žádostí o spojení může najednou čekat ve frontě na obsloužení (implementace může hodnotu `backlog` změnit, pokud není v podporovaném rozsahu). Žádosti, které se nevejdou do fronty, jsou odmítnuty (tj. volání `connect` skončí s chybou).
- soket čeká na spojení na adrese, která mu byla dříve přiřazena voláním `bind`.

## Akceptování spojení: `accept()`

```
int accept(int socket, struct sockaddr *address,  
          socklen_t *address_len);
```

- vytvoří spojení mezi lokálním soketem `socket` (který dříve zavolal `listen`) a vzdáleným soketem, žádajícím o spojení pomocí `connect`. Vrátí deskriptor (nový soket), který lze používat pro komunikaci se vzdáleným procesem. Původní soket může hned přijímat další spojení pomocí `accept`.
- v `address` vrátí adresu vzdáleného soketu.
- `address_len` je velikost struktury pro uložení adresy v bajtech, po návratu obsahuje skutečnou délku adresy.

## Práce s IPv4 a IPv6 adresami

- binární reprezentace adresy se nám špatně čte
- a reprezentaci adresy řetězcem nelze použít při práci se `sockaddr` strukturami

```
int inet_pton(int af, const char *src, void *dst);
```

- převede řetězec na binární adresu, tj. to co je možné použít v `in_addr` nebo `in6_addr` položkách `sockaddr` struktur
- vrátí 1 (OK), 0 (chybně zadaná adresa) nebo -1 (a nastaví `errno`)

```
const char *inet_ntop(int af, const void *src,  
                      char *dst, size_t size);
```

- opak k `inet_pton`; vrátí `dst` nebo `NULL` (a nastaví `errno`)
- pro obě volání platí, že `af` je `AF_INET` nebo `AF_INET6`

## Navázání spojení: connect()

```
int connect(int sock, struct sockaddr *address,  
            socklen_t address_len);
```

- naváže spojení lokálního soketu `sock` se vzdáleným procesem, který pomocí `listen` a `accept` čeká na spojení na adrese `address` (o délce `address_len`).
- jestliže pro soket `sock` nebyla definována adresa voláním `bind`, je mu přiřazena nějaká nepoužitá adresa dle zvolené rodiny protokolů.
- pokud se spojení nepovede, je soket v nedefinovaném stavu. Před novým pokusem o spojení by aplikace měla zavřít deskriptor `sock` a vytvořit nový soket.

## Přijetí zprávy: `recvfrom()`

```
ssize_t recvfrom(int sock, void *buf, size_t len,  
                int flg, struct sockaddr *address,  
                socklen_t *address_len);
```

- přijme zprávu ze soketu `sock`, uloží ji do bufferu `buf` o velikosti `len`, do `address` dá adresu odesílatele zprávy, do `address_len` délku adresy. Vrátí délku zprávy. Když je zpráva delší než `len`, nadbytečná data se zahodí (`SOCK_STREAM` nedělí data na zprávy, data se nezahazují).
- ve `flg` mohou být příznaky:
  - `MSG_PEEK` ... zpráva se bere jako nepřečtená, další `recvfrom` ji vrátí znovu
  - `MSG_OOB` ... přečte urgentní (out-of-band) data
  - `MSG_WAITALL` ... čeká, dokud není načten plný objem dat, tj. `len` bajtů

## Poslání zprávy: `sendto()`

```
ssize_t sendto(int socket, void *msg, size_t len,  
               int flags, struct sockaddr *addr,  
               socklen_t addr_len);
```

- prostřednictvím soketu `socket` pošle zprávu `msg` o délce `len` na adresu `addr` (o délce `addr_len`).
- parametr `flags` může obsahovat příznaky:
  - `MSG_EOB` ... ukončení záznamu (pokud je podporováno protokolem)
  - `MSG_OOB` ... poslání urgentních (out-of-band) dat, jejichž význam je závislý na protokolu

## Další funkce pro sokety

```
int setsockopt(int socket, int level, int opt_name,  
               const void *opt_value, socklen_t option_len);
```

- nastavení parametrů soketu

```
int getsockopt(int socket, int level, int opt_name,  
              void *opt_value, socklen_t *option_len);
```

- přečtení parametrů soketu

```
int getsockname(int socket, struct sockaddr *address,  
                socklen_t *address_len);
```

