UNIX

Programování a správa systému I

Jan Kasprzak (kas@fi.muni.cz)

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

Úvodní informace

Předpoklady

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

Cíle kursu

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

Ukončení předmětu

- **Test** – 20 otázek.
- **Hodnocení**: –1 až 4 body na otázku, na kolokvium je potřeba 40 bodů a více.

Obsah přednášky

- **Základy programování pod UNIXem** – nástroje.
- **Normy API pro jazyk C pro UN*X**
- **Program podle ANSI C** – limity, start a ukončení programu, argumenty, proměnné prostředí, práce s pamětí, vzdálené skoky. Hlavičkové soubory a knihovny. Sdílené knihovny.
- **Jádro** – start jádra, architektura jádra, paměťový model, komunikace s jádrem, knihovna versus systémové volání.
- **Proces** – paměťový model, vznik a zánik procesu, program na disku.
- **Vstupní/výstupní operace** – deskriptor, operace s deskriptory.
- **Soubory a adresáře** – i-uzel, operace s ním. Architektura souborového systému.
- **Komunikace mezi procesy** – roura, signály.
- **Pokročilé V/V operace** – zamykání souborů, scatter-gather I/O, soubory mapované do paměti, multiplexování vstupů a výstupů.

Materiály ke studiu

- **Slidy z přednášek:** `http://www.fi.muni.cz/~kas/p065/`

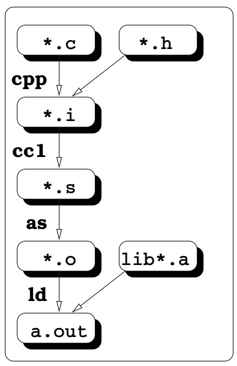
- **Maurice J. Bach:** *Principy Operačního Systému UNIX*, Softwarové Aplikace a Systémy, 1993, ISBN 80–901507–0–5
- **Uresh Vahalia:** *UNIX Internals – the New Frontiers*, Prentice Hall 1996, ISBN 0–13–101908–2
- **J. Brodský, L. Skočovský:** *Operační systém UNIX a jazyk C*, SNTL 1989, ISBN 80–03–00049–1
- **W. Richard Stevens:** *Advanced Programming in the UNIX Environment*, Addison-Wesley 1992, ISBN 0–201–56317–7
- **IEEE Std. 1003.1:** *Information Technology – Portable Operating System Interface (POSIX) – Part 1: System API (C language)*, IEEE 1996, ISBN 1–55937–573–6
- **Andrew Josey:** *The UNIX System Today – The Authorized Guide to Version 2 of the Single UNIX Specification*, The Open Group 2000, ISBN 1–85912–296–5, `www.opennc.org/onlinepubs/7908799/`

Vývojové prostředí

Rychlý start

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


Kompilace C-programu



Kompilátor cc

- **Vývojové prostředí** – obvykle za příplatek.
- **GNU C/C++**
- **Spouští další programy** – `cpp(1)`, `comp/cc1`, `as(1)`, `ld(1)`.
- Lze spouštět i jen jednotlivé části překladu.

- **Start kompilace** se řídí příponou souboru.

- **Konec kompilace:**

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

- **Parametry preprocesoru:**

-D⟨*makro*⟩
-D⟨*makro*⟩=⟨*hodnota*⟩
– nadefinuje makro pro preprocesor.
-U⟨*makro*⟩ – ruší definici makra.
-I⟨*adresář*⟩ – adresář pro hlavičkové soubory.
-I- – vypíná standardní adresáře (`/usr/include`).

- **Parametry kompilátoru:**

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

- **Parametry linkeru:**

-L⟨*adresář*⟩ – adresář pro knihovny.
-nostdlib – bez standardních knihoven.
-l⟨*knihovna*⟩ – přidá soubor `lib`⟨*knihovna*⟩`.a`, případně `.so`.
-static – statické linkování.
-shared – sdílené knihovny.
-s – odstranit tabulku symbolů.

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

Opakování – jazyk C

Program v paměti

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

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

Zde `&funkce` je adresa do textu programu, `&znak` do datové části a `&cislo` na zásobník.

◇ **Úkol:**Kam padne adresa `&argument`?

Umístění proměnných v paměti

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

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

Na zásobníku je uložena proměnná `ptakopysk`. Ostatní dvě jsou v datové části.

Viditelnost proměnných

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

```
/* data.c */
int odpoved;
static char *otazka;

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

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

Linkování selže. Po zrušení klíčového slova `static` v `data.c` projde.

Program make

- **Řízená kompilace z více modulů**
- **Soubor Makefile**

-f ⟨*Makefile*⟩ – soubor místo `makefile` nebo `Makefile`.
-i – ignoruj chyby.
-n – vypiš příkazy, ale neprováděj.
-s – nevypisuj prováděné příkazy.

Proměnné

⟨*proměnná*⟩=⟨*hodnota*⟩
```
CC=gcc -g
CFLAGS=$(OPT_FLAGS) $(DEBUG_FLAGS)
```

Závislosti

⟨*cíl*⟩: ⟨*zdroj*...⟩
```
program.o: program.c program.h
```

Akce

⟨*tabulátor*⟩⟨*příkaz*⟩
```
     $(CC) -c program.c
```

Příklad Makefile

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

- **Implicitní pravidla** – převod souborů podle přípon (GNU make umí i podle obecnějších pravidel).

Další programy

nm(1) . . . . Výpis tabulky symbolů

```
$ nm ⟨program⟩
$ nm richie.o
00000000 t gcc2_compiled.
00000000 T main
         U printf
```

strip(1) . . . . Odstranění tabulky symbolů

```
$ strip ⟨executable⟩
```

size(1) . . . . Velikost objektového souboru

```
$ size ⟨objfile⟩
$ size x.o
text data bss    dec    hex    filename
20   3    0      23     17     x.o
```

ar(1) . . . . Archivace programů

Program `ar(1)` se používá při vytváření statických knihoven (knihovna je v podstatě arovský archív s tabulkou symbolů – `ranlib(1)`).

ranlib(1) . . . . Index archívu

Program vytvoří index všech symbolů ve všech objektových souborech daného archívu. Tento index přidá do archívu.

V některých systémech – totéž co `ar -s`.

objdump(1) . . . . Vypisuje obsah objektového souboru

- Program vypisuje různé informace z objektového souboru (`*.o`).
- Může sloužit i jako disassembler.

Knihovny

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

Statické knihovny

- **Statická knihovna** – stane se součástí spustitelného souboru.
- **Formát** – archív programu `ar(1)`.
- **Tabulka symbolů** – pro urychlení linkování – vytvářená pomocí `ranlib(1)`.
- Některé systémy vyžadují spuštění `ranlib` při vytváření knihovny.
- **GNU ar** – umí generovat tabulku symbolů sám.
- **Staticky linkovaný program** – je větší, neumí sdílet kód s jinými programy, ale je v podstatě nezávislý.

Sdílené knihovny

- **Dynamicky linkované knihovny/moduly** – části kódu, které jsou přičleněny k programu až po jeho spuštění. Obvykle jde o sdílené knihovny nebo tzv. plug-iny.
- **Dynamický linker** – `/lib/ld.so` – program, který je dynamicky přičleněn jako první. Stará se dynamické linkování knihoven.

Proměnné pro dynamický linker

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

LD_PRELOAD – cesta k dynamicky linkovanému objektu, který bude přilinkován jako první. Je možno použít například pro předefinování knihovní funkce nebo ke vnucení jiné verze knihovny.

- U set-uid a set-gid programů dynamický linker ignoruje výše uvedené proměnné.

/etc/ld.so.conf
– globální konfigurace.

Linkování v době kompilace

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

Linkování v době běhu – formát ELF

- **Extended Loadable Format**
- **AT&T/USRG SVR4, Linux libc5+**
- **Křížové odkazy** – řešeny v době běhu.
- **Problém** – nesdílitelné části kódu (křížové odkazy).
- **Řešení** – kód nezávislý na umístění (*position independent code, PIC*).
- **Verze symbolů** – při změně způsobu volání funkce apod.
- **Výhody** – dynamické linkování (např. plug-iny), možnost předefinovat symbol v knihovně.
- **Nevýhody** – pomalejší start programu, potenciálně pomalejší běh PIC kódu (je nutno alokovat jeden registr jako adresu začátku knihovny).

ldd(1) . . . . Loader dependencies

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

Vypíše, se kterými dynamickými knihovnami bude program linkován.

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

◇ **Úkol:** Zjistěte, které programy jsou v systémových adresářích `/bin` a `/sbin` (nebo `/etc`) staticky linkované.

Hlavičkové soubory

- **Definice rozhraní ke knihovnám** – typové kontroly a podobně.
- **Definice konstant** – `NULL`, `stdin`, `EAGAIN` ...
- **Definice maker** – `isspace()`, `ntohl()`, ...
- Neobsahují vlastní definice funkcí, jen deklarace prototypů.
- **Umístění:** – adresář `/usr/include` a podadresáře.
- **Poznámka k privátním symbolům:** Symboly, jejichž jméno začíná podtržítkem, jsou privátní symboly operačního systému a můžou být definovány v libovolné podobě. Proto je zakázáno používat a definovat v uživatelských programech jakékoli symboly, začínající podtržítkem.

◇ **Úkol:** Je v systému definována konstanta pro π? Ve kterém hlavičkovém souboru? Jak se jmenuje tato konstanta? Návod: projděte hlavičkové soubory.

◇ **Úkol:** Napište program, který v době běhu zjistí, jde-li o big-endian nebo little-endian systém.

Ladění programu

- **Ladící informace** – přepínač `-g` u kompilátoru.
- **Soubor `core`** – obraz paměti procesu v době havárie. Lze vytvořit i uměle například zasláním signálu `SIGQUIT` (`Ctrl-\`). Slouží k posmrtné analýze programu.
- Ladění programu probíhá přes službu jádra `ptrace(2)`, nebo přes souborový systém `/proc`.

Debuggery

sdb – symbolic debugger. Dostupný na starších UNIXech. Jednoduché řádkové ovládání.
adb – assembler debugger. Slouží k ladění programu ve strojovém kódu. Umí i disassemblovat. Ovládání podobné jako u `sdb(1)`.
dbx – pochází ze SVR4. Širší možnosti, ovládání příkazy ve formě slov.
gdb – GNU debugger. Nejrozšířenější možnosti (volání funkcí z programu, změna volací sekvence na zásobníku, atd.).
xxgdb – grafický front-end pro `gdb(1)`.
ddd – grafický front-end pro `gdb(1)` nebo `dbx(1)`.

Rozsáhlé projekty

- **`Makefile`** – závislé na systému.
- **Existence/umístění knihoven** – závislé na konkrétní instalaci.
- **Cílový adresář (adresáře)** – závislé na lokálních zvyklostech.
- **Potřeba stavět software různým způsobem**

- **GNU Autoconf**
- **GNU Automake**
- **Imake**
- **Confgen**

- **GNU Libtool** – výroba sdílených knihoven.

Normy API

ANSI C

Schváleno 1989 – ANSI Standard X3.159–1989. Jazyk C plus standardní knihovna (15 sekcí knihovny podle 15 hlavičkových souborů). Základní přenositelnost programů v C. Oproti UN*Xu nedefinuje proces ani vztahy mezi procesy.

IEEE POSIX

Portable Operating System Interface – IEEE 1003. Některé části schváleny IEEE a ISO, další se připravují. Nejdůležitější sekce normy POSIX:

- **POSIX.1** – Basic OS Interfaces: Schváleno IEEE a ISO
- **POSIX.1a** – Miscellaneous extensions.
- **POSIX.2** – Commands (sh a další): Schváleno IEEE a ISO.
- **POSIX.3** – Test methods: Schváleno IEEE.
- **POSIX.4 (1b)** – Real-time extensions: IEEE 1003.1b-1993.
- **POSIX.4a (1c)** – Threads extensions.
- **POSIX.4b (1d)** – More real-time extensions.
- **POSIX.5** – ADA binding to POSIX.1: Schváleno IEEE.
- **POSIX.6 (1e)** – Security extensions.

Další normy

- **X/Open XPG3,4** X/Open Portability Guide – rozšíření POSIX.1.
- **FIPS 151-1 a 151-2** – Federal Information Processing Standard. Upřesnění normy POSIX.1.
- **SVID3** – System V Interface Description – norma AT&T, popisující System V Release 4.
- **BSD** – označení pro extenze z 4.x BSD.

Single UNIX Specification

- **`www.unix-systems.org`**
- **Verze** – z roku 1995 a 1998 (Single-UNIX Spec. v2), tzv. UNIX 95 a 98.
- **V současné době používaná „definice UNIXu"**

Limity

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

ANSI C Limity

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

POSIX.1 a POSIX.4 limity

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE  199309
#include <unistd.h>
```

Konstanta `_POSIX_VERSION` pak určuje verzi normy POSIX, kterou systém splňuje:

- **Nedefinováno** – systém není POSIX.1.
- **198808** – POSIX.1 je podporován (FIPS 151-1).
- **199009** – POSIX.1 je podporován (FIPS 151-2).
- **199309** – POSIX.4 je podporován.
- **více než 199309** – POSIX.4 plus další možná rozšíření.

POSIX.4 vlastnosti jsou všechy volitelné v čase kompilace. Některé POSIX.1 konstanty: `ARG_MAX`, `CHILD_MAX`, `PIPE_BUF`, `LINK_MAX`, `_POSIX_JOB_CONTROL`.

sysconf . . . . Run-time limity v POSIX.1

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

Slouží k získání limitů pro dobu běhu, nezávislých na souboru (maximální délka cesty, maximální počet argumentů).

pathconf, fpathconf . . . . Limity závislé na souboru

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

Získání run-time limitů závislých na souboru (maximální délka jména souboru, maximální počet odkazů a podobně).

