UNIX
Programování a správa systému I

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

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

Úvod

Předpoklady

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

Cíle kursu

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

Ukončení předmětu

- Test – 20 otázek.
- Hodnocení: –1 až 2 body na otázku, na kolokvium je potřeba 18 bodů a více.

Obsah přednášky – I.

- 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.

Obsah přednášky – II.

- 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.

Vývojové prostředí

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* versus 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.

Ovládání dynamického linkeru

- `LD_LIBRARY_PATH` – seznam adresářů, oddělený dvojtečkami. Určuje, kde se budou hledat dynamicky linkované knihovny.
- `LD_PRELOAD` – objekt, který bude přilinkován jako první. Např. pro předefinování knihovní funkce.
- U set-uid a set-gid programů dynamický linker ignoruje výše uvedené proměnné.
- `/etc/ld.so.conf` – globální konfigurace.
- `/etc/ld.so.conf.d/` – usnadnění práce správcům balíčků. 
- `ldconfig(8)` – generuje symlinky podle verzí a cache. 

Linkování v době kompilace

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

Linkování v době běhu – formát ELF

- 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 začínjící podtržítkem jsou privátní symboly systému.

Úkol:

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

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

- gdb – GNU debugger. Nejrozšířenější možnosti (volání funkcí z programu, změna volací sekvence na zásobníku, atd.).
- dbx – pochází ze SVR4. Širší možnosti, ovládání příkazy ve formě slov.
- 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.

GNU Autoconf

- Systém automatické konfigurace programu
- Generuje `Makefile`, `config.h` a `configure` na základě `Makefile.in`, `config.h.in` a `configure.ac`.
- Používá shell a `m4`

Autoconf: příklad Makefile.in

```
PERL5=@PERL5@
MATH_LIB=@MATH_LIB@
CFLAGS=@CFLAGS@
CC=@CC@
program: program.o
   $(CC) $(CFLAGS) -o program program.o $(MATH_LIB)
```

Autoconf: příklad configure.ac

```
AC_INIT(singlept.c)
AC_CONFIG_HEADER(config.h)
dnl Checks for programs.
AC_PROG_INSTALL
AC_PROG_MAKE_SET
AC_PROG_CC
AC_CANONICAL_SYSTEM
AC_CHECK_PROGS(PERL5, perl5 perl, false)
AC_CHECK_LIB(m, cos, MATH_LIB=-lm,
    AC_MSG_ERROR(The cos() function not \
    found in the math library))
AC_SUBST(MATH_LIB)
AC_OUTPUT(Makefile doc/Makefile)
```

Autoconf: příklad config.h.in

Poznámka: lze též generovat pomocí `autoheader(1)`.

```
/* Define if you have the getopt_long()
   C-library function. */
#undef HAVE_GETOPT_LONG
```

Automake

- Konstrukce `Makefile.in` – zdlouhavé.
- Zohlednění všech možností – náročné (cross-kompilace, instalace, knihovny, ...).

Příklad `Makefile.am`

```
bin_PROGRAMS = mujprogram mujprogram_SOURCES =
mujprogram.c # mujprogram_LDADD = $(LIBOBJS)
```

Libtool

- Výroba sdílených knihoven – není přenositelná.
- Speciální kompilace pro sdílené knihovny.
- Verze symbolů.
- Verze knihovny.
- Princip činnosti – modifikace příkazové řádky.

Příklad

```
libtool –mode=cc gcc -O2 -c mujprogram.c
```

Pkg-config

- Větší knihovny – specifické volby kompilace, cesty, atd.
- Více verzí knihoven – v různých adresářích.
- Vynucení verze knihovny.

Příklad

```
$ pkg-config --cflags glib-2.0
-I/usr/include/glib-2.0 \
-I/usr/lib64/glib-2.0/include
```

- Autoconf: `PKG_CHECK_MODULES()`, ...

Normy API

ANSI C

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

IEEE POSIX

- Portable Operating System Interface – IEEE 1003.
- API UNIXu – POSIX.1, nejnovější revize 2004.
- Rozhraní shellu – POSIX.2.
- Real-time extenze – POSIX.1b (dříve POSIX.4).
- Vlákna – POSIX.1c (dříve POSIX.4a).

Single UNIX Specification

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

Další normy

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

Volitelné vlastnosti v normách

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

ANSI C Limity

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

POSIX.1 – detekce verzí

```
#define _POSIX_SOURCE
#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: volitelné v době kompilace

Globální limity v POSIX.1

sysconf(2) — Globální limity

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

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

Souborové limity v POSIX.1

pathconf(2) Souborové limity

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

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

POSIX.1 compile-time limity

- `ARG_MAX`
- `CHILD_MAX`
- `PIPE_BUF`
- `LINK_MAX`
- `_POSIX_JOB_CONTROL`
- ... a další.

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

Úkol:

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

Program v uživatelském prostoru

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()`.

Start uživatelského programu

`main()` Vstupní bod programu

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

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

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

Ukončení programu v C

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

Ukončení programu v C

exit(3) Ukončení programu

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

- Knihovní funkce.
- Uzavření otevřených souborů (i vylití bufferů).
- Volání statických destruktorů (v C++).
- Ukončení procesu.

Uživatelský úklid v programu

atexit(3) Vyvolání funkce při exit(3)

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

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

Ukončení procesu

_exit(2) Ukončení procesu

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

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

Násilné ukončení programu

abort(3) — Násilné ukončení

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

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

Úkol:

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

Práce s argumenty programu

Bývá zvykem akceptovat přepínače (volby) s následující syntaxí:

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

Úkol:

Jak smažete soubor jménem `-Z`?

Zpracování přepínačů

getopt(3) Zpracování přepínačů

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

getopt(3): Příklad

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

Zpracování dlouhých přepínačů

getopt_long(3)

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

Knihovna POPT

- Modulární struktura – např. Glib, GTK+ a GNOME vrstva.
- ftp://ftp.redhat.com/pub/redhat/code/popt/

Chybový stav služeb jádra

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

`errno`	Chybová hodnota služby jádra

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

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

Příklad: errno

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

Textový popis chyby

perror — Tisk chybového hlášení

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

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

Příklad: perror(3)

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

Pro ENOENT vypíše následující:

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

Získání chybové zprávy

strerror(3) Textový popis chyby