- zjištění (lokální) adresy soketu

```
int getpeername(int socket, struct sockaddr *address,  
                socklen_t *address_len);
```

- zjištění adresy vzdáleného soketu (druhého konce spojení)



# Uzavření socketu: close()

```
int close(int fildes);
```

- zruší deskriptor, při zrušení posledního deskriptoru socketu zavře socket.
- pro `SOCK_STREAM` socket záleží na nastavení příznaku `SO_LINGER` (default je `l_onoff == 0`, mění se funkcí `setsockopt`).
  - `l_onoff == 0` ... volání `close` se vrátí, ale jádro se snaží dál přenést zbylá data
  - `l_onoff == 1 && l_linger != 0` ... jádro se snaží přenést zbylá data do vypršení timeoutu `l_linger` v sekundách, když se to nepovede, `close` vrátí chybu, jinak vrátí OK po přenesení dat.
  - `l_onoff == 1 && l_linger == 0` ... provede se reset spojení

## Uzavření soketu: shutdown()

```
int shutdown(int socket, int how);
```

- Uzavře soket (ale neruší deskriptor) podle hodnoty *how*:
  - SHUT\_RD ... pro čtení
  - SHUT\_WR ... pro zápis
  - SHUT\_RDWR ... pro čtení i zápis

# Pořadí bajtů

- síťové služby používají pořadí bajtů, které se může lišit od pořadí používaného na lokálním systému. Pro převod lze použít funkce (makra):
  - `uint32_t htonl(uint32_t hostlong);`  
host → síť, 32 bitů
  - `uint16_t htons(uint16_t hostshort);`  
host → síť, 16 bitů
  - `uint32_t ntohl(uint32_t netlong);`  
síť → host, 32 bitů
  - `uint16_t ntohs(uint16_t netshort);`  
síť → host, 16 bitů
- síťové pořadí bajtů je big-endian, tj. nejprve vyšší bajt. Používá se hlavně ve funkcích pracujících s adresami a čísly portů.

# Číslo protokolů a portů

```
struct protoent *getprotobyname(const char *name);
```

- v položce `p_proto` vrátí číslo protokolu se jménem `name` (např. pro "tcp" vrátí 6).
- čísla protokolů jsou uložena v souboru `/etc/protocols`.

```
struct servent *getservbyname(const char *name,  
                              const char *proto);
```

- pro zadané jméno služby `name` a jméno protokolu `proto` vrátí v položce `s_port` číslo portu.
- čísla portů jsou uložena v souboru `/etc/services`.

funkce vrací NULL, když v databázi není odpovídající položka.

## Převod hostname na adresy: getaddrinfo()

- ze zadaných parametrů přímo generuje sockaddr struktury
- pracuje s adresami i s porty
- její chování je ovlivnitelné pomocí tzv. *hintů*

```
int getaddrinfo(const char *nodename,  
               const char *servicename,  
               const struct addrinfo *hint,  
               struct addrinfo **res);
```

- položky struktury addrinfo: ai\_flags (pro hinty), ai\_family (address family), ai\_socktype, ai\_protocol, ai\_addrlen, struct sockaddr \*ai\_addr (výsledné adresy), char \*ai\_canonname, struct addrinfo \*ai\_next (další prvek v seznamu)

## Převod adresy na hostname: getnameinfo()

- protějšek k funkci getaddrinfo
- jako vstup pracuje s celými sockaddr strukturami, tedy funguje s IPv4 i IPv6 adresami, narozdíl od funkce gethostbyaddr.

```
int getnameinfo(const struct sockaddr *sa,  
               socklen_t *sa_len,  
               char *nodename,  
               socketlen_t *nodelen,  
               char *servicename,  
               socketlen_t *servicelen,  
               unsigned flags);
```

## Příklad: TCP server

```
struct sockaddr_storage ca; int nclients = 10, fd, nfd;
struct addrinfo *r, *rorig, hi;
memset(&hi, 0, sizeof (hi)); hi.ai_family = AF_UNSPEC;
hi.ai_socktype = SOCK_STREAM; hi.ai_flags = AI_PASSIVE;
getaddrinfo(NULL, portstr, &hi, &rorig);
for (r = rorig; r != NULL; r = r->ai_next) {
    fd = socket(r->ai_family, r->ai_socktype,
               r->ai_protocol);
    if (!bind(fd, r->ai_addr, r->ai_addrlen)) break;
}
freeaddrinfo(rorig); listen(fd, nclients);
for (;;) { sz = sizeof(ca);
    nfd = accept(fd, (struct sockaddr *)&ca, &sz);
    /* Komunikace s klientem */
    close(newsock);
}
```