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

◇ **Úkol:** Zjistěte a srovnejte POSIX.1 run-time a compile-time limity různých systémů.

Start programu

- **Linkování programu** – `crt0.o`, objektové moduly, knihovny, `libc.a` (nebo `libc.so`).
- **Vstupní bod** – závislý na binárním formátu. Ukazuje obvykle do `crt0.o`.
- **Mapování sdílených knihoven** – namapování dynamického linkeru do adresového prostoru procesu; spuštění dynamického linkeru.
- **Inicializace** – například konstruktory statických proměnných v C++. V GCC voláno z funkce `__main`.
- **Nastavení globálních proměnných (`environ`)**.
- **Volání funkce `main()`**.

main . . . . Vstupní bod programu

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

`argc` – počet argumentů programu + 1.
`argv` – pole argumentů.
`envp` – pole proměnných z prostředí procesu (⟨*jméno*⟩=⟨*hodnota*⟩).

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

Ukončení programu

Proces vrací volajícímu procesu návratovou hodnotu – obvykle osmibitové celé číslo se znaménkem. Jednou z možností ukončení procesu je ukončení funkce `main()`.

exit . . . . Ukončení procesu

```
#include <stdlib.h>
void exit(int status);
```

Knihovní funkce; pokusí se uzavřít otevřené soubory, zavolat destruktory statických objektů (v C++) a ukončit proces.

atexit . . . . Vyvolání funkce při `exit()`

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

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

_exit . . . . Ukončení procesu

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

Služba jádra pro ukončení procesu. Je volána například ze standardní funkce `exit()`.

abort . . . . Násilné ukončení

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

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

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

Práce s argumenty programu

Bývá zvykem akceptovat přepínače (volby) s následující syntaxí:

-⟨*písmena*⟩
-⟨*písmeno*⟩ ⟨*argument*⟩
-- (ukončení přepínačů)
--⟨*slovo*⟩
--⟨*slovo*⟩ ⟨*argument*⟩
--⟨*slovo*⟩**=**⟨*argument*⟩

◇ **Příklad:**

```
$ diff -uN --recursive --ifdef=PRIVATE -- \
       linux-2.0.0 linux
```

◇ **Úkol:** Jak smažete soubor jménem `-Z`?

getopt . . . . Zpracování přepínačů

```
int getopt(int argc, char **argv,
           char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

◇ **Příklad:**

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

getopt_long . . . . Zpracování dlouhých přepínačů

Není součástí standardu POSIX.1. Je použito například v GNU programech.

- **POPT** – knihovna na procházení příkazové řádky.
  `ftp://ftp.redhat.com/pub/redhat/code/popt/`

errno . . . . Chybová hodnota služby jádra

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

V případě chyby v průběhu služby jádra je sem uloženo číslo chyby. Viz `<sys/errno.h>`, `<linux/errno.h>` a stránky `errno(3)` a `errno(7)`.

◇ **Příklad:**

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


perror . . . . Tisk chybového hlášení

```
void perror(char *msg);
```

vytiskne zprávu `msg` a textovou informaci na základě proměnné `errno`.

◇ **Příklad:**

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

Tento kód při spuštění a chybě `ENOENT` vypíše tento chybový výstup:

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

strerror . . . . Získání chybové zprávy

```
char *strerror(int errnum);
```

Funkce `perror(3)` a `strerror(3)` používají hlášení v poli zpráv `sys_errlist`, které má `sys_nerr` položek:

```
#include <errno.h>
extern char *sys_errlist[];
extern int sys_nerr;
```


Proměnné prostředí

Environment variables. Pole řetězců tvaru ⟨*jméno*⟩=⟨*hodnota*⟩. Je dostupné přes třetí argument funkce `main()` nebo přes globální proměnnou `environ`:

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

Této proměnné používat pouze pro získání obsahu všech proměnných prostředí. Jinak používat následující knihovní funkce:

getenv . . . . Získání obsahu proměnné

```
char *getenv(char *name);
```

K jménu proměnné vrátí obsah proměnné.

putenv . . . . Nastavení proměnné

```
int putenv(char *str);
```

Do prostředí zařadí danou proměnnou (řetězec `str` má opět syntaxi ⟨*proměnná*⟩=⟨*hodnota*⟩).

Funkce `getenv(3)` a `putenv(3)` odpovídají normě POSIX.1. Kromě toho lze ještě nalézt tyto funkce:

```
int setenv(char *name, char *value,
           int rewrite);
int unsetenv(char *name);
```

◇ **Úkol:** Zjistěte, ve které části adresového prostoru procesu jsou uloženy jeho argumenty a jeho proměnné prostředí. Mění se umístění proměnných, přidáváte-li do prostředí nové proměnné?

Práce s pamětí

malloc, calloc, realloc, free . . . . Alokace paměti

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

Funkce vrátí ukazatel na prostor o velikosti minimálně `size` bajtů paměti. Ukazatel je zarovnán pro libovolný typ proměnné.

```
void *calloc(size_t nmemb, size_t size);
```

Funkce alokuje místo pro `nmemb` objektů velikosti `size`. Toto místo je inicializováno nulami.

```
void *realloc(void *ptr, size_t size);
```

Změna velikosti dříve alokovaného místa. Tato funkce může přemístit data na jiné místo, není-li na stávajícím místě prostor pro rozšíření.

```
void free(void *ptr);
```

Uvolní místo, alokované dříve pomocí výše uvedených funkcí. Pozor: Některé systémy neakceptují `free(NULL)`.

alloca . . . . Alokace na zásobníku

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

Alokuje dočasné místo v zásobníkové oblasti volající funkce. Po ukončení této funkce je místo automaticky uvolněno. Funkce `alloca(3)` není dostupná na všech systémech.

brk, sbrk . . . . Změna velikosti datového segmentu

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

Tyto služby jádra slouží k nastavení velikosti datového segmentu. Jsou používány například funkcemi typu `malloc()`.

Alokace paměti je častým zdrojem chyb (uvolnění paměti, která předtím nebyla alokována, překročení přiděleného rozsahu paměti, uchovávání ukazatelů po realloc, atd.)

Ladící prostředky pro odhalení podobných chyb: ElectricFence, předefinování `malloc()` a `free()` a kontrola argumentů `free()`.

Většina implementací `malloc(3)` neumí vracet uvolněnou paměť zpět operačnímu systému.

Nelokální skoky

Tento mechanismus lze použít pro násilné ukončení několika vnořených funkcí (například v případě fatálních chyb programu).

setjmp . . . . Inicializace skoku

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

- **Inicializuje návratové místo**
- **Při prvním volání vrací nulu**
- **Struktura `jmp_buf`** – návratová adresa, vrchol zásobníku.

longjmp . . . . Nelokální skok

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

- **Skok na místo volání `setjmp()`**
- **Návratová hodnota je tentokrát `retval`**

Příklad použití vzdáleného skoku

```
#include <setjmp.h>
jmp_buf env;
int main()
{
      if (setjmp(env) != 0)
           dispatch_error();
      ...
      somewhere_else();
      ...
}
void somewhere_else()
{
      ...
      if (fatal_error)
           longjmp(env, errno);
      ...
}
```


Dynamické linkování

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

`dlopen(3)` . . . . Otevření dynamického objektu

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

Otevře dynamicky linkovaný objekt, přidá jej k procesu a případně vyřeší křížové odkazy. Je-li v objektu definován symbol `_init`, zavolá jej jako funkci (používá se např. u konstruktorů statických proměnných v C++). `flag` může být jedno z následujících:

`RTLD_NOW` – volání vyřeší křížové odkazy a vrátí chybu, jsou-li nedefinované symboly.

`RTLD_LAZY` – křížové odkazy se řeší až v okamžiku, kdy je kód z knihovny poprvé proveden. Právě jeden z těchto dvou flagů musí být použit.

`RTLD_GLOBAL` – globální symboly z knihovny jsou dány k dispozici dalším později linkovaným knihovnám.

`dlclose(3)` . . . . Uzavření dynamické knihovny

```
#include <dlfcn.h>
int dlclose(void *handle);
```

Uzavře a odmapuje dynamický objekt. Toto se stane až poté, co je `dlclose(3)` na tento objekt zavoláno tolikrát, kolikrát bylo předtím spuštěno `dlopen(3)`. Obsahuje-li dynamický objekt symbol `_fini`, je interpretován jako funkce a tato je zavolána před odmapováním knihovny (použití: destruktory statických proměnných v C++).

`dlsym(3)` . . . . Získání symbolu z knihovny

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

Vrátí adresu daného symbolu v dynamické knihovně.

`dlerror(3)` . . . . Chybové hlášení dynamického linkeru

```
#include <dlfcn.h>
char *dlerror();
```

Vrátí řetězec s chybovým hlášením v případě, že nastala chyba u některé funkce pro dynamické linkování, nebo NULL. Další volání téže funkce vrací opět NULL.

◇ **Příklad:** Načte matematickou knihovnu a vypíše kosinus 1.0.

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

◇ **Úkol:** Vytvořte následující program:

`$ callsym` ⟨*knihovna*⟩ ⟨*symbol*⟩

Tento program načte jmenovanou knihovnu a zavolá ⟨*symbol*⟩ jako funkci bez parametrů. Doplňte program o testování návratových hodnot funkcí `dl*` a v případě chyby vypisujte chybové hlášení pomocí `dlerror(3)`.

Lokalizace

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

Kategorie lokalizace

`LC_COLLATE` – třídění řetězců.
`LC_CTYPE` – typy znaků (písmeno, číslice, nepísmenný znak, převod velká/malá písmena, atd).
`LC_MESSAGES` – jazyk, ve kterém se vypisují zprávy (viz též GNU gettext).
`LC_MONETARY` – formát měnových řetězců (znak měny, jeho umístění, počet desetinných míst, atd).
`LC_NUMERIC` – formát čísla (oddělovač desetin, oddělovač tisícovek apod.)
`LC_TIME` – formát času, názvy dní v týdnu, měsíců atd.

Názvy locales

⟨*jazyk*⟩[_⟨*teritorium*⟩][.⟨*charset*⟩][@⟨*modifikátor*⟩]

- **Jazyk** – dle ISO 639 (pro nás `cs`)
- **Teritorium** – dle ISO 3316 (pro nás `CZ`)
- **Znaková sada** – například (ISO8859-2 nebo UTF-8)
- **Příklady** `cs_CZ.ISO8859-2, cs, cs_CZ, en_GB.UTF-8`

`setlocale(3)` . . . . Nastavení lokalizace

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

Nastavení/zjištění hodnoty locale. Pokud je `locale` rovno `NULL`, jen vrátí stávající nastavení. Pokud je `locale` rovno `""`, nastaví hodnotu podle proměnných prostředí. Jinak nastaví hodnotu podle textu v řetězci `locale`.

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

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


Proměnné prostředí

`LANG` – implicitní hodnota pro všechny kategorie.
`LC_*` – nastavení jednotlivých kategorií.
`LC_ALL` – přebíjí výše uvedená nastavení pro všechny kategorie.

`strcoll(3)` . . . . Porovnávání řetězců podle locale

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

Funguje podobně jako `strcmp(3)`, jen bere ohled na nastavení hodnoty `LC_COLLATE`.

`strxfrm(3)` . . . . Trabsformace řetězce podle locale

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

Převede řetězec `src` na řetězec `dest` délky maximálně `len` tak, že výsledek porovnání takto získaných řetězců pomocí `strcpy(3)` je ekvivalentní porovnání původních řetězců pomocí `strcoll(3)`.

Pokud je potřeba alespoň `len` znaků, je hodnota `dest` nedefinována.

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

Katalogy zpráv

- **P**ro kategorii `LC_MESSAGES`.
- **G**NU `gettext` – překladové tabulky, vyhledávání řetězců.

Další programy

`locale(1)` . . . . Lokalizačně specifické informace

Bez parametrů vypíše informace o právě nastavých locales. S parametrem `-a` vypíše všechny definované locales. Jako parametr lze dát konkrétní vlastnost locale, například:

```
$ locale charmap
UTF-8
$ locale mon
leden;únor;březen;duben;květen;...
```

`localedef(8)` . . . . Kompilace lokalizačního souboru

`$ localedef [-f` ⟨*charmap*⟩`] [-i` ⟨*inputfile*⟩`]` ⟨*outdir*⟩

Vytvoří binární podobu locale pro přímé použití v aplikacích.

Start systému

Firmware

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

Primární zavaděč systému

- Program v boot bloku disku; pevná délka; zavádí sekundární zavaděč.

Sekundární zavaděč systému

- Načítá jádro, předává mu parametry.
- Někdy poskytuje příkazový řádek.
- Někdy umí číst souborový systém.
- Používá firmware k zavedení jádra.

Start jádra

Parametry jádra

- Systémová konzola a kořenový disk.
- Parametry pro ovladače zařízení.
- Ostatní parametry předány do uživatelského prostoru.

Průběh inicializace jádra

- Virtuální paměť – co nejdříve.
- Inicializace konzoly.
- Inicializace sběrnic. (Autokonfigurovaná zařízení.)
- Inicializace CPU.
- Inicializace zařízení.
- Vytvoření procesu číslo 0 (idle task, swapper, scheduler).
- Start kernel threadů (`kflushd`, `kswapd`).
- Inicializace ostatních CPU (a start idle procesů).
- Připojení kořenového systému souborů.
- Start procesu číslo 1 v souboru `/sbin/init`.

Inicializace zařízení

- **UNIX v7** – bloková/znaková zařízení, statické tabulky (`bdevsw[]`, `cdevsw[]`).
- **Linux** – bloková/znaková/SCSI/síťová zařízení, dynamické tabulky.
- **Obsluha zařízení** – funkce pro otevření, čtení, zápis, řídící operace, atd. Privátní data zařízení.

Detekce zařízení

- **Autokonfigurovaná zařízení** – např. PCI, SBUS a podobně.
- **Empirické testy** – ISA. Zápis na nějaký port, očekávání reakce zařízení. Záleží na pořadí testů, může dojít až k zablokování sběrnice (síťové karty NE 2000).
- **Hot-swap zařízení** – konfigurace počítače se mění v době provozu (CardBus, USB, hot-swap SCSI, atd).