```
#include <string.h>

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

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

Proměnné prostředí

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

`getenv(3)` Získání obsahu proměnné

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

Nastavení proměnných

putenv(3), setenv(3) Nastavení proměnné

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

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

Rušení proměnných

unsetenv(3), clearenv(3)Rušení proměnných

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

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

Úkol:

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

Alokace paměti

malloc(3) — Alokace paměti

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

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

calloc(3) — Alokace pole

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

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

Alokace paměti

realloc(3) Změna alokovaného bloku

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

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

free(3) Uvolnění dynamické paměti

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

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

Alokace na zásobníku

alloca(3) Alokace na zásobníku

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

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

Nízkoúrovňová alokace

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

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

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

Problémy dynamické paměti

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

Ladící prostředky pro alokátor

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

Nelokální skoky

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

`setjmp(3)` Inicializace skoku

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

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

Volání nelokálního skoku

longjmp(3) — Nelokální skok

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

- Skok na místo volání `setjmp()`.
- 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 flags);
```

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

Parametr `flags` může být jedno z následujících:

- `RTLD_NOW` – křížové odkazy řešit hned a vrátí chybu, jsou-li nedefinované symboly.
- `RTLD_LAZY` – křížové odkazy se řeší až při použití (jen funkce).
- `RTLD_GLOBAL` – globální symboly dány k dispozici dalším později linkovaným objektům.

dlclose(3) Uzavření dynamické knihovny

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

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

dlsym(3) Získání symbolu z knihovny

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

dlerror(3) Chybové hlášení libdl

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

Příklad: knihovna libdl

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

Úkol: knihovna `libdl`

Úkol

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 rekompilace programu.
- Možnost nastavovat na úrovni uživatele.
- Možnost nastavovat různé kategorie.

Kategorie lokalizace

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

Názvy locales

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

- Jazyk – dle ISO 639 (pro nás cs)
- Teritorium – dle ISO 3316 (pro nás CZ)
- Znaková sada – například (ISO8859-2 nebo UTF-8)
- Modifikátor – například (EURO)

Názvy locales