## Příklad: TCP klient

```
int fd; struct addrinfo *r, *rorig, hi;
memset(&hi, 0, sizeof (hi)); hi.ai_family = AF_UNSPEC;
hi.ai_socktype = SOCK_STREAM;
getaddrinfo(hoststr, portstr, &hi, &r);
for (rorig = r; r != NULL; r = r->ai_next) {
    fd = socket(r->ai_family, r->ai_socktype,
                r->ai_protocol);
    if (connect(fd, (struct sockaddr *)r->ai_addr,
                r->ai_addrlen) == 0)
        break;
}
freeaddrinfo(resorig);
/* Komunikace se serverem */
close(fd);
```



## Čekání na data: `select()`

```
int select(int nfds, fd_set *readfds,  
          fd_set *writefds, fd_set *errorfds,  
          struct timeval *timeout);
```

- zjistí, které ze zadaných deskriptorů jsou připraveny pro čtení, zápis, nebo na kterých došlo k výjimečnému stavu. Pokud žádný takový deskriptor není, čeká do vypršení času `timeout` (NULL ...čeká se libovolně dlouho). Parametr `nfds` udává rozsah testovaných deskriptorů (0, ..., `nfds-1`).
- pro nastavení a test masek deskriptorů slouží funkce:
  - void **FD\_ZERO**(fd\_set \**fdset*) ... inicializace
  - void **FD\_SET**(int *fd*, fd\_set \**fdset*) ...nastavení
  - void **FD\_CLR**(int *fd*, fd\_set \**fdset*) ... zrušení
  - int **FD\_ISSET**(int *fd*, fd\_set \**fdset*) ... test

## Příklad: použití `select()`

- Deskriptor `fd` odkazuje na soket, přepisuje síťovou komunikaci na terminál a naopak.

```
int sz; fd_set rfdset, efdset; char buf[BUFSZ];
for(;;) {
    FD_ZERO(&rfdset); FD_SET(0, &rfdset);
    FD_SET(fd, &rfdset); efdset = rfdset;
    select(fd+1, &rfdset, NULL, &efdset, NULL);
    if(FD_ISSET(0, &efdset))
        /* Výjimka na stdin */ ;
    if(FD_ISSET(fd, &efdset))
        /* Výjimka na fd */ ;
    if(FD_ISSET(0, &rfdset)) {
        sz = read(0, buf, BUFSZ); write(fd, buf, sz); }
    if(FD_ISSET(fd, &rfdset)) {
        sz = read(fd, buf, BUFSZ); write(1, buf, sz); }
}
```

## Čekání na data: poll()

```
int poll(struct pollfd fds [], nfdst nfds, int timeout);
```

- čeká na událost na některém z deskriptorů v poli *fds* o *nfds* prvcích po dobu *timeout* ms (0 ... vrátí se hned, -1 ... čeká se libovolně dlouho).
- prvky struktury *pollfd*:
  - *fd* ... číslo deskriptoru
  - *events* ... očekávané události, OR-kombinace POLLIN (lze číst), POLLOUT (lze psát), atd.
  - *revents* ... události, které nastaly, příznaky jako v *events*, navíc např. POLLERR (nastala chyba)

# Správa síťových služeb: inetd

- servery síťových služeb se spouští buď při startu systému, nebo je startuje démon `inetd` při připojení klienta.
- démon `inetd` čeká na portech definovaných v `/etc/inetd.conf` a když detekuje příchozí spojení/datagram, spustí příslušný server a přesměruje mu deskriptory.
- příklad obsahu `/etc/inetd.conf`:

```
ftp stream tcp nowait root /usr/etc/ftpd ftpd -l
shell stream tcp nowait root /usr/etc/rshd rshd -L
login stream tcp nowait root /usr/etc/rlogind rlogind
exec stream tcp nowait root /usr/etc/rexecd rexecd
finger stream tcp nowait guest /usr/etc/fingerd fingerd
ntalk dgram udp wait root /usr/etc/talkd talkd
tcpmux stream tcp nowait root internal
echo stream tcp nowait root internal
```