```
Linux version 2.4.0-test9 (kron@pyrrha.fi.muni.cz) \
(gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 \
release)) #1 Mon Oct 9 08:32:07 CEST 2000
BIOS-provided physical RAM map:
 BIOS-e820: 000000000009fc00 @ 0000000000000000 (usbl)
 BIOS-e820: 0000000000000400 @ 000000000009fc00 (rsvd)
 BIOS-e820: 0000000000010000 @ 00000000000f0000 (rsvd)
 BIOS-e820: 0000000007ef0000 @ 0000000000100000 (usbl)
On node 0 totalpages: 32752
zone(0): 4096 pages.
zone(1): 28656 pages.
Kernel command line: auto BOOT_IMAGE=linux ro \
root=1601 console=ttyS1,38400n8
Initializing CPU#0
Detected 677.944 MHz processor.
Console: colour VGA+ 80x25
Calibrating delay loop... 1353.32 BogoMIPS
Memory: 126748k/131008k available (1225k kernel code, \
3872k reserved, 73k data, 176k init, 0k highmem)
```



```
Dentry-cache hash table entries: 16384 (order: 5, \
131072 bytes)
Buffer-cache hash table entries: 4096 (order: 2, \
16384 bytes)
VFS: Diskquotas version dquot_6.4.0 initialized
CPU: L1 I Cache: 64K  L1 D Cache: 64K (64 bytes/line)
CPU: L2 Cache: 512K
CPU: AMD Athlon(tm) Processor stepping 01
Checking 'hlt' instruction... OK.
mtrr: v1.36 (20000221) Richard Gooch
PCI: PCI BIOS revision 2.10 entry at 0xfdaf1, lbus=1
PCI: Using configuration type 1
PCI: Probing PCI hardware
PCI: Using IRQ router default [1022/740b] at 00:07.3
Linux NET4.0 for Linux 2.4
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 8192 bind 8192)
```



```
Starting kswapd v1.8
pty: 256 Unix98 ptys configured
Uniform Multi-Platform E-IDE driver Revision: 6.31
ide: Assuming 33MHz system bus speed for PIO modes; \
override with idebus=xx
AMD7409: IDE controller on PCI bus 00 dev 39
    ide0: BM-DMA at 0xf000-0xf007, BIOS settings: \
    hda:DMA, hdb:pio
hda: QUANTUM FIREBALLP LM10.2, ATA DISK drive
ide0 at 0x1f0-0x1f7,0x3f6 on irq 14
hda: 20066251 sectors (10274 MB) w/1900KiB Cache, \
CHS=1249/255/63, UDMA(66)
Partition check:
 /dev/ide/host0/bus0/target0/lun0: p1 p2 p3
Serial driver version 5.02 (2000-08-09) with \
MANY_PORTS SHARE_IRQ SERIAL_PCI enabled
ttyS00 at 0x03f8 (irq = 4) is a 16550A
eepro100.c:v1.09j-t 9/29/99 Donald Becker
eth0: Intel Corporation 82557 [Ethernet Pro 100], \
00:D0:B7:6B:4A:B2, IRQ 11.
```



```
  Board assembly 721383-008, Physical connectors \
  present: RJ45
  Primary interface chip i82555 PHY #1.
Linux agpgart interface v0.99 (c) Jeff Hartmann
agpgart: Maximum main memory to use for agp memory: 94M
agpgart: Detected AMD Irongate chipset
agpgart: AGP aperture is 64M @ 0xe0000000
devfs: v0.102 (20000622) Richard Gooch
devfs: boot_options: 0x2
kmem_create: Forcing size word alignment - nfs_fh
VFS: Mounted root (ext2 filesystem) readonly.
Freeing unused kernel memory: 176k freed
```

Hlášení jádra lze vypsat příkazem `dmesg(8)`.

Konfigurace jádra

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

Monolitické jádro

- Jeden soubor na disku.
- Všechny používané ovladače jsou uvnitř jádra.
- Často bez autodetekce zařízení.
- Paměť dostupná všem částem jádra stejně.

Mikrokernel

- CMU Mach, OSF Mach, minix, NT HAL, QNX.
- Co nemusí být v kernelu, dát mimo něj.
- Procesy (servery) pro správu virtuální paměti, ovládání zařízení, disků a podobně.
- Dobře definovatelné podmínky činnosti.
- Předávání zpráv – malá propustnost, velká latence.

Modulární jádro

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

Modulární jádro v Linuxu

- Dynamické přidávání ovladačů podle potřeby.
- Kernel daemon/`kmod`.
- Závislosti mezi moduly (`depmod(8)`).
- Dynamická registrace modulů: `register_chrdev()`, `register_blkdev()`, `register_netdev()`, `register_fs()`, `register_binfmt()` a podobně.

Bootování s ramdiskem

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

Ramdisk v Linuxu

- **Komprimovaný soubor**
- **Obraz souborového systému**
- **Startovací skript `/linuxrc`**
- **Mimo jiné určení kořenového svazku**
- **Po ukončení** – přemontování jako `/initrd`.

Architektura jádra

- **Při startu** – kontext procesu číslo 0 – později idle task. Idle task nemůže být zablokován uvnitř čekací rutiny.
- **Kontext** – stav systému, příslušný běhu jednoho procesu.
- **Přepnutí kontextu** – výměna právě běžícího procesu za jiný.
- **Linux** – `struct task_struct`, `current`.

- **Problém:** Pod jakým kontextem mají běžet služby jádra?
- **UNIX** – použije se kontext volajícího procesu. Proces pak má dva režimy činnosti – user-space a kernel-space.
- **Mikrokernel** – předá se řízení jinému procesu (serveru).

- **Problém:** Pod jakým kontextem lze provádět přerušení?
- **Zvláštní kontext** – nutnost přepnutí kontextu $\longrightarrow$ zvýšení doby odezvy (latence) přerušení. Navíc je nutno případně mít více kontextů pro možná paralelně běžící přerušení.
- **UNIX** (ve většině implementací): Přerušení se provádí pod kontextem právě běžícího procesu. Obsluha přerušení nesmí zablokovat proces.

Přerušení

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

- **Peemptivní/nepreemptivní jádro** – může dojít k přepnutí kontextu kdekoli v jádře?

Odložené vykonání kódu

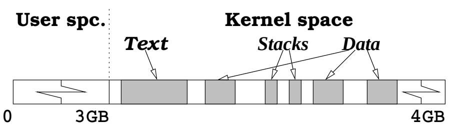
- **Funkce, vykonaná později** (po návratu z přerušení, při volání scheduleru, atd.)
- **Spodní polovina** obsluhy přerušení.
- **Časově nekritický kód**
- **Může být přerušen**
- **Linux** – bottom half (serializováno globálně). Tasklety (serializovány pouze vzhledem k sobě).

Virtuální paměť

- **Virtuální adresa** – adresa z hlediska instrukcí CPU.
- **Překlad mezi virtuální a fyzickou adresou** – stránková tabulka (obvykle dvou- nebo tříúrovňová). Každý proces má svoji virtuální paměť $\Longrightarrow$ každý proces má svoji stránkovou tabulku.
- **Výpadek stránky** (page fault) – stránka není v paměti, stránkový adresář neexistuje, stránka je jen pro čtení a podobně.
- **Obsluha výpadku stránky** – musí zjistit, jestli jde (například) o copy-on-write, o žádost o natažení stránky z odkládacího prostoru, o naalokování stránky, nebo jestli jde o skutečné porušení ochrany paměti procesem.
- **TLB** – translation look-aside buffer – asociativní paměť několika posledních použitých párů *(virtuální adresa, fyzická adresa)*.
- **Přepnutí kontextu** – vyžaduje vyprázdnění TLB, v případě virtuálně adresované cache také vyprázdnění cache. Proto je přepnutí mezi vlákny rychlejší.
- **Softwarový TLB** – OS-specifický formát stránkových tabulek.

Virtuální paměť z hlediska jádra

- **Kód a paměť jádra** – mapován obvykle na konec adresního prostoru.
- **Přepnutí do režimu jádra** – zpřístupnění horních (virtuálních) adres.
- **Fyzická paměť** – mapována také 1:1 do paměťové oblasti jádra (Linux bez `CONFIG_HIGHMEM`).
- **Zásobník** – pro každý thread/kontext (Linux – 1 stránka/thread, nastavitelné 2 stránky/thread).



- **Jiný přístup** – přepnutí stránkové tabulky při přepnutí do jádra (rozdělení 4:4 GB na 32-bitových systémech).
- **Použití víc než 4 GB paměti na 32-bitových systémech** – Intel PAE, 36-bitová fyzická adresa.

Paměť z hlediska hardwaru

- **Fyzická adresa** – adresa na paměťové sběrnici, vycházející z CPU (0 je to, co CPU dostane, vystaví-li nuly na všechny bity adresové sběrnice).
- **Virtuální adresa** – interní v CPU. Instrukce adresují paměť touto adresou.
- **Sběrnicová adresa** – adresa místa v paměti tak, jak je vidí ostatní zařízení. V některých případech stejná jako fyzická, u některých architektur vlastní MMU pro sběrnici (Sun 4M IOMMU), případně vlastní MMU na zařízení.

◇ **Příklad:** PowerPC Reference Platform (PRep):

- **Z hlediska CPU**: 0–2GB fyzická paměť, 2GB–3GB system I/O (ekvivalent adres I/O portů pro `inb()` a `outb()` na x86, 3GB–4GB I/O memory (sdílená paměť na I/O sběrnici; ekvivalent sdílené paměti mezi 640 KB a 1 MB na x86).
- **Z hlediska HW**: Fyzická paměť je na 2GB–4GB, I/O adresy jsou v prvních dvou GB.

◇ **Příklad:** Intel Xeon PAE (36-bitová fyzická paměť, virtuální i sběrnicová adresa je ale 32-bitová).

Přístup do uživatelského prostoru

- Není možný uvnitř ovladače přerušení (aktuální kontext není v žádném vztahu k probíhající operaci).
- Je nutné kontrolovat správnost ukazatelů z uživatelského prostoru.
- Chybná uživatelská data nesmí způsobit pád jádra.
- Problémy ve vícevláknových programech (přístup versus změna mapování v jiném vláknu).

Řešení Linuxu

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

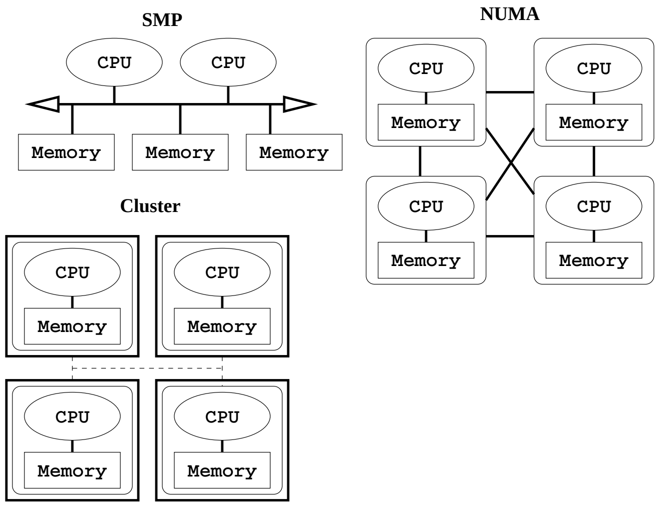

Implementace v Linuxu

- **Využití hardwaru CPU** – kontrola přístupu do paměti. Přidání kontroly do `do_page_fault()`.
- **Makra preprocesoru**
- **Tabulka výjimek** – adresa instrukce, která může způsobit chybu, opravný kód.
- **Normální běh** – cca 10 instrukcí bez skoku.
- **ELF sekce**

Paralelní stroje

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

Paralelní stroje



Zamykání kódu

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

Na jednom CPU

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

Na paralelním systému

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

Semafory

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

Spinlocky

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

R/W zámky

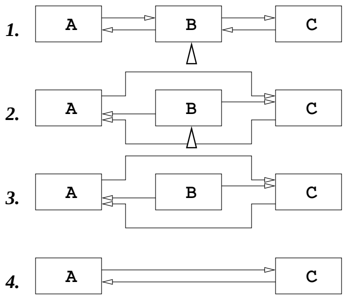
- **Paralelní čtení – exkluzivní zápis**
- **Linux** – `struct rwlock`, `struct rwsem`.
- **Problémy** – priority? upgrade r-zámku na w-zámek (deadlock).

Read-copy-update

- **RCU** – původně Sequent (Dynix/PTX), později IBM, implementace i v Linuxu.
- **Atomické instrukce** – pomalé (stovky taktů; přístup do hlavní paměti).
- **Obvyklá cesta** (např. čtení) by měla být rychlá.
- **Kód bez zamykání** – ale omezující podmínky (například držení odkazu na strukturu).
- **Linux** – omezující podmínka – přepnutí kontextu na všech procesorech. Odložené vykonání funkce po splnění podmínky.
- **Slabě uspořádané architektury** – instrukce čtení (někde i instrukce zápisu) mohou být přeuspořádány. Nutnost explicitních paměťových bariér (speciální instrukce CPU).

Read-copy-update – příklad

◇ **Příklad:** Rušení prvku ze seznamu



- **Čekání na splnění omezující podmínky** – bezi body 2. a 3. Využívá se odloženého spuštění kódu.

Časovače

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

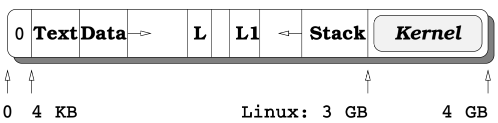
Čekací fronty

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

Procesy

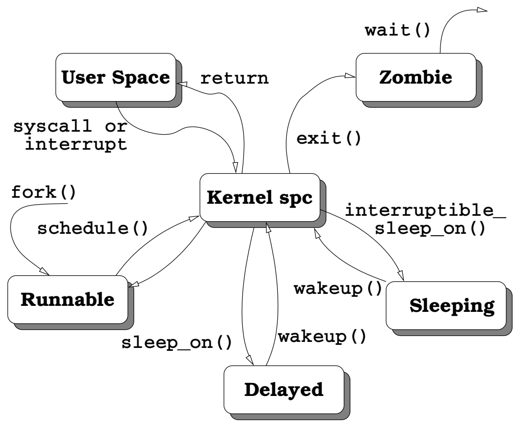
- **Proces** – běžící program.
- **Proces** – kontext procesoru se samostatnou VM.
- **Thread** – kontext bez samostatné VM.

Paměť procesu:



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

Stavy procesu



Atributy procesu

- **Stav procesu**
- **Program counter** – čítač instrukcí; místo, kde je proces zablokován.
- **Číslo procesu** – PID.
- **Rodič procesu** – PPID (rovno 1, pokud neexistuje).
- **Vlastník procesu** – (real) UID.
- **Skupina procesu** – (real) GID.
- **Skupina procesů, session** – seskupování procesů do logických celků.
- **Priorita procesu**
- **Reakce na signály, Čekající signály**
- **Časy běhu**
- **Pracovní a kořenový adresář**
- **Tabulka otevřených souborů**
- **Odkazy na potomky**
- **Limity** – na velikost souboru, max. spotřebovaný čas, max. počet otevřených souborů atd (`setrlimit(2)`).

Atributy procesu lze číst programem `ps(1)`. Funguje nad virtuálním souborovým systémem `/proc` nebo nad `/dev/mem`.

Služba jádra

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

Knihovní funkce

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

Vznik procesu

`fork(2)` . . . . Vytvoření procesu

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

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

Potomek dědí téměř vše od rodiče. Vyjímkou jsou čísla PID, PPID, zámky na souborech, návratová hodnota `fork(2)`, signál od časovače, čekající signály, hodnoty spotřebovaného strojového času.

`vfork(2)` . . . . Virtuální fork()

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

Vytvoří potomka bez kopírování adresového prostoru. Rodič je pozastaven dokud potomek nevyvolá `exec(2)` nebo `_exit(2)`.

Zavedeno původně jako BSD extenze.

Čekání na ukončení potomka

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

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

`WIFEXITED(status)`
: – proces skončil pomocí `_exit(2)`. Návratový kód zjistíme pomocí `WEXITSTATUS(status)`.

`WIFSIGNALED(status)`
: – potomek byl ukončen signálem. Číslo signálu zjistíme pomocí `WTERMSIG(status)`. Navíc SVR4 i 4.3BSD (ale ne POSIX.1) definují makro `WCOREDUMP(status)`, které nabývá hodnoty pravda, byl-li vygenerován `core` soubor.

`WIFSTOPPED(status)`
: – proces byl pozastaven. Důvod pozastavení zjistíme makrem `WSTOPSIG(status)`.

Parametr `options` je nula nebo logický součet následujících:

`WNOHANG` – nezablokouje se čekáním.
`WUNTRACED` – i při pozastavení nebo ladění potomka.

`wait3(2), wait4(2)` . . . . Čekání na ukončení potomka

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

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

◇ **Příklad:** Použití `fork()` a `wait()`

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

Spuštění programu

exec(3) . . . . Spuštění procesu

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

Nahradí text procesu jiným textem a začne tento text vykonávat. Uzavře deskriptory, které mají flag `FD_CLOEXEC`. Tento flag je zejména normou POSIX.1 vyžadován u adresářových deskriptorů.

Obvykle je `execve(2)` a zbytek jsou knihovní funkce implementované pomocí `execve(2)`. Všechny funkce jsou označovány souhrnně jako `exec(2)` nebo `exec(3)`.

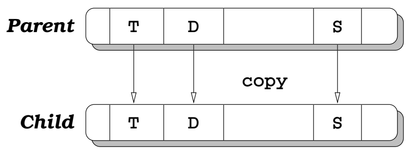
system(3) . . . . Vyvolání příkazu shellu

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

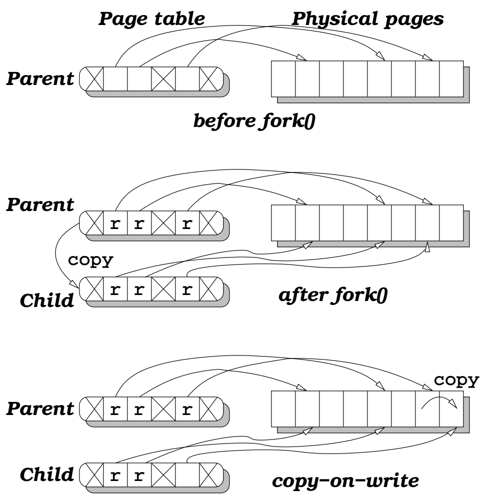
Spustí příkaz `/bin/sh -c string` jako potomka a počká na jeho dokončení.

`fork(2)` v moderních systémech

- **Systém bez stránkování** – nutnost kopírování celého adresního prostoru. Ve většině případů následuje `exec(2)` ⟶ nový adresní prostor se znovu přepíše.



- **Systém se stránkováním** – kopíruje se jen stránková tabulka (několik KB). Stránky se nastaví jako r-only a kopírují se až při zápisu.



Systémy se stránkováním

- **Unifikovaný systém diskových bufferů a virtuální paměti** – některé stránky mohou být sdíleny s diskovými buffery – stránka má svůj *obraz v souboru*.
- **fork(2)** – sdílení dat mezi rodičem a potomkem, copy-on-write.
- **Další možnosti sdílení**

Sdílení textu programu

- **Sdílení s diskovými buffery** – o určité stránce se např. ví, že je to stránka ze souboru `/bin/sh`, offset 8192 bajtů. Takto může být stránka namapována do adresního prostoru více procesů.
- **Sdílené knihovny** – úplně stejný mechanismus.

Stránkování na žádost

- **Demand-paging**
- **Text procesu** se nenačítá do paměti, pouze se pro každou stránku v textové části adresního prostoru označí, kde má své místo v souboru.
- **Přístup k textu** ⟶ výpadek stránky; stránka se načte ze souboru.
- **Při nedostatku paměti** lze stránky v textu přímo rušit (bez zápisu na odkládací zařízení). Později je možno je zase načíst ze souboru. *Text file busy*.
- **Výhoda** – nenačítá se celý text, který se možná ani nevyužije (případ chybných parametrů na příkazové řádce a podobně).

I/O operace

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

Alokace paměti

- **Služba sbrk(2)** pouze posune konec dat, ale nealokuje nové stránky.
- **Přístup k nově alokovanému prostoru** – výpadek stránky, obsluha přidělí novou stránku, namapuje ji a restartuje instrukci, která způsobila výpadek.
- **Výhody** – paměť se přiděluje až v okamžiku použití. Viz pole ve Fortranu.
- **Memory overcommitment** – má systém počítat, kolik paměti ještě „dluží“ procesům? Ano ⟶ nedojde tak brzo k vyčerpání zdrojů, většina programů stejně všechnu přidělenou paměť nevyužije. Ne ⟶ nenastane situace, kdy OS nemůže dostát svým slibům a musí násilně ukončit proces.

Výhody stránkovacích systémů

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

Program na disku

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

Binární formát script

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

Starší binární formáty

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

Binární formát ELF

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

Změna práv procesu

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

getuid(2), getgid(2) . . . . Vlastník/skupina procesu

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

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

Zjištění reálného a efektivního UID případně GID procesu.

setuid(2), setgid(2) . . . . Změna efektivního UID/GID

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

- Pokud proces má superuživatelská práva, nastaví služba `setuid(2)` reálné, efektivní i uložené UID na `uid`.
- Pokud proces nemá práva superuživatele, ale `uid` je rovno reálnému nebo uloženému UID, změní `setuid(2)` pouze efektivní UID na `uid`.
- Jinak končí s chybou a proměná `errno` je nastavena na `EPERM`.

setreuid(2) . . . . Výměna reálného za efektivní ID