```
cs_CZ.UTF-8, cs, cs_CZ, en_GB, de@EURO
```

Proměnné prostředí

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

setlocale(3) Nastavení lokalizace

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

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

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

Inicializace locales

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

Lokalizované třídění

strcoll(3) Porovnávání řetězců podle locale

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

Jako strcmp(3), bere ohled na LC_COLLATE.

strxfrm(3) Transformace řetězce podle locale

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

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

Lokalizované třídění

Úkol:

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

Katalogy zpráv

- Pro kategorii LC_MESSAGES.
- GNU gettext – překladové tabulky, vyhledávání řetězců.
- Zdrojové soubory: .po.
- Zkompilované soubory: .mo.

Příklad katalogu zpráv

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

Locales v programu v C

nl_langinfo(3) Zjištění informací o locale

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

`item` může být jedno z následujících:

- `CODESET` – název znakové sady.
- `D_T_FMT` – formát data a času (pro `strftime(3)`).
- `D_FMT, T_FMT`
- `DAY_1-7` – název dne v týdnu.
- `ABDAY_1-7` – zkratka dne v týdnu.
- `MON_1-7, ABMON_1-7`
- `RADIXCHAR` – oddělovač desetinných míst.
- `YESEXPR, NOEXPR`.
- `CRNCYSTR` – symbol měny a umístění (+, –, .).

Locales na příkazové řádce

locale(1) Lokalizačně specifické informace

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

Znakové sady

iconv(3) Konverze znakových sad

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

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

Úkol:

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

Definice locales

localedef(8) — Definice locale

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

Vytvoří binární podobu locale pro přímé použití v aplikacích.

Jádro systému

Start systému – firmware.

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

Primární zavaděč systému

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

Sekundární zavaděč systému

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

Start jádra: parametry jádra

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