# Formát souboru `/etc/inetd.conf`

`služba socket proto čekání uživ server argumenty`

- `služba ...` jméno síťové služby podle `/etc/services`
- `socket ...` `stream` nebo `dgram`
- `proto ...` komunikační protokol (`tcp`, `udp`)
- `čekání ...` `wait` (`inetd` čeká na ukončení serveru před akceptováním dalšího klienta), `nowait` (`inetd` akceptuje dalšího klienta hned)
- `uživatel ...` server poběží s identitou tohoto uživatele
- `server ...` úplná cesta k programu serveru nebo `internal` (službu zajišťuje `inetd`)
- `argumenty ...` příkazový řádek pro server, včetně `argv[0]`

# Obsah

- úvod, vývoj UNIXu a C, programátorské nástroje
- základní pojmy a konvence UNIXu a jeho API
- přístupová práva, periferní zařízení, systém souborů
- manipulace s procesy, spouštění programů
- signály
- synchronizace a komunikace procesů
- síťová komunikace
- **vlákna, synchronizace vláken**
- ??? - bude definováno později, podle toho kolik zbyde času

# Vlákna

- vlákno (*thread*) = linie výpočtu (*thread of execution*)
- vlákna umožňují mít více linií výpočtu v rámci jednoho procesu
- klasický unixový model: jednovláknové procesy
- vlákna nejsou vhodná pro všechny aplikace
- výhody vláken:
  - zrychlení aplikace, typicky na víceprocesorech (vlákna jednoho procesu mohou běžet současně na různých procesorech)
  - modulární programování
- nevýhody vláken:
  - není jednoduché korektně napsat složitější kód s vlákny
  - obtížnější debugging

# Implementace vláken

## library-thread model

- vlákna jsou implementována v knihovnách, jádro je nevidí.
- run-time knihovna plánuje vlákna na procesy a jádro plánuje procesy na procesory.
- ⊕ menší režie přepínání kontextu
- ⊖ nemůže běžet více vláken stejného procesu najednou.

## kernel-thread model

- vlákna jsou implementována přímo jádrem.
- ⊕ více vláken jednoho procesu může běžet najednou na různých procesorech.
- ⊖ plánování threadů používá systémová volání místo knihovnických funkcí, tím více zatěžuje systém.

## hybridní modely

- vlákna se multiplexují na několik jádrem plánovaných entit.



# POSIX vlákna (pthreads)

- definované rozšířením POSIX.1c
- volání týkající se vláken začínají prefixem `pthread_`
- tyto funkce vrací 0 (OK) nebo přímo číslo chyby
  - ... a nenastavují `errno`!
  - nelze s nimi proto použít funkce `perror` nebo `err`
- POSIX.1c definuje i další funkce, například nové verze k těm, které nebylo možné upravit pro použití ve vláknech bez změny API, např `readdir_r`, `strtok_r`, atd.
  - `_r` znamená *reentrant*, tedy že funkci může volat více vláken najednou bez vedlejších efektů

# Vytvoření vlákna

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

- vytvoří nové vlákno, do `thread` uloží jeho ID.
- podle `attr` nastaví jeho atributy, např. velikost zásobníku či plánovací politiku. `NULL` znamená použít implicitní atributy.
- ve vytvořeném vláknu spustí funkci `start_fun()` s argumentem `arg`. Po návratu z této funkce se zruší vlákno.
- s objekty `pthread_attr_t` lze manipulovat funkcemi `pthread_attr_init()`, `pthread_attr_destroy()`, `pthread_attr_setstackaddr()`, atd...

# Soukromé atributy vláken

- čítač instrukcí
- zásobník (automatické proměnné)
- thread ID, dostupné funkcí  
`pthread_t pthread_self(void);`
- plánovací priorita a politika
- hodnota `errno`
- klíčované hodnoty – dvojice (`pthread_key_t key`, `void *ptr`)
  - klíč vytvořený voláním `pthread_key_create()` je viditelný ve všech vláknech procesu.
  - v každém vláknu může být s klíčem asociována jiná hodnota voláním `pthread_setspecific()`.