```
#include <sys/types.h>
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

Funkce definovaná v 4.3BSD. Umožňuje výměnu reálného a efektivního UID v systémech bez uloženého UID.

Uložené ID

- Pokud je definováno `_POSIX_SAVED_IDS`
- SVR4 podporuje uložená ID.
- FIPS 151–1 vyžaduje tuto vlastnost.

- Pouze superuživatel může měnit reálné UID.
- Efektivní UID je nastaveno funkcí `exec(2)`, pokud má příslušný program nastavený set-uid bit. Jinak se efektivní UID nemění.
- Při `exec(2)` se kopíruje uložené UID z efektivního UID.

◇ **Příklad:** Mějme set-uid program, který patří uživateli číslo 1 a je spuštěn uživatelem číslo 2. UID procesu se může měnit například takto:

Akce	real UID	effective UID	saved UID
Start programu	2	1	1
`setuid(2)`	2	2	1
`setuid(1)`	2	1	1
`exec()`	2	1	1
nebo:			
`setuid(2)`	2	2	1
`exec()`	2	2	2

seteuid(2) . . . . Nastavení efektivního UID

```
#include <sys/types.h>
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
```

Navrhovaná změna normy POSIX.1: Umožní superuživatelskému procesu změnit efektivní UID beze změny ostatních dvou UID. Vyžaduje systém, podporující uložené UID.

Doplňková GID

- **Starší verze UNIXu** – při přihlášení uživatele: UID a GID podle souboru /etc/passwd, změna GID pomocí newgrp(1).
- **4.2 BSD** – doplňková GID. Proces má kromě reálného, efektivního a uloženého GID navíc ještě seznam *doplňkových GID (supplementary GIDs)*, který se inicializuje při přihlášení podle /etc/group.
- **Přístupová práva** – kontrolují se vzhledem k efektivnímu GID a všem doplňkovým GID.
- **POSIX.1** – doplňková GID jsou volitelnou vlastností. Konstanta NGROUPS_MAX určuje, kolik max. doplňkových GID může být. Je-li rovna nule, systém nepodporuje doplňková GID.
- **SVR4 a 4.3+BSD** – podporují doplňková GID.
- **FIPS 151–1** – vyžaduje podporu doplňkových GID a hodnotu NGROUPS_MAX aspoň 8.

getgroups(2) . . . . Získání doplňkových GID

```
#include <sys/types.h>
#include <unistd.h>
int getgroups(int size, gid_t grouplist[]);
```

Do pole grouplist[] uloží doplňková GID až do počtu size. Vrátí počet skutečně zapsaných položek pole grouplist[]. Speciální případ: je-li size nulové, vrátí počet doplňkových GID pro daný proces.

setgroups(2) . . . . Nastavení doplňkových GID

```
#include <sys/types.h>
#include <unistd.h>
int setgroups(int size, gid_t grouplist[]);
```

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

initgroups(3) . . Nastavení doplňkových GID podle /etc/group

```
#include <grp.h>
#include <sys/types.h>
int initgroups(char *user, gid_t group);
```

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

Další atributy procesu

getpid(2), getppid(2) . . . . Čísla procesu

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

Zjištění čísla procesu a čísla rodičovského procesu.