Průběh inicializace jádra

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

Inicializace zařízení

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

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 `initu`.

Ramdisk v Linuxu

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

Bootovací zprávy jádra

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

Příklad:

bootovací zprávy reálného systému

Konfigurace jádra

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

Monolitické jádro

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

Mikrojaderné systémy

- CMU Mach, OSF Mach, L4, minix, Windows NT HAL, QNX, VxWorks, ...
- Co nemusí být v jádře, dát mimo něj.
- Procesy (servery) pro správu virtuální paměti, ovládání zařízení, disků a podobně.
- Dobře definovatelné podmínky činnosti (jen teoreticky, protože: DMA, SMI, IOMMU a další problémy).
- Předávání zpráv – malá propustnost, velká latence.
- Linux is obsolete (1992) – `http://oreilly.com/catalog/opensources/book/appa.html`

Modulární jádro

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

Modulární jádro v Linuxu

- Dynamické přidávání ovladačů podle potřeby.
- Závislosti mezi moduly (`depmod(8)`).
- Dynamická registrace ovladačů: `register_chrdev()`, `register_blkdev()`, `register_netdev()`, `register_fs()`, `register_binfmt()` a podobně.
- Dohledávání pomocí identifikátorů sběrnice (např. PCI ID).

Procesy v jádře

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

Kontext

Stav systému, příslušný běhu jednoho procesu/vlákna.

- Přepnutí kontextu – výměna právě běžícího procesu za jiný.
- Linux – `struct task_struct`, `current`  .

Procesy uvnitř jádra

Problém

Pod jakým kontextem mají běžet služby jádra?

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

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.
- Preemptivní/nepreemptivní jádro – může dojít k přepnutí kontextu kdekoli v jádře?

Zpracování přerušení v jádře

Problém

Pod jakým kontextem lze provádět přerušení?

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

Odložené vykonání kódu

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

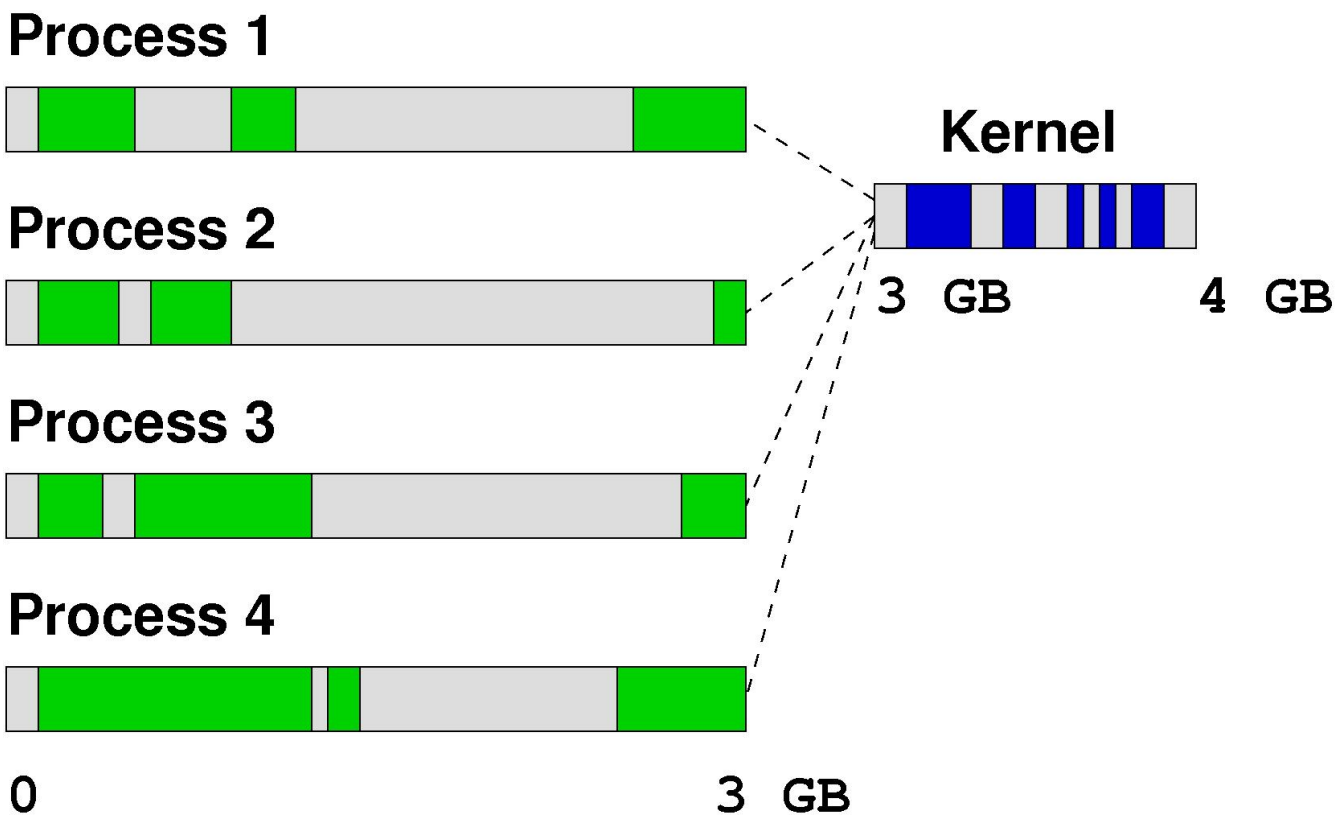
Virtuální paměť

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

Translation Look-aside Buffer

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

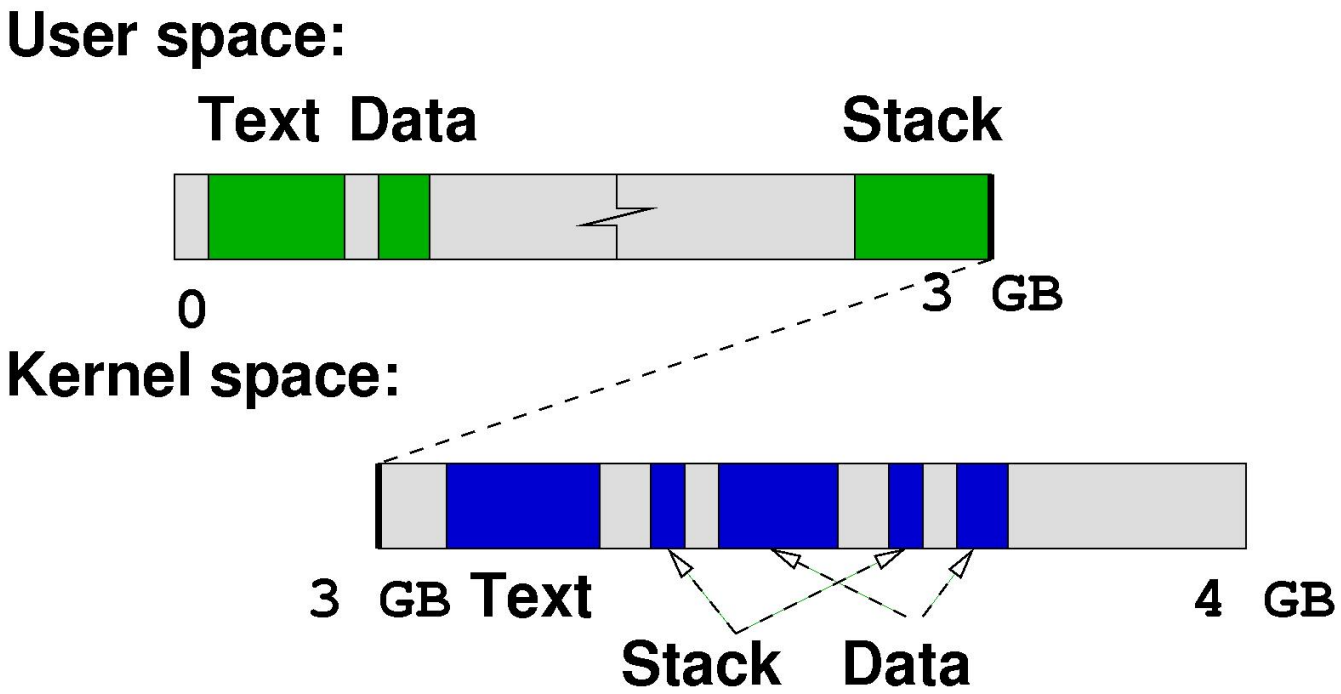