# Ukončení vlákna

```
void pthread_exit(void *val_ptr);
```

- ukončí volající vlákno, je to obdoba `exit` pro proces

```
int pthread_join(pthread_t thr, void **val_ptr);
```

- počká na ukončení vlákna `thr` a ve `val_ptr` vrátí hodnotu ukazatele z `pthread_exit` nebo návratovou hodnotu vláknové funkce. Pokud vlákno skončilo dříve, funkce hned vrací příslušně nastavené `val_ptr`.
- obdoba čekání na synovský proces pomocí `wait`

```
int pthread_detach(pthread_t thr);
```

- nastaví okamžité uvolnění paměti po ukončení vlákna, na vlákno nelze použít `pthread_join`.

# Inicializace

```
int pthread_once(pthread_once_t *once_control,  
                void (*init_routine)(void));
```

- V parametru `once_control` se předává ukazatel na staticky inicializovanou proměnnou

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

- První vlákno, které zavolá `pthread_once()`, provede inicializační funkci `init_routine()`. Ostatní vlákna už tuto funkci neprovádějí, navíc, pokud inicializační funkce ještě neskončila, zablokují se a čekají na její dokončení.
- Lze použít např. na dynamickou inicializaci globálních dat v knihovnách, jejichž kód může zavolat více vláken současně, ale je třeba zajistit, že inicializace proběhne jen jednou.

# Zrušení vlákna

```
int pthread_cancel(pthread_t thread);
```

- požádá o zrušení vlákna `thread`. Závisí na nastavení

```
int pthread_setcancelstate(int state, int *old);
```

- nastaví nový stav a vrátí starý:
  - `PTHREAD_CANCEL_ENABLE` ... povoleno zrušit
  - `PTHREAD_CANCEL_DISABLE` ... nelze zrušit, žádost bude čekat na povolení

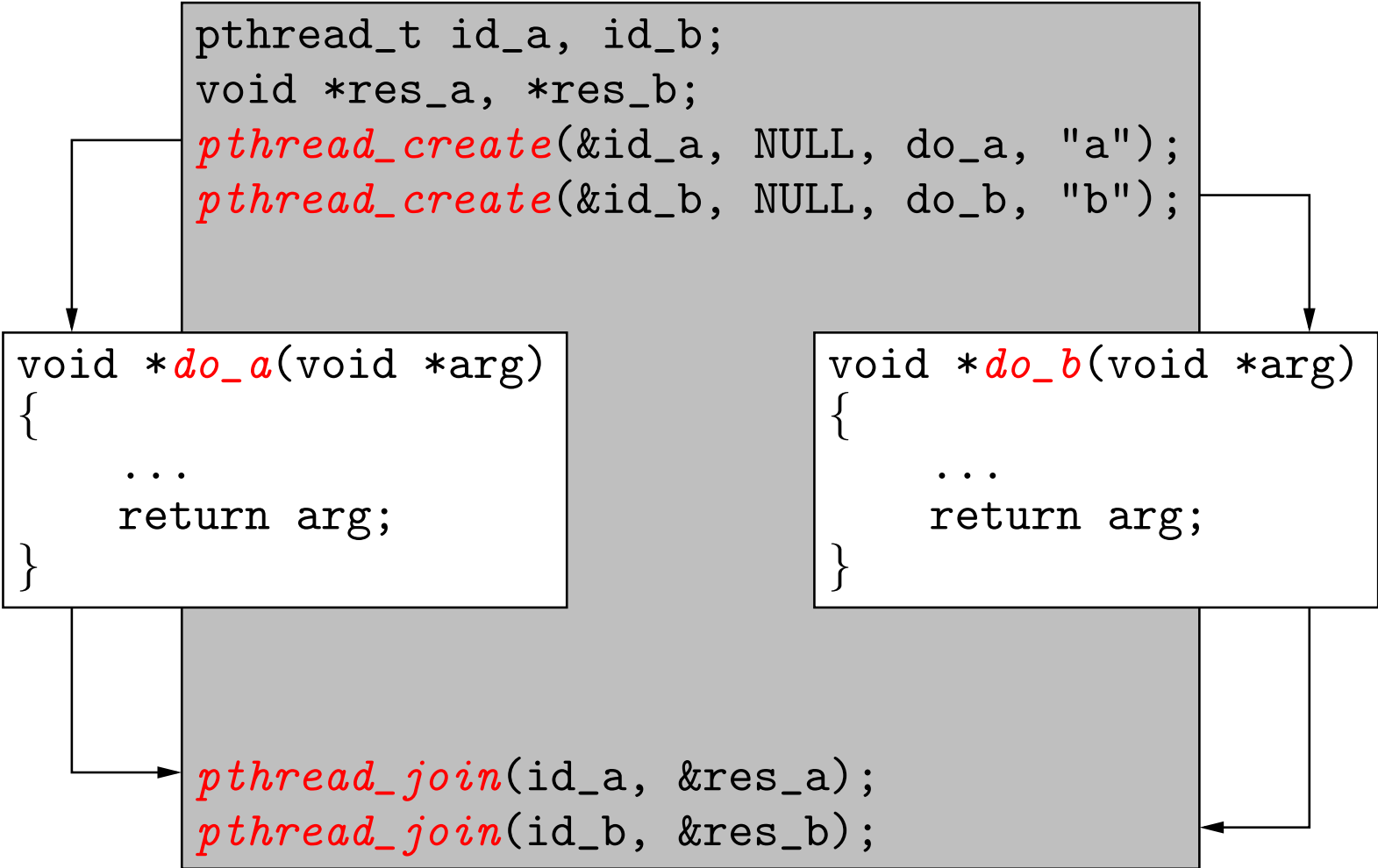
```
int pthread_setcanceltype(int type, int *old);
```