Skupiny procesů

- **Každý proces je v právě jedné skupině**
- **V každé skupině je jeden vedoucí proces**
- **Číslo skupiny je číslo vedoucího procesu**
- **Použití** – zasílání signálu, přístup k terminálu (viz termios(4)).

setpgid(2), setpgrp(2) . . . . Nastavení kupiny procesů

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
pid_t setpgrp(void);
```

pid je číslo procesu, pgid je číslo skupiny procesů. Je-li některé z nich 0, bere se PID aktuálního procesu.
setpgrp() je totéž co setpgid(0, 0).

getpgid(2), getpgrp(2) . . . . Zjištění skupiny procesů

```
#include <unistd.h>
pid_t getpgid(pid_t pid);
pid_t getpgrp(void);
```

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

Systémové zdroje

times(2) . . . . Získání časových informací o procesu

```
#include <sys/times.h>
clock_t times(struct tms *buf);
struct tms {
   time_t tms_utime;
   time_t tms_stime;
   time_t tms_cutime;
   time_t tms_cstime;
}
```

Spotřebovaný čas v uživatelském prostoru a v prostoru jádra (v rámci procesu a včetně potomků).

getrusage(2) . . . . Spotřebované systémové zdroje

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
int getrusage(int who, struct rusage *r);
```

Zjistí spotřebované systémové zdroje. Parametr who je buďto RUSAGE_SELF nebo RUSAGE_CHILDREN.

getrlimit(2), setrlimit(2) . . . . Limity systémových zdrojů

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, struct rlimit *rlim);
```

Parametr resource může být jeden z následujících:

RLIMIT_CORE	velikost souboru core
RLIMIT_CPU	strojový čas
RLIMIT_FSIZE	velikost vygenerovaného souboru
RLIMIT_DATA	velikost datové oblasti
RLIMIT_STACK	velikost zásobníku
RLIMIT_RSS	resident set size (většinou neimplementováno)
RLIMIT_NPROC	počet procesů daného uživatele
RLIMIT_NOFILE	počet otevřených souborů
RLIMIT_MEMLOCK	uzamčená paměť
RLIMIT_AS	velikost virtuální paměti

Není součástí normy POSIX.1, ale je v BSD 4.3 a SVr4.

time(2) . . . . Systémový čas

```
#include <time.h>
time_t time(time_t *t);
```

Získá systémový čas v sekundách od 1. ledna 1970. Lze použít pro měření reálného času.

Priorita procesu

nice(2) . . . . Změna priority procesu

```
#include <unistd.h>
int nice(int inc);
```

Přičte `inc` k prioritě volajícího procesu. Pouze superuživatel může uvést negativní inkrement.

sched_yield(2) . . . . Kooperativní multitasking

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

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

getpriority(2) . . . . Čtení priority procesu

```
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, int who);
int setpriority(int which, int who, int pri);
```

Služba `getpriority(2)` čte prioritu procesu. Hodnota parametru `which` je jedna z následujících:

PRIO_PROCESS – priorita procesu.
PRIO_PGRP – priorita skupiny procesů.
PRIO_USER – priorita procesů daného uživatele.

Nulová hodnota parametru `who` značí volající proces, skupinu procesů nebo uživatele.

Služba `setpriority(2)` nastavuje prioritu procesu/skupiny procesů/uživatelových procesů na `pri`.

Službu `setpriority(2)` používá například program `renice(1)`.

I/O operace

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

open(2), creat(2) . . . . Otevření souboru

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(char *path, int flags);
int open(char *path, int flags, mode_t mode);
int creat(char *path, mode_t mode);
```

Vrací deskriptor, příslušný souboru `path`. Parametr `flags` je jedno z `O_RDONLY`, `O_WRONLY` nebo `O_RDWR`, plus logický součet některých z konstant:

O_CREAT – vytvoření souboru, pokud neexistuje.
O_EXCL – chyba, pokud soubor existuje.
O_TRUNC – zarovnání souboru na nulovou délku.
O_APPEND – před každým zápisem do souboru je ukazatel pozice v souboru nastaven na konec souboru (jako u `lseek(2)`).
O_NONBLOCK, O_NDELAY – otevření v neblokujícím režimu.
O_SYNC – synchronní výstup.

close(2) . . . . Uzavření deskriptoru

```
#include <unistd.h>
int close(int fd);
```

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

lseek(2) . . . . Nastavení pozice v souboru

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int odkud);
```

Nastaví ukazatel aktuální pozice v souboru. Parametr `odkud` nabývá těchto hodnot:

SEEK_SET – offset od začátku souboru.
SEEK_CUR – offset od aktuální pozice souboru.
SEEK_END – offset od konce souboru.

V některých systémech existuje i služba `llseek(2)`, která má parametr `offset` typu `long long`. Slouží pro přístup k souborům větším než `1«31` bajtů.
Na některé typy souborů nelze použít `lseek()`.

read(2) . . . . Čtení souboru

```
#include <unistd.h>
ssize_t read(int fd,void *buf,size_t count);
```

Načte nejvýše `count` bajtů ze souboru do bufferu `buf`. Vrátí –1 v případě chyby, 0 na konci souboru nebo počet načtených bajtů.

write(2) . . . . Zápis do souboru

```
#include <unistd.h>
ssize_t write(int fd,void *buf,size_t count);
```

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

◇ **Příklad:**

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
...
#define BUFFER     (1«14)
char *name1, *name2, *p, buffer[BUFFER];
int fd1, fd2, l1, l2;
...
if ((fd1=open(name1,O_RDONLY)) == -1) {
     perror("Opening input file");
     exit(1);
}
```

```
if ((fd2=open(name2,O_WRONLY|O_CREAT, 0777))==-1){
     perror("Opening output file");
     exit(2);
}
while((l1=read(fd1,buffer,BUFFER))>0) {
     for(p=buffer; (l2=write(fd2,p,l1))>0; p+=l2)
          if(!(l1-=l2))
               break;
     if (l2 <= 0) {
          perror("Writing output file");
          exit(3);
     }
}
if (l1 < 0) {
     perror ("Reading input file");
     exit(4);
}
close(fd1);
close(fd2);
```


Zápis na konec souboru

Bit O_APPEND při otevírání souboru nelze v UN*Xu dobře emulovat. Viz následující dva úseky kódu:

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



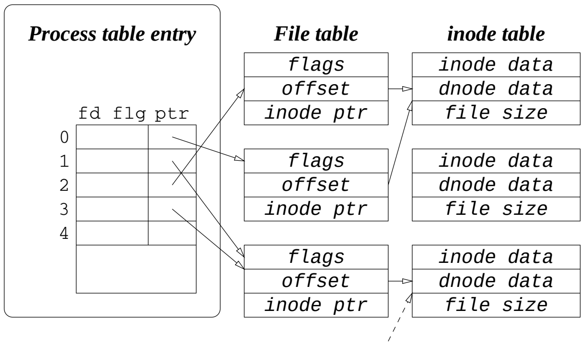
```
if ((fd=open(filename,O_WRONLY|O_APPEND)) == -1) {
      perror("open");
      exit(1);
}
if (write(fd, buffer, size) == -1) {
      perror("write");
      exit(3);
}
```

Problém nastává v případě zápisu více procesy do téhož souboru – pokud v prvním případě dojde k přepnutí kontextu mezi voláním lseek(2) a write(2), nemusí dojít k zápisu na skutečný konec souboru.

◇ **Úkol:**

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

Tabulka otevřených souborů



dup(2),dup2(2) . . . . Duplikace deskriptoru

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Duplikuje deskriptor – vytvoří nový odkaz do tabulky otevřených souborů na strukturu file. Použití: přesměrování v shellu.

◇ **Úkol:** Jak se liší funkce následujících dvou úseků kódu?
Varianta 1:

```
fd1=open("file",O_WRONLY|O_CREAT,0777);
fd2=dup(fd1);
write(fd1,"Hello, world\n",13);
write(fd2,"Hello, world\n",13);
close(fd1); close(fd2);
```

Varianta 2:

```
fd1=open("file",O_WRONLY|O_CREAT,0777);
fd2=open("file",O_WRONLY|O_CREAT,0777);
write(fd1,"Hello, world\n",13);
write(fd2,"Hello, world\n",13);
close(fd1); close(fd2);
```


fcntl(2) . . . . Změna vlastností deskriptoru

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

Služba fcntl(2) provádí různé akce nad otevřeným deskriptorem. Hodnota cmd může být následující:

F_DUPFD – duplikuje deskriptor fd do arg, podobně jako dup2(2).
F_GETFD – čte flagy deskriptoru (momentálně pouze FD_CLOEXEC).
F_SETFD – nastavuje flagy deskriptoru (FD_CLOEXEC).
F_GETFL – čte flagy struktury file. Tyto odpovídají druhému parametru volání open(2), kterým byla tato struktura vtvořena, nebo předchozímu fcntl(,F_SETFL,).
F_SETFL – nastavuje flagy struktury file. Lze nastavovat pouze O_APPEND, O_NONBLOCK, O_ASYNC a O_SYNC.
F_GETLK, F_SETLK
– zamykání části souboru.

ioctl(2) . . . . Práce s I/O zařízením

```
#include <unistd.h> /* SVR4 */
#include <sys/ioctl.h> /* SVR4 */
int ioctl(int fd, int cmd, long arg);
```

Tato služba slouží k nastavení I/O zařízení, ke čtení jeho stavu a k posílání příkazů do zařízení. Není v POSIX.1.

◇ **Příklad:** Nastavení signálu DTR na sériové lince na log. 1:

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


Organizace souborových systémů

Systém souborů musí zajišťovat:

- **Efektivní přístup k souborům** – adresářové operace (vyhledání souboru, přejmenování, atd.).
- **Efektivní operace nad soubory** – čtení/zápis (malá fragmentace etc.)
- **Spolehlivé zotavení po havárii**
- **Co nejmenší prostor na režii** – velikost metadat.

Svazek (systém souborů) je reprezentován blokovým zařízením. Většinou jde o diskovou oblast.

- **Boot block** je první blok svazku. Zavádí se z něj operační systém, nebo je prázdný.
- **Super block** – další blok svazku. Obsahuje sumární informace o svazku.
- **Tabulka i-uzlů** – informace o souborech.
- **Datové bloky**

i-uzly

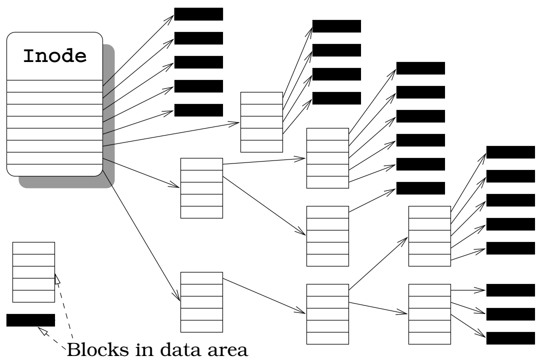
i-uzel (identifikační uzel, inode) je struktura na disku, která popisuje soubor.

Následující atributy i-uzlů je možno zjišťovat na příkazové řádce příkazem `ls`, v programu voláním jádra `stat()`:

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

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

i-uzel a datové bloky



- **Výhoda** – přímý přístup ke kterémukoli místu souboru.
- **Díry v souborech** – `/var/log/lastlog`, `core`.

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

Práce se soubory

stat(2) . . . . Informace o i-uzlu

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(char *path, struct stat *st);
int lstat(char *path, struct stat *st);
int fstat(int fd, struct stat *st);
```

Tyto služby systému zjišťují informace, uložené v diskovém i-uzlu. Příkaz `ls -l` používá služby jádra `lstat(2)`.

Služba `lstat(2)` není součástí standardu POSIX.1, ale stane se pravděpodobně součástí standardu POSIX.1a.

Struktura `struct stat` má tyto položky:

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

Přístupová práva

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

- **`S_ISREG()`** – běžný soubor.
- **`S_ISDIR()`** – adresář.
- **`S_ISCHR()`** – znakový speciální soubor.
- **`S_ISBLK()`** – blokový speciální soubor.
- **`S_ISFIFO()`** – roura nebo pojmenovaná roura.
- **`S_ISLNK()`** – symbolický link (není v POSIX.1).
- **`S_ISSOCK()`** – pojmenovaný socket (není v POSIX.1).

Přístupová práva lze z `st_mode` získat těmito maskami:

`S_ISUID, S_ISGID, S_ISVTX`
– set-uid bit, set-gid bit a sticky bit.

`S_IRUSR, S_IWUSR, S_IXUSR`
– práva vlastníka souboru.

`S_IRGRP, S_IWGRP, S_IXGRP`
– práva skupiny.

`S_IROTH, S_IWOTH, S_IXOTH`
– práva ostatního světa.

`access(2)` . . . . Ověření přístupových práv

```
#include <unistd.h>
int access(char *path, int mode);
```

Služba ověří, jestli vlastník procesu (to jest reálné UID/GID) má přístup k souboru. Parametr `mode` určuje typ přístupu – maska z jedné nebo více hodnot z `F_OK`, `R_OK`, `W_OK` a `X_OK`.

POZOR: Nebezpečí změny práv mezi voláním `access(2)` a skutečným přístupem. Potenciální bezpečnostní problém.

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

Nově vytvářené soubory

Vlastník souboru, který vznikne pomocí `open(2)` nebo `creat(2)` je nastaven podle efektivního UID procesu, který tento soubor vytvořil.

U skupiny souboru připouští POSIX.1 jednu ze dvou možností:

- Skupina je přidělena podle efektivního GID procesu, který soubor vytvořil.
- Skupina je přidělena podle GID adresáře, ve kterém je soubor vytvářen.

První varianta je v SVR4, druhá v BSD systémech (a vyžaduje ji FIPS 151-1). V SVR4 lze druhé varianty dosáhnout přidáním set-gid bitu do přístupových práv adresáře.

`umask(2)` . . . . Maska přístupových práv

```
#include <sys/stat.h>
int umask(int newmask);
```

Služba nastavuje masku přísupových práv pro nově vytvářené soubory. Vrací předchozí nastavení této masky. Bity, které jsou v masce nastaveny na 1, se u nově vytvářeného souboru nulují.

⋄ **Příklad:** Je-li `umask` rovno `022` a třetí parametr `open(2)` je roven `0776`, má výsledný soubor práva `0776 & ~022 = 0754`.

Změna práv souboru

V některých systémech (4.3 BSD) může vlastníka souboru měnit jen superuživatel (hlavním důvodem jsou diskové kvóty).

V POSIX.1 je toto volitelné – v době kompilace podle makra `_POSIX_CHOWN_RESTRICTED`, nebo v době běhu pomocí funkce `fpathconf(3)`, resp. `pathconf(3)`.

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

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

Některé systémy (BSD 4.4 a z něj odvozené) nulují set-uid a set-gid bity v okamžiku zápisu do souboru procesem, který nemá práva superuživatele. Vždy jsou tyto bity nulovány také při změně vlastníka nebo skupiny souboru.

`chmod(2)` . . . . Změna přístupových práv

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Nastaví přístupová práva na soubor. Služba `fchmod(2)` není součástí POSIX.1, ale BSD i SVR4 systémy ji podporují.

`chown(2)` . . . . Změna vlastníka/skupiny souboru