Prostor jádra a uživatelský prostor



Prostor jádra a uživatelský prostor

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

Virtuální paměť uvnitř jádra

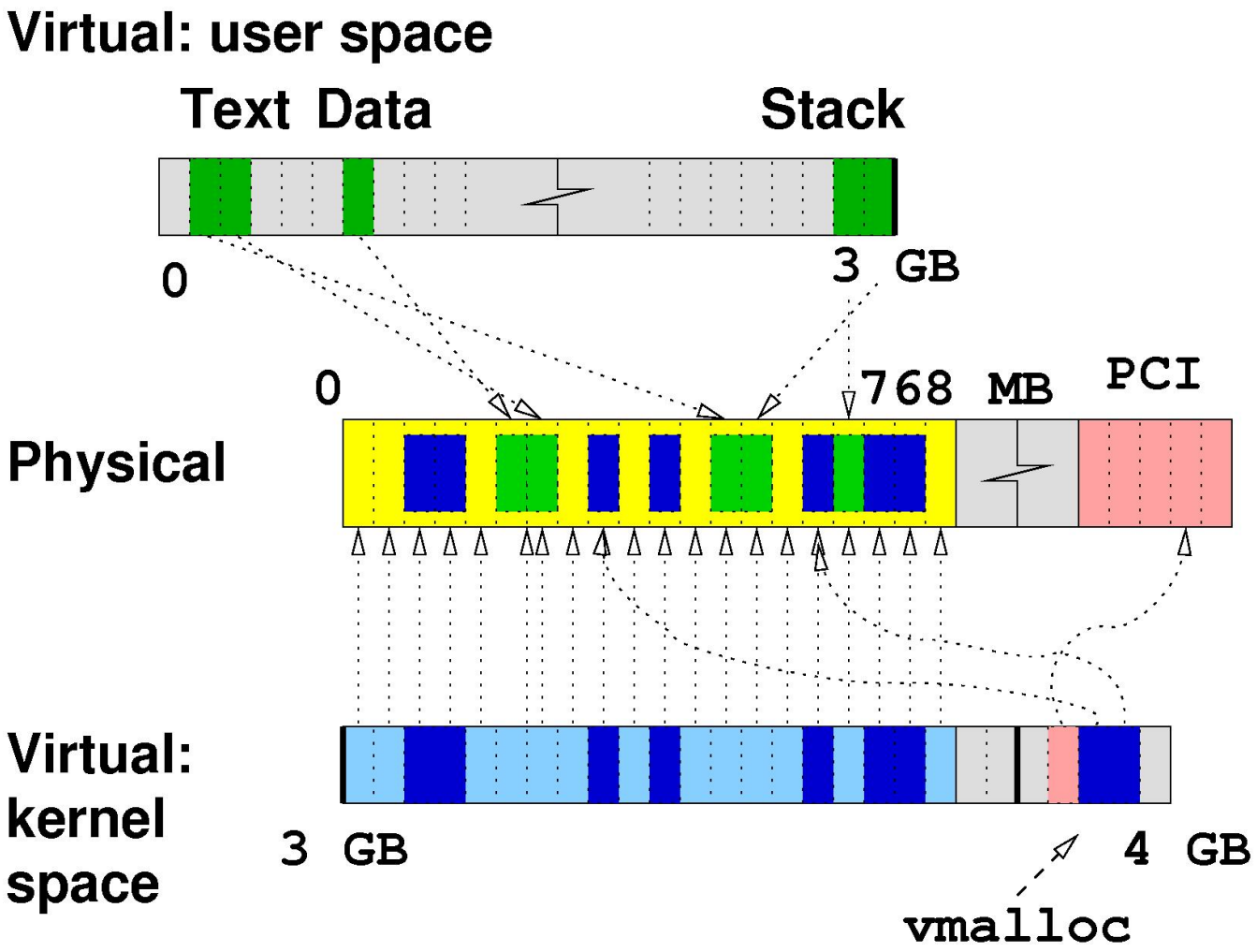


- Zásobník v jádře – pro každý thread/kontext.
- Linux – 1 stránka/thread, nastavitelné 2 stránky/thread  .

Jádro a fyzická paměť

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

Jádro a fyzická paměť



Fyzická paměť 32-bitového Linuxu

Úkol:

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

Paměť z hlediska hardwaru

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

Úkol:

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

Přístup do uživatelského prostoru

- Přístup do user-space: proces předá jádru ukazatel (např. buffer pro `read(2)`).
- Robustnost – user-space nesmí způsobit pád jádra.
- Validace před použitím? Problémy ve vícevláknových programech (přístup versus změna mapování v jiném vlákně).
- Poznámka: Přístup do uživatelského prostoru není možný uvnitř ovladače přerušení (proč?).

Přístup do user-space v Linuxu

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

Implementace v Linuxu

- Využití hardwaru CPU – kontrola přístupu do paměti. Přidání kontroly do `do_page_fault()`.
- Tabulka výjimek – adresa instrukce, která může způsobit chybu, opravný kód.
- Normální běh – cca 10 instrukcí bez skoku.
- ELF sekce – pro generování druhého toku instrukcí.
- Viz též `linux/arch/x86/include/asm/uaccess.h`, např. `__put_user_asm_u64()`.

Použití uživatelského ukazatele

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

Problém: volání služby jádra zevnitř jádra

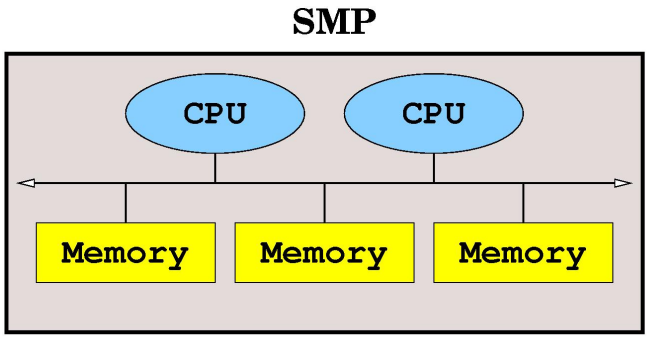
- Nutno předem oznámit. Linux: `set_fs(KERNEL_DS)`
- Například: `net/socket.c: kernel_sendmsg()`.

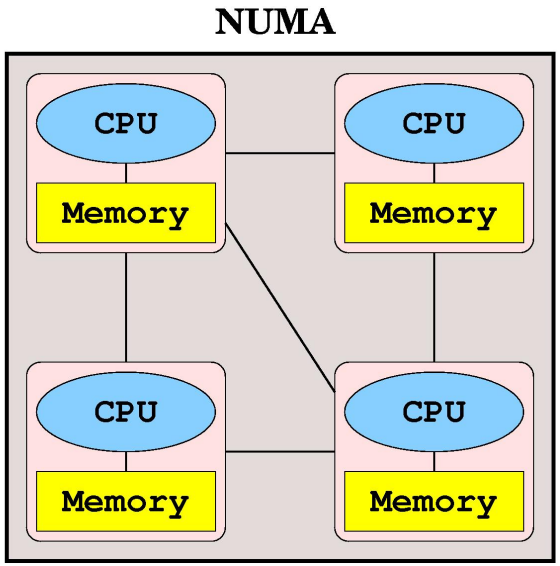
Paralelní stroje

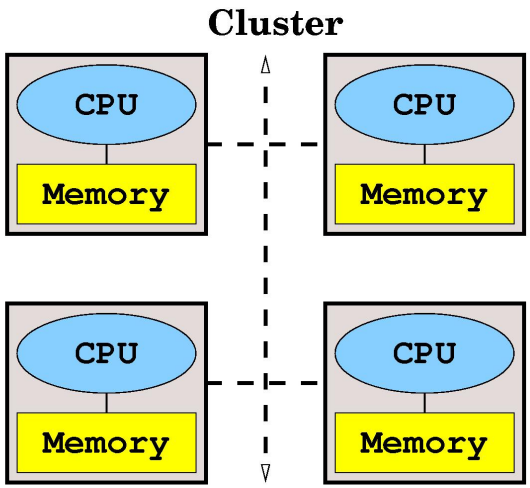
- SMP – symetrický multiprocesing. Společný přístup všech CPU k paměti.
- NUMA – hierarchická paměť – z určitých CPU rychlejší přístup než z jiných (cc-NUMA – cache coherent).
- Multipočítače – na částech systému běží zvláštní kopie jádra (clustery a podobně).

- 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.).

Zamykání a 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.

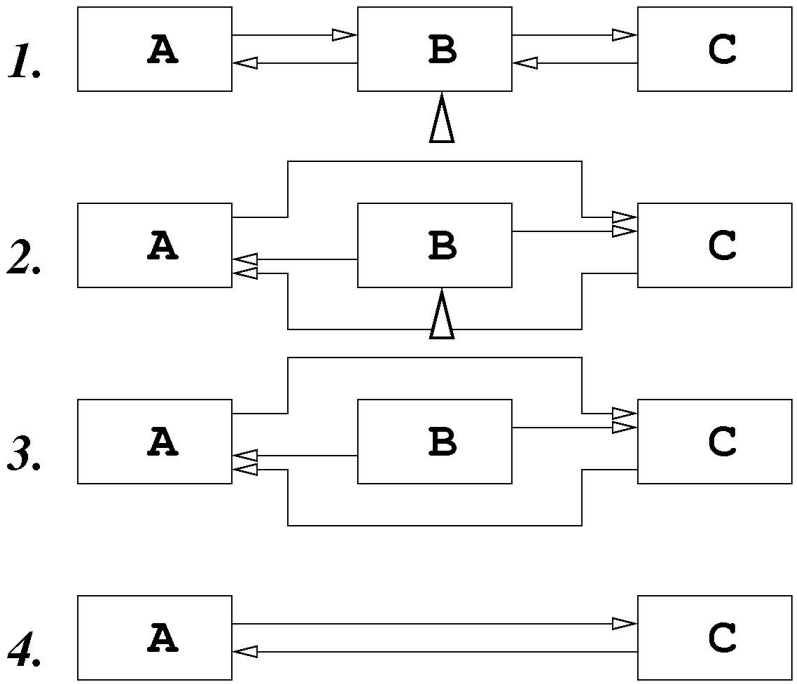
Princip činnosti RCU

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