- `PTHREAD_CANCEL_ASYNC` ... okamžité zrušení
- `PTHREAD_CANCEL_DEFERRED` ... žádost čeká na vstup do určitých funkcí (např. `open()`, `read()`, `wait()`), nebo na volání

```
void pthread_testcancel(void);
```

# Příklad: vlákna

```
pthread_t id_a, id_b;  
void *res_a, *res_b;  
pthread_create(&id_a, NULL, do_a, "a");  
pthread_create(&id_b, NULL, do_b, "b");
```



```
void *do_a(void *arg)  
{  
    ...  
    return arg;  
}
```

```
void *do_b(void *arg)  
{  
    ...  
    return arg;  
}
```

```
pthread_join(id_a, &res_a);  
pthread_join(id_b, &res_b);
```

# Globální proměnné pro vlákno

```
int pthread_key_create(pthread_key_t *key,  
                      void (*destructor)(void *));
```

- vytvoří klíč, který lze asociovat s hodnotou typu (void \*). Funkce `destructor()` se volají opakovaně pro všechny klíče, jejichž hodnota není NULL, při ukončení vlákna.

```
int pthread_key_delete(pthread_key_t key);
```

- zruší klíč, nemění asociovaná data.

```
int pthread_setspecific(pthread_key_t key,  
                       const void *value);
```

- přiřadí ukazatel `value` ke klíči `key`.

```
void *pthread_getspecific(pthread_key_t key);
```

- vrátí hodnotu ukazatele příslušného ke klíči `key`.



## Úklid při ukončení/zrušení vlákna

- vlákno má zásobník úklidových handlerů, které se volají při ukončení nebo zrušení vlákna funkcemi `pthread_exit` a `pthread_cancel` (ale ne při `return`). Handlerly se spouští v opačném pořadí než byly vkládány do zásobníku.
- po provedení handlerů se volají destruktory privátních klíčovaných dat vlákna (pořadí není specifikované)

```
void pthread_cleanup_push(void (*routine)(void *),  
                          void *arg);
```

- vloží handler do zásobníku.

```
void pthread_cleanup_pop(int execute);
```

- vyjme naposledy vložený handler ze zásobníku. Provede ho, pokud je `execute` nenulové.

## fork() a vlákna (POSIX)

- je nutné definovat sémantiku volání `fork` v aplikacích používajících vlákna. Norma definuje, že:
  - nový proces obsahuje přesnou kopii volajícího vlákna, včetně případných stavů mutexů a jiných zdrojů
  - ostatní vlákna v synovském procesu neexistují
  - pokud taková vlákna měla naalokovanou paměť, zůstane tato paměť naalokovaná (= ztracená)
  - obdobně to platí pro zamčený mutex již neexistujícího vlákna
- vytvoření nového procesu z multivláknové aplikace má smysl pro následné volání `exec` (tj. včetně volání `popen`, `system` apod.)