```
#include <sys/types.h>
#include <unistd.h>
int chown(char *path,uid_t owner,gid_t grp);
int lchown(char *path,uid_t owner,gid_t grp);
int fchown(int fd,uid_t owner,gid_t grp);
```

Změní vlastníka a skupinu souboru. Je-li jeden z parametrů `owner` nebo `grp` roven -1, neprovádí se změna tohoto údaje.

`fchown(2)` není součástí POSIX.1. Podporováno SVR4 i tak i 4.3+ BSD.

`lchown(2)` je pouze v SVR4. V ostatních systémech mění `chown(2)` práva symbolického linku a ne souboru, na který tento link ukazuje (bezpečnost!).

`truncate(2)` . . . . Nastavení velikosti souboru

```
#include <sys/types.h>
#include <unistd.h>
int truncate(char *path, off_t length);
int ftruncate(int fd, off_t length);
```

Nastaví velikost souboru na `length`. Některé systémy (například 4.3 BSD a novější) nedovolí zvětšit velikost souboru. Tyto služby nejsou v POSIX.1, ale SVR4 i BSD je podporují.

SVR4 implementuje navíc `fcntl(F_FREESP)` – vytvoření díry v již existujícím souboru.

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

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

link(2) . . . . Vytvoření odkazu na i-uzel

```
#include <unistd.h>
int link(char *path, char *newpath);
```

Vytvoří další odkaz na i-uzel (tzv. *pevný link*). POSIX.1 specifikuje, že `link(2)` může skončit s chybou, pokud `path` a `newpath` nejsou na tomtéž svazku.

unlink(2) . . . . Zrušení odkazu na i-uzel

```
#include <unistd.h>
int unlink(char *path);
```

Zruší odkaz na i-uzel. Pokud je počet odkazů na i-uzel nulový, systém i-uzel smaže a uvolní příslušné datové bloky. Pozor: Za odkaz se považuje také odkaz z tabulky otevřených souborů.

⋄ **Příklad:** Vytvoření anonymního dočasného souboru:

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


remove(3) . . . . Zrušení souboru/adresáře

```
#include <stdio.h>
int remove(char *path);
```

Smaže soubor nebo adresář. Tato funkce je součástí normy ANSI C.

rename(2) . . . . Přejmenování souboru/adresáře

```
#include <unistd.h>
int rename(char *oldpath, char *newpath);
```

Funkce je definována v POSIX.1 i v ANSI C (zde jen pro soubory). Atomické přejmenování/přesunutí souboru v rámci jednoho svazku.

utime(2) . . . . Nastavení časů souboru

```
#include <sys/types.h>
#include <utime.h>
int utime(char *path, struct utimbuf *times);
struct utimbuf {
    time_t actime;
    time_t modtime;
}
```

Nastavení času posledního přístupu/modifikace. Používá například program `touch(1)`. Pokud je druhý parametr nulový ukazatel, jsou oba časy nastaveny na současný systémový čas.

Nastavovat čas smí pouze vlastník souboru (nebo superuživatel). Právo zápisu k tomuto nestačí.

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

Symbolické linky

- Symbolický odkaz na soubor pomocí cesty.
- Relativní versus absolutní symbolické linky.
- Nejsou v POSIX.1 (ale jsou součástí POSIX.1a).

symlink(2) . . . . Vytvoření symbolického linku

```
#include <unistd.h>
int symlink(char *sympath, char *path);
```

Vytvoří symbolický link `path`, obsahující řetězec `sympath`.

readlink(2) . . . . Čtení symbolického linku

```
#include <unistd.h>
int readlink(char *path,char *buf,size_t sz);
```

Přečte obsah symbolického linku (provádí ekvivalent služeb jádra `open(2)`, `read(2)` a `close(2)`). Obsah bufferu není ukončen nulovým znakem.

Symbolické linky a přístup k souborům

Služby jádra, které neprocházejí symbolické linky: `chown(2)` (pokud v systému neexistuje `lchown(2)`), `lchown(2)`, `lstat(2)`, `readlink(2)`, `rename(2)` a `unlink(2)`.

⋄ **Úkol:** Co bude výsledkem těchto tří příkazů na různých systémech?

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


Vytváření dočasných souborů

- **Adresář `/tmp`, `/var/tmp`**
- **Sticky bit**
- **Exkluzivita**
- **Bezpečnostní problém se symbolickými linky**
- **Linux** – `O_CREAT|O_EXCL`
- **FreeBSD** – `O_NOFOLLOW`
- **Ze shellu** – `mktemp(1)`.

mkstemp(3) . . . . Vytvoření dočasného souboru

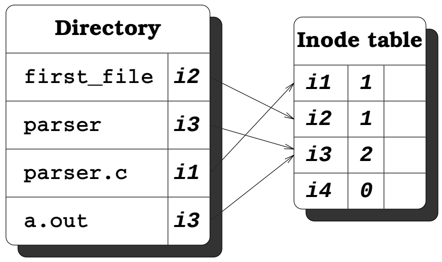
```
#include <stdlib.h>
int mkstemp(char *template);
int mkstemp("/tmp/mail.XXXXXX");
```

Vytvoří dočasný soubor podle dané masky. Vrátí deskriptor na tento soubor, do parametru zapíše skutečné jméno.

Adresáře v UNIXu

- **Adresář** – soubor, obsahující záznamy tvaru (*název, i-uzel*).
- **Položka „`.`“** – odkaz na sebe.
- **Položka „`..`“** – odkaz na nadřazený adresář; v kořenovém adresáři ukazuje na sebe.
- **Implementace** – položky „`.`“ a „`..`“ jsou často implementovány na úrovni OS, nikoli nutně fyzicky na disku.
- **Soubor pod více jmény** – ne adresáře (nejasný význam „`..`“ v adresáři).
- **Délka jména** – záleží na FS. Původní UNIX - 14, dnes většinou aspoň 252.
- **Délka struktury** – pevná nebo proměnná.
- **Organizace adresáře** – seznam, pole, strom.
- Každý adresář má aspoň dva odkazy.
- Sémantika konstrukce „`//`“.

Struktura adresáře



`mkdir(2)` . . . . Vytvoření adresáře

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(char *path, mode_t mode);
```

Vytvoří nový prázdný adresář s právy `mode` (modifikovanými podle `umask(2)`).

`rmdir(2)` . . . . Smazání adresáře

```
#include <unistd.h>
int rmdir(char *path);
```

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

Čtení adresáře

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

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(char *path);
struct dirent *readdir(DIR *dp);
void rewinddir(DIR *dp);
int closedir(DIR *dp);
struct dirent {
    ino_t d_ino;
    char d_name[NAME_MAX+1];
}
```

POSIX.1 definuje pouze položku `d_name`. Pořadí jmen souborů (struktur `dirent`) vrácených při čtení adresáře závisí na implementaci.

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

Adresáře procesu

`getcwd(2)` . . . . Jméno pracovního adresáře

```
#include <unistd.h>
char *getcwd(char *buf, size_t sz);
```

Vrátí cestu k pracovnímu adresáři. Je-li `sz` příliš malé, skončí s chybou. (Pozor na rozdíl mezi `pwd` a `/bin/pwd`).

`chdir(2)` . . . . Změna pracovního adresáře

```
#include <unistd.h>
int chdir(char *path);
int fchdir(int fd);
```

Změní pracovní adresář na zadaný adresář (kontrola přístupových práv). Služba `fchdir(2)` není v POSIX.1 – jde o BSD rozšíření. (Proč neexistuje `cd(1)`?)

`chroot(2)` . . . . Změna kořenového adresáře procesu

```
#include <unistd,h>
int chroot(char *path);
```

Změní kořenový adresář procesu. Povoleno pouze superuživateli.

⋄ **Úkol:** Co všechno je potřeba k tomu, aby proces mohl „uniknout“ z prostředí se změněným kořenovým adresářem?

`sync(2)` . . . . Synchronizování disků

```
#include <unistd.h>
void sync(void);
```

Synchronizování diskových bufferů. Tyto operace zařadí buffery které se mají ukládat na disk do fronty pro okamžitý zápis a nastartují tento zápis (obvykle speciální thread jádra).

`fsync(2), fdatasync(2)` . . . . . . Synchronizace deskriptoru

```
#include <unistd.h>
int fdatasync(int fd);
int fsync(int fd);
```

Zapíše všechny modifikované části souboru na disk. Služba `fdatasync(2)` nezapisuje metadata souboru (čas modifikace, ...).

mknod(2) . . . . Vytvoření souboru

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(char *path,mode_t mode,dev_t dev);
```

Vytvoří soubor daného jména. Parametr `mode` specifikuje přístupová práva a typ souboru (jedna z konstant `S_IFREG`, `S_IFCHR`, `S_IFBLK` nebo `S_IFIFO`, viz `stat(2)`).