Jak implementovat RCU?

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

Příklad: seznamy a RCU



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

Alokátor paměti v jádře

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

Alokace paměti v jádře Linuxu

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

Cache alokovaných objektů

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

Problémy velkého množství alokací

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

SLAB alokátor

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

Časovače

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

Čekací fronty

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

Procesy

Procesy

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

Paměť procesu:

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

Atributy procesu

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

Atributy procesu jsou:

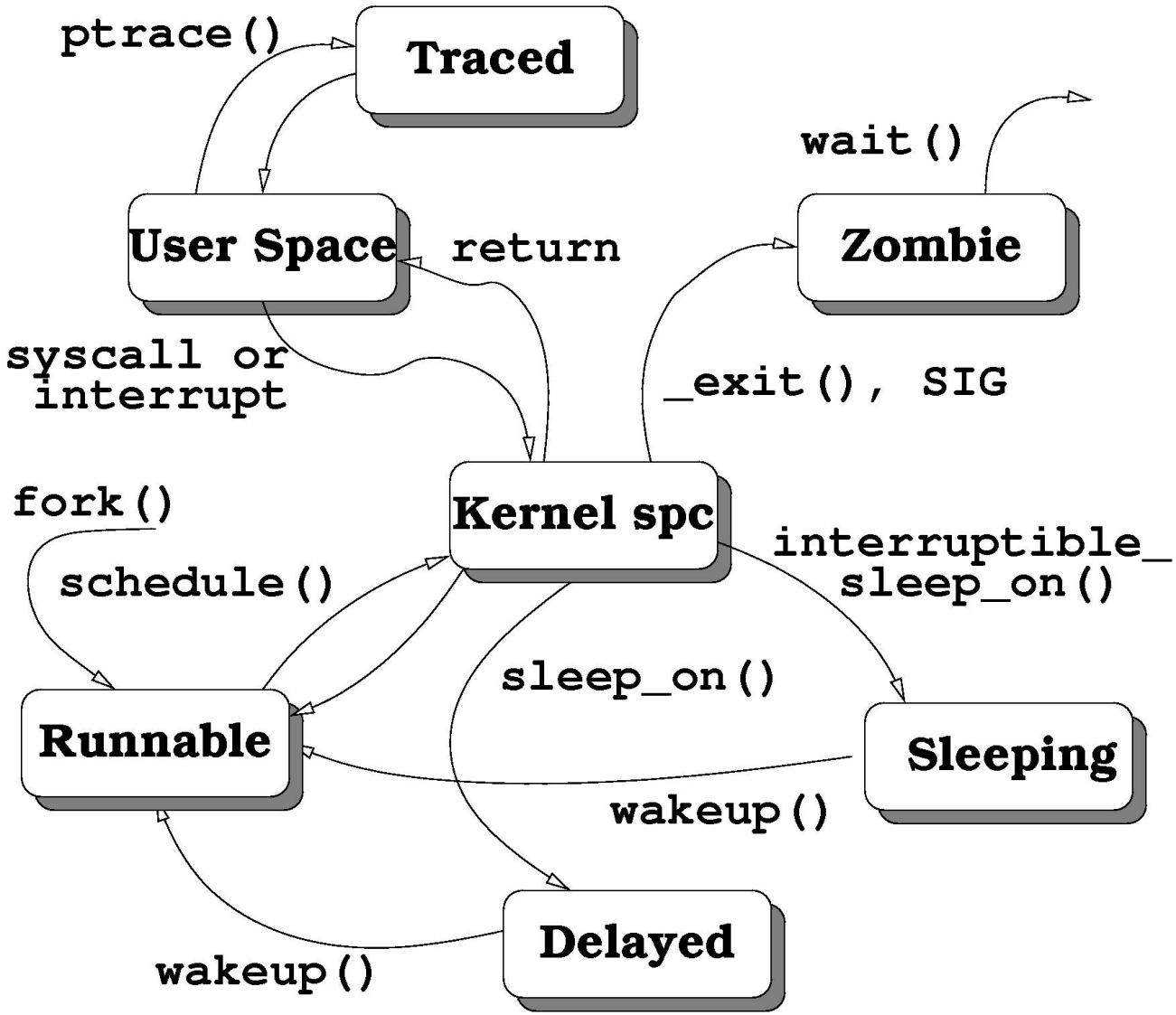
- Stav procesu – viz dále.
- Program counter – čítač instrukcí; místo, kde je proces zablokován (WCHAN).
- Číslo procesu – PID.
- Rodič procesu – PPID (rovno 1, pokud neexistuje).
- Priorita procesu

(pokračování)

Další atributy procesu

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

Stavy procesu



Služba jádra

- Kód definován v jádře
- Přepnutí oprávnění CPU
- 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 versus rodič

Potomek dědí téměř vše od rodiče. Výjimky jsou:

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

Optimalizace vfork(2)

vfork(2) — Virtuální fork()

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

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

Čekání na ukončení potomka

wait*(2) — Zjištění stavu potomka

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

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

Informace o stavu potomka

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

Upřesnění `waitpid(2)`

Parametr `options` je nula nebo logický součet následujících:

- `WNOHANG` – nezablokouje se čekáním.
- `WUNTRACED` – i při pozastavení nebo ladění potomka.
- `WCONTINUED` – i při znovuspuštění potomka (Linux > 2.6.9 ).

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: fork() a wait() – I.

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

Příklad: fork() a wait() – II.