# Signály a vlákna

- signály mohou být generovány pro proces (voláním `kill`), nebo pro vlákno (chybové události, volání `pthread_kill`).
- nastavení obsluhy signálů je stejné pro všechna vlákna procesu, ale masku blokováných signálů má každé vlákno vlastní, nastavuje se funkcí

```
int pthread_sigmask(int how, const sigset_t *set,  
                    sigset_t *oset);
```

- signál určený pro proces ošetří vždy právě jedno vlákno, které nemá tento signál zablokovaný.
- lze vyhradit jedno vlákno pro synchronní příjem signálů pomocí volání `sigwait`. Ve všech vláknech se signály zablokují.

# Synchronizace vláken obecně

- většina programů používajících vlákna mezi nimi sdílí data
- nebo vyžaduje, aby různé akce byly prováděny v jistém pořadí
- ...toto vše potřebuje **synchronizovat** aktivitu běžících vláken
- jak bylo zmíněno u implementace vláken, u procesů je pro sdílení dat nutné vynaložit jisté úsilí, u vláken naopak musíme investovat do toho, abychom přirozené sdílení dat uhlídali
- my se vzhledem k vláknům budeme zabývat:
  - mutexy
  - podmínkovými proměnnými
  - read-write zámky

# Synchronizace vláken: mutexy (1)

- nejjednodušší způsob zajištění synchronizovaného přístupu ke sdíleným datům mezi vlákny je použitím mutexu
- inicializace staticky definovaného mutexu:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- inicializace dynamicky alokovaného mutexu *mx* s atributy *attr* (nastavují se pomocí `pthread_mutexattr...`; a je-li místo *attr* použit NULL, vezmou se defaultní atributy)

```
int pthread_mutex_init(pthread_mutex_t *mx,  
                        const pthread_mutexattr_t *attr);
```

- po skončení používání mutexu je možné ho zrušit:

```
int pthread_mutex_destroy(pthread_mutex_t *mx);
```

## Mutexy (2)

- pro zamčení a odemčení mutexu použijeme volání:

```
int pthread_mutex_lock(pthread_mutex_t *m);
```

a

```
int pthread_mutex_unlock(pthread_mutex_t *m);
```

- pokud je mutex již zamčený, pokus o zamknutí vyústí v zablokování vlákna. Je možné použít i volání:

```
int pthread_mutex_trylock(pthread_mutex_t *m);
```

... které se pokusí zamknout mutex, a pokud to nelze provést, skončí s chybou

# Podmínkové proměnné (1)

- mutexy slouží pro synchronizaci přístupu ke sdíleným datům
- podmínkové proměnné pak k předání informací o těchto sdílených datech – například že se hodnota dat změnila
- ... a umožní tak vlákna podle potřeby uspávat a probouzet
- z toho plyne, že **každá podmínková proměnná je vždy asociována s právě jedním mutexem**
- jeden mutex ale může být asociován s více podmínkovými proměnnými
- společně pomocí mutexů a podmínkových proměnných je možné vytvářet další synchronizační primitiva – semaforey, bariéry, ...

## Podmínkové proměnné (2)

```
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t *attr);
```

- Inicializuje podmínkovou proměnnou `cond` s atributy `attr` (nastavují je funkce `pthread_condattr_...()`), `NULL` = default.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- zruší podmínkovou proměnnou.

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);
```

- čeká na podmínkové proměnné dokud jiné vlákno nezavolá `pthread_cond_signal()` nebo `pthread_cond_broadcast()`.



## Podmínkové proměnné (3)

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- probudí jeden proces čekající na podmínkové proměnné `cond`.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- probudí všechny procesy čekající na podmínkové proměnné `cond`.

```
int pthread_cond_timedwait(pthread_cond_t *cond,  
                           pthread_mutex_t *mutex,  
                           const struct timespec *atime);
```

- čeká na `pthread_cond_signal()` nebo `pthread_cond_broadcast()`, ale maximálně do doby než systémový čas dosáhne hodnoty dané `atime`.

# Použití podmínkových proměnných

```
pthread_cond_t cond; pthread_mutex_t mutex;
```

```
...
```

```
pthread_mutex_lock(&mutex);
```

```
while (!podminka(data))
```

```
    pthread_cond_wait(&cond, &mutex);
```

```
process_data(data, ...);
```

```
pthread_mutex_unlock(&mutex);
```

```
...
```

```
pthread_mutex_lock(&mutex);
```

```
produce_data(data, ...);
```

```
pthread_cond_signal(&cond);
```

```
pthread_mutex_unlock(&mutex);
```

# Read-write zámky (1)