mkfifo(2) . . . . Vytvoření pojmenované roury

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(char *path, mode_t mode);
```

Vytvoří pojmenovanou rouru. Ve starších BSD systémech nebylo volání `mknod(2)`; pojmenovaná roura se vytvářela touto službou jádra.

Implementace souborových systémů

Zotavení po havárii

- **Možné nekonzistence** – pořadí zápisových operací, write-back cache, změny dat/metadat, ale i chyby HW nebo OS.
- **Kontrola konzistence** `fsck(8)`. Časově náročné.
- **Synchronní zápis metadat?** – problémy se starými daty v souborech (bezpečnost!).
- **BSD Soft-updates** – závislosti mezi diskovými operacemi. Omezení počtu typů nekonzistencí ⟶ rychlejší `fsck(8)`. Ale: problém pořadí data versus metadata; neřeší se chyba OS nebo HW.
- **Žurnálování** – transakční přístup. Změny nejprve zapsány do logu (žurnálu) a pak provedeny. Po havárii – přehrání celých transakcí. Někdy i rychlejší než nežurnálovaný FS. Většinou o něco pomalejší. Žurnál jen metadat nebo i dat. Chyba OS nebo HW se řeší pomocí `fsck(8)`.

Access Control Lists

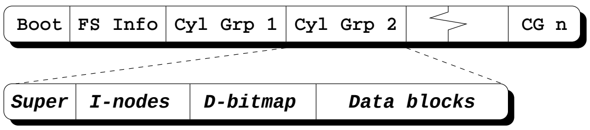
- **Řízení přístupu pomocí GID** – dostatečně silné, ale vyžaduje spoluúčast superuživatele.
- **ACL** – plné řízení přístupu vlastníkem souboru.
- **ACL** – seznam položek tvaru ⟨*typ*⟩**:**[⟨*hodnota*⟩]**:**[r][w][x]
- **Implicitní položky** – typ `u`, `g`, `o` s prázdnnou hodnotou.
- **Další položky** – typ `u` a `g` s neprázdnou hodnotou. Je-li aspoň jedna takováto položka, je povinná další položka typu `m` – maska.
- **Příklady** – `u::rwx,g::r-x,o::r--`
  `u::rwx,g::r-x,o:---,u:bob:rwx,g:wheel:rw-,m:r-x`
- **Vyhodnocování** – hledá se shoda efektivního UID procesu, pokud se nenalezne, tak efektivní GID a doplňková GID, pokud se ani tady nenalezne, použije se položka `o:`. U nepovinných položek log. součin s maskou.
- **Omezení** – právě jedna položka od typu `u::`, `g::`, `o::`. Nejvýše jedna položka `m::`. Nejvýše jeden záznam pro každého uživatele a skupinu.
- **Korespondence s UNIXovými právy** – práva vlastníka souboru = položka `u::`, práva skupiny souboru = položka `m::`; není-li, pak `g::`.
- **Implicitní ACL** – u adresářů. Použije se pro nově vytvářené soubory.
- **Programy** – `getfacl(1)`, `setfacl(1)`, `chacl(1)`. Též `acl(5)`.

FAT

- Nemá i-uzly (nelze mít soubor ve více adresářích, nemá UNIXová přístupová práva).
- Pomalý přímý přístup k souboru (sekvenční procházení přes FAT).
- Fragmentace už při současném zápisu do dvou souborů.
- Fragmentace při rušení souboru.
- Na větších FS velká délka bloku ⟶ špatné využití místa.
- Výhody – na menších FS malá režie, jednoduchá implementace.

UFS

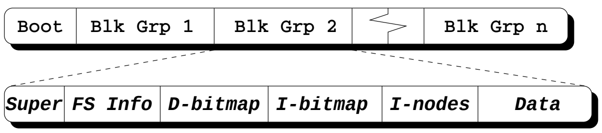
- **FFS, EFS, UFS** – původně v 4.x BSD.
- Cylinder groups. Nutná znalost geometrie disku.
- Snížení fragmentace, 4–8 KB bloky.
- Fragmenty – lepší využití místa na disku.
- Kopie superbloku.
- Rezervované místo pro superuživatele
- Synchronní zápis metadat; novější implementace (FreeBSD) umí i asynchronní.
- Soft updates – zachování pořadí (některých) změn v datech a metadatech. Jednodušší `fsck(8)` – pevně dané typy nekonzistencí.
- Kontrola disku na pozadí.



- **Implementace** – *BSD, Solaris (+ žurnálování), Linux.

LINUX ext2 filesystem

- Skupiny bloků (block groups) – místo CG u FFS. Není nutná znalost geometrie disku ⟶ jednodušší implementace, využití celých bloků.
- Obvykle 1 KB (až 4 KB) bloky – rychlejší než FFS s 4 KB bloky.
- Alokační strategie: Předalokované bloky, alokace dat poblíž příslušných metadat, zamezení zaplnění jedné skupiny bloků.
- Bitmapa volných i-uzlů.
- Téměř neexistuje fragmentace.
- Priorita zápisu metadat (ale data se zapisují zároveň).
- Asynchronní zápis metadat; na požádání umí i synchronní.
- Velikost až do 4 TB dat. Velká odolnost proti havárii.
- Rychlé symbolické linky.
- No-atime volba.
- Maximum mount count. `tune2fs(8)`.
- Možnosti při chybě – panic, remount r-only, ignore.
- `libe2fs` – knihovna pro přístup k e2fs. `e2defrag`.



LINUX ext3 filesystem

- Struktury na disku – zpětně kompatibilní s ext2.
- **Žurnálování** – změny zapisovány přes transakční log.
- **Žurnálování dat** – `journal`, `ordered`, `writeback`.
- **Rozšířené atributy** – další metadata (např. security context).
- **Access control lists** – rozšíření přístupových práv (viz dále).
- **Adresáře** – lineární struktura nebo strom.

LINUX ReiserFS, Reiser4

- **Všechna data v jednom B+ stromu** (R4 – „dancing tree").
- **Alokace místa** – i menší kousky než jeden sektor.
- **I-uzly** – alokace podle potřeby.
- Efektivní i při velkém množství souborů v adresáři nebo velkém množství malých souborů.

Navíc v Reiser4:

- **Plug-iny** souborového systému (např. vyhledávání/indexace).
- **Soubory s více proudy dat** (např. metadata) – každý soubor je také adresář.
- **Transakce** – více datových operací může být spojeno do jedné atomické transakce.

SGI XFS

- **Plně 64-bitový**
- **Rozdělení svazku** – kousky velikosti 0.5 až 4 GB.
- **Organizace dat** – B+ strom
- **DMAPI** – data manipulation API – zpřístupnění vlastností B-stromu (vkládání/rušení dat uprostřed souboru).
- **ACL** – viz dále.
- **Žurnálování**
- **Real-time extenze** – možnost alokace šířky pásma; garantovaná propustnost.
- **O_DIRECT** – přístup bez cachování.
- **Allocate on flush** – další snížení fragmentace.
- **CXFS** – nadstavba pro clustery (za příplatek).
- **Implementace** – IRIX, Linux.

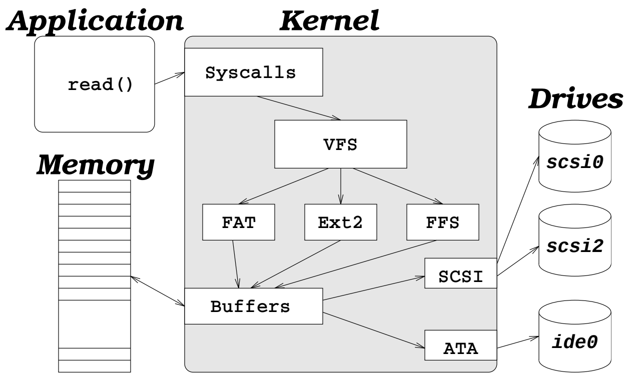
Další služby FS

- **Synchronní/asynchronní zápis dat nebo metadat** – často volitelné.
- **Komprese dat** – celý FS nebo jen určité soubory.
- **Fragmenty** – FFS/UFS.
- **Obnova smazaných souborů**.
- **Bezpečné mazání souborů** – LINUX ext[23]fs.
- **Nepřemistitelné soubory** – LINUX ext[23]fs.
- **Soubory, umožňující pouze přidávat data** – append-only.
- **Změna velikosti svazku za běhu** – AIX jfs, Tru64 advfs, ...
- **Kontrolní součty dat** – Sun/Solaris ZFS.

Virtual file system

VFS je vrstva jádra, zajišťující stejný přístup zbytku jádra k různým souborovým systémům. Tato vrstva leží mezi vstupním bodem jádra (obsluha systémových volání) a jednotlivými implementacemi FS.

Původní idea VFS pochází od Sun Microsystems (z důvodů implementace NFS). Dnes je VFS vrstva ve většině systémů.



Správa logických svazků

- **Logical Volume Manager (lvm)**
- **Spojení více fyzických zařízení do jednoho**

Struktura

- **Physical volume (pv)** – disk, disková oblast. Skládá se z
- **Physical extent (pe)** – část diskové oblasti, pevná délka (např. 4 MB).
- **Volume group (vg)** – obsahuje několik PV, jejichž PE jsou v ní zpřístupněny jako
- **Logical extent (le)** – odpovídá příslušnému PE.
- **Logical volume (lv)** – odpovídá blokovému zařízení. Skládá se z několika LE v rámci jedné VG. Na LV se vytvoří souborový systém a používá se.

Výhody LVM

- **Změna velikosti VG** – přidání/odebrání několika PV.
- **Změna velikosti LV** – přidání/odebrání několika LE. Musí navazovat změna velikosti souborového systému.
- **Odebrání PV** – transparentní.
- **Klon LV** – atomický snímek, nezabírá mnoho místa, copy-on-write.

Komunikace mezi procesy

Nepojmenovaná roura

- Zasílání dat mezi příbuznými procesy.
- Implementace – kruhový buffer velikosti `PIPE_BUF`.

pipe(2) . . . . Vytvoření roury

```
#include <unistd.h>
int pipe(int fd[2]);
```

Vrátí dva deskriptory – `fd[0]` pro čtení a `fd[1]` pro zápis. Využití: Proces otevře rouru pomocí `pipe(2)` a pak vytvoří potomka pomocí `fork(2)`. Roura je využívána ke komunikaci mezi rodičem a potomkem.

Pojmenovaná roura

- **Vznik** – službou jádra `mknod(2)`.
- **Otevření** – služba `open(2)` s příslušnou cestou.
- **Vlastnosti** – stejné jako u nepojmenované roury.
- Pomocí pojmenované roury mohou spolu komunikovat i nesouvisející procesy.

Vlastnosti roury

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

◇ **Příklad:** Použití roury:

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



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


Signály

- Asynchronní událost.
- Akce při vzniku signálu – ignorovat, zachytit ovladačem (*handler*), implicitní akce.

`signal(2)` . . . . Nastavení reakce na signál

```
#include <signal.h>
void (*signal(int sig, void (*hndlr)(int)))(int);
```

nebo také jinak:

```
typedef void SigHandler(int);
SigHandler *signal(int sig, SigHandler *hndlr);
```

Nainstaluje ovladač signálu, vrátí jeho předešlou hodnotu. Hodnoty parametru `hndlr`:

`SIG_IGN` – ignorovat signál.
`SIG_DFL` – nastavit implicitní hodnotu.
`funkce` – nastaví se jako ovladač signálu. V okamžiku vyvolání signálu dostane funkce jako parametr číslo signálu.

`kill(2), raise(2)` . . . . Zaslání signálu

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

Činnost služby `kill(2)` závisí na hodnotě `pid` takto:

`pid > 0` Signál je zaslán procesu s číslem `pid`.
`pid == 0` Signál je zaslán procesům ze stejné skupiny, jako je volající proces.
`pid < 0` Signál je zaslán procesům ze skupiny, jejíž číslo je rovno absolutní hodnotě `pid`.
`pid == -1` POSIX.1 zde ponechává nespecifikovaný výsledek.

Pokud je číslo signálu 0, vrátí `kill(2)` hodnotu -1 a `errno` bude mít hodnotu `ESRCH`, pokud proces daného `pid` neexistuje. Pokud existuje, je vrácena nula nebo `errno` nastaveno na `EPERM`.

Funkce `raise(3)` je součástí ANSI C – v UNIXu způsobí zaslání signálu stejnému procesu.

Tabulka signálů

`A` – ANSI C
`P` – POSIX.1
`J` – POSIX.1, systém podporuje job control
`S` – System V Release 4
`B` – 4.3BSD

Jméno	Popis	Norma	Akce
SIGABRT	Abnormální ukončení	APSB	core
SIGALRM	Časovač	PSB	ukončení
SIGBUS	Hardwarová chyba	SB	core
SIGCHLD	Změna stavu potomka	JSB	ignorování
SIGCONT	Pokračování po STOP	JSB	znovuspuštění
SIGEMT	Hardwarová chyba	SB	core
SIGFPE	Chyba reálné aritmetiky	APSB	core
SIGHUP	Zavěšení linky	PSB	ukončení
SIGILL	Neplatná instrukce	APSB	core
SIGINFO	Získání stavu z terminálu	B	ignorování
SIGINT	Přerušení z terminálu	APSB	ukončení
SIGIO	Asynchronní I/O	SB	core
SIGIOT	Hardwarová chyba	SB	core

SIGKILL	Ukončení procesu	PSB	ukončení
SIGPIPE	Rouru nikdo nečte	PSB	ukončení
SIGPOLL	Sledovatelná událost	S	ukončení
SIGPROF	Profilovací časovač	SB	ukončení
SIGPWR	Výpadek napájení	S	ignorování
SIGQUIT	Znak `Quit` na terminálu	PSB	core
SIGSEGV	Chyba segmentace	APSB	core
SIGSTOP	Pozastavení procesu	JSB	pozastavení
SIGSYS	Neplatná služba jádra	SB	core
SIGTERM	Výzva k ukončení	APSB	ukončení
SIGTRAP	Hardwarová chyba	SB	core
SIGTSTP	Znak `Stop` na terminálu	JSB	pozastavení
SIGTTIN	Pokus o čtení z terminálu	JSB	pozastavení
SIGTTOU	Pokus o zápis na terminál	JSB	pozastavení
SIGURG	Urgentní událost	SB	ignorování
SIGUSR1	Uživatelský signál 1	PSB	ukončení
SIGUSR2	Uživatelský signál 2	PSB	ukončení
SIGVTALRM	Virtuální časovač	SB	ukončení
SIGWINCH	Změna velikosti okna	SB	ignorování
SIGXCPU	Překročení strojového času	SB	core
SIGXFSZ	Překročení velikosti souboru	SB	core

pause(2) . . . . Čekání na signál

```
#include <unistd.h>
int pause();
```

Čeká na příchod signálu, který není ignorován. Služba vrací vždy hodnotu -1.

◇ **Úkol:** Zjistěte, jakou hodnotu `errno` nastavuje služba jádra `pause(2)`.

Vlastnosti signálů

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

Spolehlivé signály

- **Nová sémantika**
- **Vygenerování signálu** – v okamžiku volání `kill(2)`, vypršení časovače, chybě segmentace a podobně.
- **Doručení signálu** (*delivery*) – okamžik, kdy se vykoná reakce na signál podle stavu procesu.
- **Čekající signál** (*pending*) – stav signálu mezi vygenerováním a doručením.
- **Blokování signálu** – proces má možnost dočasně odložit doručení signálu. Zablokovaný signál zůstává ve stavu *pending* dokud proces nezruší blokování nebo dokud nenastaví reakci na signál na ignorování signálu.
- **Signál vygenerován vícekrát** – v původním rozhraní se mohl doručit jednou nebo vícekrát. Novější systémy mohou mít více čekajících signálů stejného čísla. Signály *čekají ve frontě* (*queued signals*).
- **Restartování služeb jádra** – systém umí po příchodu signálu restartovat některé služby jádra místo toho, aby vrátily chybu a `EINTR`.

Množiny signálů

- **Množina signálů** – nový datový typ. Slouží ke změně reakcí na více signálů jednou (atomickou) službou jádra.
- **Operace nad množinou** – funkce/makra `sigemptyset(3)`, `sigfillset(3)`, `sigaddset(3)`, `sigdelset(3)` a `sigismember(3)`.

sigsetops(3) . . . . Operace nad množinou signálů

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(sigset_t *set, int signo);
```


sigprocmask(2) . . . . Blokování signálů

```
#include <signal.h>
int sigprocmask(int how, sigset_t *set,
    sigset_t *old);
```

Je-li `old` nenulový ukazatel, uloží se sem původní množina blokovaných signálů. Je-li `set` nenulový ukazatel, změní se maska blokovaných signálů podle hodnoty parametru `how`:

SIG_BLOCK – nová množina je sjednocením původní množiny a `set`.

SIG_UNBLOCK – nová množina je průnikem původní množiny a doplňku `set`.

SIG_SETMASK – nová množina je rovna `set`.