```
potomek() {
    ...
    exit(status);
}

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

Spuštění jiného programu

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

Služby jádra třídy exec*(2,3)

exec(3) Spuštění procesu

```
#include <unistd.h>
extern char **environ;
int execl(char *path, char *arg, ...);
int execlp(char *path, char *arg, ...);
int 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);
```

Další vlastnosti exec*(2,3)

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

Vyvolání shellu

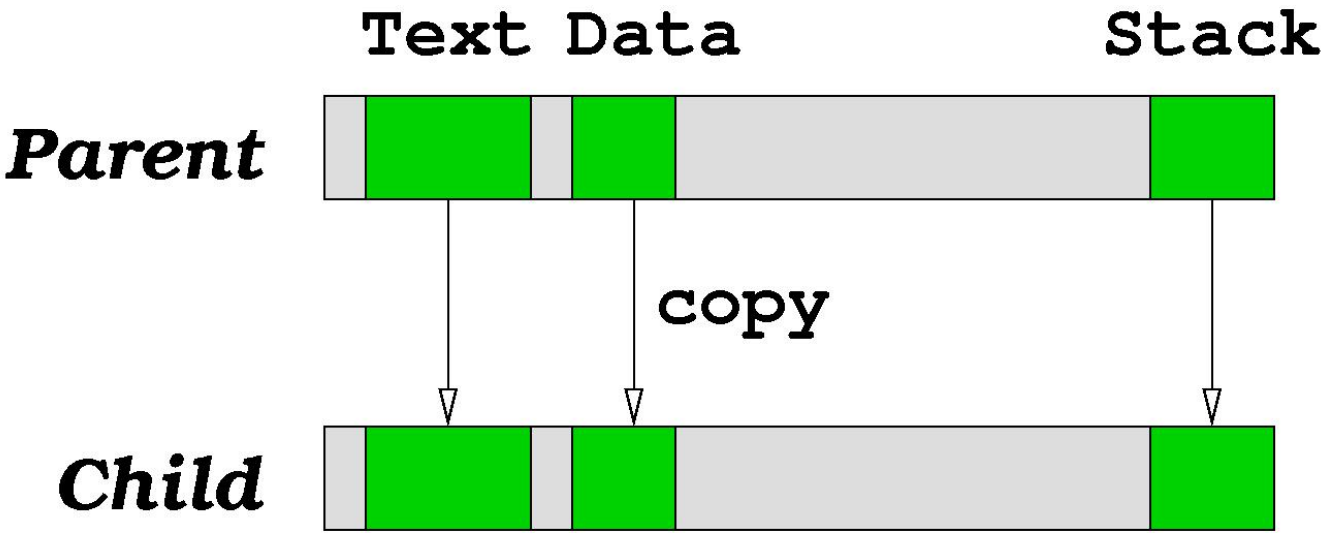
system(3) — Vyvolání příkazu shellu

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

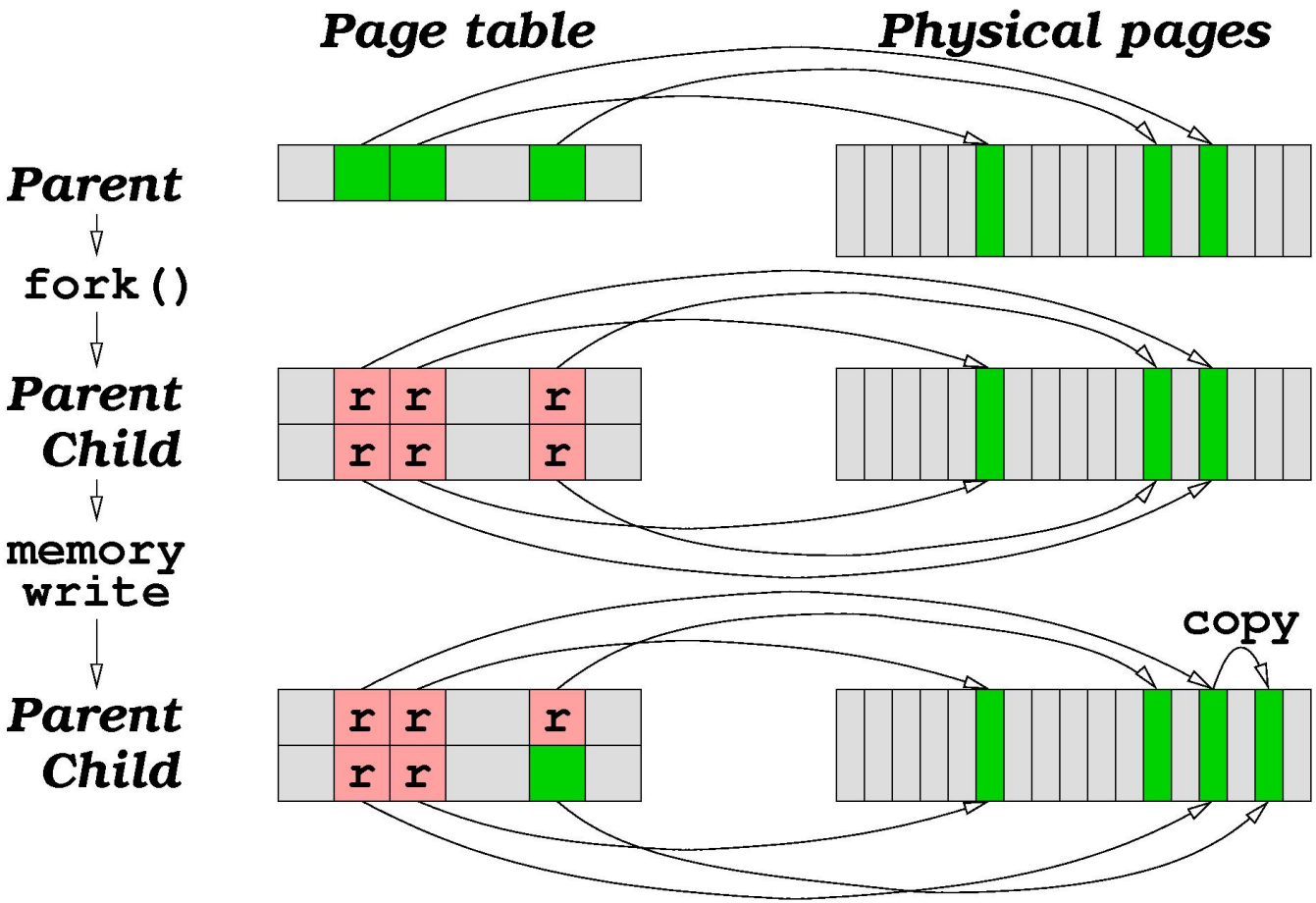
Spustí příkaz `/bin/sh -c string` jako potomka a počká na jeho dokončení.