```
int pthread_rwlock_init(pthread_rwlock_t *l,  
                        const pthread_rwlockattr_t *attr);
```

- vytvoří zámeček s atributy podle `attr` (nastavují se funkcemi `pthread_rwlockattr_...()`, `NULL` = default)

```
int pthread_rwlock_destroy(pthread_rwlock_t *l);
```

- zruší zámeček

```
int pthread_rwlock_rdlock(pthread_rwlock_t *l);
```

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

- zamkne zámeček pro čtení (více vláken může držet zámeček pro čtení), pokud má někdo zámeček pro zápis, uspí volající vlákno (`rdlock()`) resp. vrátí chybu (`tryrdlock()`).

## Read-write zámky (2)

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

- zamkne zámeček pro zápis; pokud má někdo zámeček pro čtení nebo zápis, čeká.

```
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

- jako `pthread_rwlock_wrlock()`, ale když nemůže zamknout, vrátí chybu.

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

- odemkne zámeček

# Atomické aritmetické operace

- pro architektury, kde operace sčítání není atomická
- výrazně rychlejší než jiné mechanismy pro získání exkluzivního přístupu díky použití instrukcí na dané platformě zajišťujících atomicitu.
- některé systémy dodávají funkce pro atomické operace, (např. `atomic_add(3c)` v Solarisu), obecně je lepší použít podporu v C11 standardu přes `stdatomic.h`.
- sada volání pro různé typy a operace, např. sčítání:

```
#include <stdatomic.h>
```

```
atomic_int acnt;
```

```
atomic_fetch_add(&acnt, 1);
```

# Bariéra

- bariéra (*barrier*) je způsob, jak udržet členy skupiny pohromadě
- všechna vlákna čekají na bariéře, dokud ji nedosáhne poslední vlákno; pak mohou pokračovat
- typické použití je paralelní zpracování dat

```
int pthread_barrier_init(pthread_barrier_t *barrier, attr,  
unsigned count);
```

- inicializuje bariéru pro *count* vstupů do ní

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- zablokuje se dokud není zavolána *count*-krát

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- zruší bariéru

# Semaforey

- semaforey pochází z POSIX-1003.1b (real-time extensions)
- jména funkcí nezačínají **pthread\_**, ale **sem\_** (`sem_init`, `sem_post`, `sem_wait`, ...)
- je možné je použít s vlákny

# Typické použití vláken

- **pipeline**

- každé z vláken provádí svoji operaci nad daty, která se postupně předávají mezi vlákny
- každé vlákno typicky provádí jinou operaci
- ... zpracování obrázku, kde každé vlákno provede jiný filtr

- **work crew**

- vlákna provádějí stejnou operaci, ale nad jinými daty
- ... zpracování obrázku dekompozicí – každé vlákno zpracovává jinou část obrázku, výsledkem je spojení zpracovaných dat ze všech vláken; zde se hodí řešení s bariérou

- **client – server**



# Thread-safe versus reentrantní

- *thead-safe* znamená, že kód může být volán z více vláken najednou bez destruktivních následků
  - do funkce, která nebyla navržena jako thread-safe, je možné přidat jeden zámek – na začátku funkce se zamkne, na konci odemkne
  - tento způsob ale samozřejmě není příliš efektivní
- slovem *reentrantní* se typicky myslí, že daná funkce byla navržena s přihlédnutím na existenci vláken
  - ... tedy že funkce pracuje efektivně i ve vícevláknovém prostředí
  - taková funkce by se měla vyvarovat použití statických dat a pokud možno i prostředků pro synchronizaci vláken, protože běh aplikace zpomalují

# Nepřenositelná volání

- nepřenositelná volání končí řetězcem `_np` (*non-portable*) a jednotlivé systémy si takto definují vlastní volání
- FreeBSD
  - `pthread_set_name_np(pthread_t tid, const char *name)`
  - umožňuje pojmenovat vlákno
- Solaris
  - `pthread_cond_reltimedwait_np(...)`
  - jako `timedwait`, ale časový `timeout` je relativní
- OpenBSD
  - `int pthread_main_np(void)`
  - umožňuje zjistit, zda volající vlákno je hlavní (= `main()`)

**The End.**