Čeká-li na proces nějaký signál, který je odblokovaný voláním `sigprocmask(2)`, je aspoň jeden takový signál doručen před návratem ze `sigprocmask(2)`.

sigpending(2) . . . . Dotaz na čekající signály

```
#include <signal.h>
int sigpending(sigset_t *set);
```

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

sigsuspend(2) . . . . Čekání na signál

```
#include <signal.h>
int sigsuspend(sigset_t *set);
```

Dočasně nahradí masku blokovaných signálů za `set` a zablokuje proces, dokud jeden z těchto signálů nepřijde.

sigaction(2) . . . . Změna reakce na signál

```
#include <signal.h>
int sigaction(int signum, struct sigaction
    *act, struct sigaction *old);
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
```

Je-li `act` nenulový ukazatel, změní podle něj reakci na signál. Původní reakce je uložena do `old`, je-li toto nenulový ukazatel.

sa_handler specifikuje reakci na signál. Může být `SIG_IGN` pro ignorování signálu, `SIG_DFL` pro implicitní hodnotu nebo ukazatel na funkci pro obsluhu signálu.

sa_mask specifikuje masku signálů, které mají být zablokovány během provádění obslužné funkce `sa_handler` pro daný signál. Navíc signál sám je blokován během provádění svého ovladače, pokud není nastaven flag `SA_NODEFER` nebo `SA_NOMASK`.

sa_flags dále nastavuje vlastnosti reakce na signál. Jeho hodnota je log. součet některých z následujících konstant:

SA_NOCLDSTOP – pokud je `signum` rovno `SIGCHLD`, proces nedostane tento signál při pozastavení potomka, ale jen při jeho ukončení.

SA_ONESHOT nebo **SA_RESETHAND**
– po prvním vyvolání ovladače nastaví reakci zpět na `SIG_DFL` (stejně se chová ovladač instalovaný pomocí `signal(2)`).

SA_ONSTACK – byl-li nastaven alternativní zásobník pro provádění signálů (pomocí `sigaltstack(2)`), použije se pro tento ovladač tento alternativní zásobník.

SA_NOCLDWAIT – je-li `signum` rovno `SIGCHLD`, proces nečeká na potomky a potomci nevytvářejí zombie.

SA_NODEFER nebo **SA_NOMASK**
– během provádění ovladače není zablokováno doručení stejného signálu.

SA_RESTART – restartuj případnou přerušitelnou službu jádra, je-li přerušena tímto signálem (namísto chyby `EINTR`).

I/O multiplexing

Čtení z jedoho deskriptoru a zápis na druhý:

```
while ((n=read(0, buf, bufsiz))>0)
    if (write(1, buf, n) < 0)
        error_message("write to stdout");
```

- Blokující I/O operace.
- Co když chceme číst ze dvou deskriptorů zároveň?
- Program pro komunikaci s modemem.
- Polling.
- Asynchronní I/O.
- Selektivní čekání.

select – BSD interface.
poll – SysV interface.

select(2) . . . . Selektivní čekání

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd,h>
struct timeval {
    long tv_sec; /* sekundy */
    long tv_usec; /* mikrosekundy */
};
FD_CLR(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
int select(int n, fd_set *rdset, fd_set *wrset,
     fd_set *exset, struct timeval *timeout);
```

Čekání na připravenost ke čtení, připravenost k zápisu, případně výjimku na deskriptoru. Parametr `n` je 1 + nejvyšší číslo deskriptoru, nastavené v kterékoli ze tří množin.

Vrací počet deskriptorů, na kterých nastala požadovaná situace. Do příslušných množin nastaví deskriptory, připravené k I/O operaci.

◇ **Příklad:** Použití `select(2)`.

```
static int done=0;
int dualcopy(int ttyfd, int modemfd)
{
    fd_set rfds, wfds, xfds;
    char mbuffer[BUFSIZE], tbuffer[BUFSIZE];
    char *mptr, *tptr;
    int mlen=0, tlen=0;
    int nfds, i;

    int maxfd=1+(ttyfd>modemfd?ttyfd:modemfd);
    while(!done) {
        FD_ZERO(&rfds);
        FD_ZERO(&wfds);
        FD_ZERO(&xfds);
        FD_SET(modemfd, &xfds);
        FD_SET(ttyfd, &xfds);
        if (mlen) FD_SET(ttyfd, &wfds);
             else FD_SET(modemfd, &rfds);
```



```
        if (tlen) FD_SET(modemfd, &wfds);
             else FD_SET(ttyfd, &rfds);
        nfds = select(maxfd, &rfds, &wfds,
             &xfds, NULL);
        if (nfds == -1)
             switch(errno) {
             case EBADF:
                 printf("invalid fd!\n");
                 return -1;
             case EINTR:
                 /* Dostali jsme signál. */
                 continue;
             case ENOMEM:
                 printf("not enough memory!\n");
                 return -2;
             case EINVAL:
                 printf("Internal error!\n");
                 return -3;
```



```
             default:
                 printf("Invalid errno=%d\n",
                     errno);
                 return -4;
             }
        if (nfds == 0)
             /* Toto se stane jen při timeoutu. */
             contintue;
        if (FD_ISSET(ttyfd, &xfds)) {
             printf("Exception on tty\n");
             return 0;
        }
        if (FD_ISSET(modemfd, &xfds)) {
             printf("Exception on modem\n");
             return 0;
        }
```

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


`poll(2)` . . . . I/O multiplexing

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

Vezme seznam deskriptorů a zjistí, jestli nastala některá z událostí `events`. Pokud ano, nastaví příslušný bit v `revents`. Pokud ne, čeká na příchod události minimálně po `timeout` milisekund. Je-li `timeout` rovno -1, čeká neomezeně dlouho na příchod události nebo na příchod signálu. `poll(2)` není ovlivněno flagy `O_NDELAY` nebo `O_NONBLOCK` na deskriptoru.

Návratová hodnota -1 značí chybu, 0 znamená vypršení časového limitu, kladná hodnota značí počet deskriptorů, u nichž služba nastavila nenulovou hodnotu `revents`.

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

POLLIN – data jiné než vysoké priority mohou být čtena bez blokování.
POLLRDNORM – data normální priority (priorita 0) mohou být čtena bez blokování.
POLLRDBAND – data nenulové priority mohou být čtena bez blokování.
POLLPRI – data s vysokou prioritou mohou být čtena bez blokování.

POLLOUT – data normální priority mohou být zapsána bez blokování.
POLLWRNORM – totéž jako `POLLOUT`.
POLLWRBAND – data s vysokou prioritou (> 0) mohou být zapsána bez blokování.

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

POLLERR – došlo k chybě na příslušném deskriptoru.
POLLHUP – došlo k zavěšení linky.
POLLNVAL – deskriptor tohoto čísla není otevřen.

Pokročilé I/O operace

Zamykání souborů

- **Zamykání pro čtení nebo pro zápis** – zámek pro čtení znamená, že selže pokus o vytvoření zámku pro zápis, ale ne dalšího zámku pro čtení. Zámek pro zápis je jednoznačný v celém systému.
- **Nepovinné a povinné zamykání** – *advisory/mandatory locking*. Nepovinné – jen vzhledem k dalším zámkům; povinné – i vzhledem k I/O operacím. Povinné jen v SVR3 a SVR4 (plus některé další, např. Linux). Povinné – s-gid bit na souboru, který není přístupný pro provádění pro skupinu. POSIX.1 – jen nepovinné.
- **Metody zamykání** – `fcntl(2)` (POSIX.1, SysV, 4.4BSD), `lockf(2)` (SysV), `flock(2)` (BSD).

- Zámek souvisí s procesem a se souborem (struktura `inode`). Pokud proces skončí, jsou jeho zámky zrušeny. Pokud se uzavře soubor, jsou také zámky zrušeny.
- Zámek se nedědí na potomky při `fork(2)`.
- Zámek přetrvá přes volání `exec(2)`.

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

Zamykání přes `fcntl(2)`

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

Jako `cmd` se uvede jedna z hodnot `F_GETLK`, `F_SETLK` nebo `F_SETLKW`, `arg` je ukazatel na strukturu `flock`. Položky struktury `flock` mají tento význam:

l_type – jedno z `F_RDLCK` (zámek pro čtení), `F_WRLCK` (zámek pro zápis) nebo `F_UNLCK` (pro zrušení zámku).
l_start – offset prvního bajtu zamykaného regionu.

l_whence – jedno z `SEEK_SET`, `SEEK_CUR` nebo `SEEK_END` (viz `lseek(2)`).
l_len – délka zamykaného regionu. Je-li nulové, značí zámek od `l_start` do konce souboru.
l_pid – PID procesu, který drží zámek (vrací `F_GETLK`).

Jednotlivé příkazy fungují takto:

F_GETLK – zjistí, jestli je vytvoření zámku blokováno nějakým jiným zámkem. Pokud takový zámek existuje, je struktura `flock` naplněna popisem tohoto zámku. Pokud neexistuje, je `l_type` změněno na `F_UNLCK`.
F_SETLK – nastaví nebo zruší zámek. Není-li možno zámek vytvořit, vrátí -1 s `errno` rovno `EACCESS` nebo `EAGAIN`.
F_SETLKW – blokující verze `F_SETLK`. Pokusí se nastavit zámek. Pokud to není možné, zablokuje se do doby, než bude možné zámek vytvořit nebo než přijde signál.

Scatter-gather I/O

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

---

readv(2) . . . . scatter read

```
#include <sys/types.h>
#include <sys/uio.h>
ssize_t readv(int fd, struct iovec iov[],
    int iovcount);
struct iovec {
    void *iov_base;
    size_t iov_len;
};
```

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

---

writev(2) . . . . gather write

```
#include <sys/types.h>
#include <sys/uio.h>
ssize_t writev(int fd, struct iovec iov[],
    int iovcount);
```

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

Memory-mapped I/O

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

---

mmap(2) . . . . Mapování souboru do paměti

```
#include <unistd.h>
#include <sys/mman.h>
#ifdef _POSIX_MAPPED_FILES
void *mmap(void *start, size_t length,
    int prot, int flags, int fd,
    off_t offset);
int munmap(void *start, size_t length);
#endif
```

Služba `mmap(2)` je součástí POSIX.4 (1b). Požádá systém o namapování `length` bajtů ze souboru `fd` od offsetu `offset` dále do paměti s tím, že systém se bude snažit tento blok dat namapovat do paměti na adresu `start`. Tento údaj je ale brán pouze jako doporučení; systém může namapovat soubor i jinam. Parametr `prot` určuje přístupová práva k nově mapovaným stránkám. Může být logický součet několika následujících hodnot:

PROT_EXEC – stránky mohou být prováděny.
PROT_READ – stránky jsou přístupné pro čtení.
PROT_WRITE – do stránek lze zapisovat.
PROT_NONE – ke stránkám nelze přistupovat.

Parametr `flags` specifikuje typ mapovaného objektu a to, jestli se jeho modifikace mají projevit i navenek nebo mají být považovány za soukromé. Hodnota je logický součet některých z následujících konstant:

MAP_FIXED – zakazuje jádru zvolit jinou adresu pro mapování než `start`. Parametr `start` pak musí být zarovnán na velikost stránky (viz `sysconf(2)`). Používání této volby se nedoporučuje z důvodu přenositelnosti.

MAP_SHARED – zápis do mapované oblasti se projeví v namapovaném souboru i v paměti dalších procesů, které si tento úsek souboru namapovaly.

MAP_PRIVATE – mapovaná oblast je copy-on-write kopií obsahu mapovaného souboru. Změny provedené procesem se neprojeví jinde.

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

Služba `munmap(2)` odmapuje příslušnou část mapovaného regionu. Další přístup k odmapované části je ilegální (způsobí zaslání signálu `SIGBUS` nebo `SIGSEGV`).

---

msync(2) . . . . Synchronizace regionu

```
#include <unistd.h>
#include <sys/mman.h>
#ifdef _POSIX_MAPPED_FILES
#ifdef _POSIX_SYNCHRONIZED_IO
int msync(const void *start, size_t length,
    int flags);
#endif
#endif
```

Tato služba uloží změny v namapovaném souboru zpět na disk. Uložení změn se týká paměti od adresy `start` velikosti `length`. Parametr `flags` může být log. součet následujících:

MS_SYNC – služba počká na dokončení zápisu na disk.

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

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

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

---

mprotect(2) . . . . Nastavení přístupu k regionu

```
#include <sys/mman.h>
int mprotect(const void *addr, size_t len,
    int prot);
```

Nastavuje přístupová práva pro oblast paměti od adresy `addr` velikosti `len`. Přístupová práva se nastaví parametrem `prot` jako log. součet jedné nebo více hodnot z `PROT_NONE`, `PROT_READ`, `PROT_WRITE` a `PROT_EXEC` s významem stejným jako u `mmap(2)`. Stará hodnota přístupových práv nemá vliv na nový stav. Parametr `addr` musí být zarovnán na velikost stránky. POSIX.4 (1b) říká, že `mprotect(2)` může být použito pouze na regiony získané pomocí `mmap(2)`.

---

mlock(2), munlock(2) . . . . Zákaz stránkování v regionu

```
#include <sys/mman.h>
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

Zakáže/povolí stránkování v daném rozsahu operační paměti. Stránkování na odkládací prostor je zakázáno do doby, než proces skončí nebo změní dané nastavení. Toto nastavení se nedědí přes `fork(2)` a `exec(2)`.

Využití – real-time aplikace, kryptografické aplikace.

`mlockall(2)`, `munlockall(2)`. Zamčení celého adresního prostoru

```
#include <sys/mman.h>
int mlockall(int flags);
int munlockall();
```

Zamče všechny stránky paměti procesu. Stránky nejsou odkládány na odkládací zařízení. `flags` je logický součet některých z následujících konstant:

`MCL_CURRENT` – zamče všechny stránky, které jsou momentálně namapovány v paměti procesu.

`MCL_FUTURE` – regiony, mapované v budoucnu, budou zamčeny. Pokud dojde k překročení limitu, další pokus o mapování vrátí `ENOMEM`. V případě neúspěšného pokusu o zvětšení zásobníku proces dostane `SIGSEGV`.

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

◇ **Úkol:** Jak předalokovat dostatečně velký prostor na zásobníku?