fork(2) bez stránkování

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



fork(2) se stránkováním



Systémy se stránkováním

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

Stránkování na žádost

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

I/O operace

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

Problém

Je rychlejší zkopírovat stránku nebo ještě jednou namapovat tutéž stránku?

Alokace paměti

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

Memory overcommitment

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

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

Problém

Jak operační systém pozná, že došla paměť?

Výhody stránkovacích systémů

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

Program na disku

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

Binární formát script

- Hlavička – `0x2123` (nebo `0x2321` na big-endian systému). Textová podoba – `#!`. Následuje jméno (cesta) interpreteru, který se na daný soubor spustí, plus jeho parametry.
- Příklad – `#!/usr/bin/perl -ne`, program v Perlu.
- Jméno 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.

Přístupová práva procesu

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

Čtení přístupových práv

getuid(2), getgid(2) — Čtení UID/GID

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

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

Nastavení přístupových práv

setuid(2) Změna efektivního UID

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

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

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

Záměna UID

setreuid(2) — Výměna r-e UID

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

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

4.3BSD extenze pro systémy bez uloženého UID/GID.

Uložené ID

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

- Změna reálného UID: pouze superuživatel.
- Efektivní UID je nastaveno službou exec(2), pokud má příslušný program nastavený set-uid bit. Jinak se efektivní UID nemění.
- Uložené UID: při exec(2) se kopíruje z efektivního UID.

Příklad: změny UID procesu

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

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

nebo:

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

seteuid(2) Nastavení efektivního UID

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

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

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

Doplňková GID

- Starší verze UNIXu – při přihlášení uživatele: UID a GID podle souboru `/etc/passwd`, změna GID pomocí `newgrp(1)`.
- Novější systémy – doplňková (supplementary) GID.

- Zavedeno v 4.2 BSD.
- Seznam doplňkových GID (kromě reálného, efektivního a uloženého).
- Inicializace – při přihlášení podle `/etc/group`.
- Přístupová práva – efektivní GID a všchna doplňková GID.
- `NGROUPS_MAX` – limit počtu doplňkových GID.
- FIPS 151–1 – povinné a `NGROUPS_MAX` aspoň 8.

getgroups(2) Získání doplňkových GID

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

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

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

setgroups(2) Nastavení doplňkových GID

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

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

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

initgroups(3) — GID podle /etc/group

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

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

- Nastaví doplňková GID podle /etc/group.
- Navíc do seznamu skupin přidá skupinu group.
- Používá se při přihlašování.
- 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.

Systémové zdroje

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

Úkol:

Jaká nerovnost platí pro uživatelský, systémový a reálný čas?

Systémové zdroje

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

Úkol:

Jaká nerovnost platí pro uživatelský, systémový a reálný čas?

times(2) Získání časových informací

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

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

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

Další systémové zdroje

getrusage(2) Spotřebované systémové zdroje

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

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

- Parametr who je buďto RUSAGE_SELF nebo RUSAGE_CHILDREN.

Struktura rusage

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

Omezení systémových zdrojů

getrlimit(2), setrlimit(2)

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

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, struct rlimit *rlim);

struct rlimit {
    rlim_t rlim_cur;  /* Soft limit */
    rlim_t rlim_max;  /* Hard limit */
};
```

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

Typy systémových limitů

Parametr `resource` může být jeden z následujících:

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

Priorita procesu

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. Nepoužívat!

getpriority(2) Čtení priority procesu

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

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

Hodnota parametru `which` je jedna z následujících:

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

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

Skupiny procesů

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

Nastavení skupin procesů

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

```
#include <unistd.h>

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

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

Čtení skupin procesů

getpgid(2), getpgrp(2) — Skupiny procesů

```
#include <unistd.h>

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

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

Sessions

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

getsid(2), setsid(2)

```
#include <unistd.h>
pid_t getsid(pid_t pid); pid_t setsid(void);
```

- Selže, je-li proces vedoucím procesem skupiny.

Démoni

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

Vytvoření démona

daemon(3)

```
#include <unistd.h>

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

I/O operace

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 (POSIX.1: symbolické konstanty `STDIN_FILENO`, `STDOUT_FILENO` a `STDERR_FILENO`).

Otevření souboru

open(2), creat(2) — Otevření souboru

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

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

`flags` je jedno z `O_RDONLY`, `O_WRONLY` nebo `O_RDWR`, plus logický součet některých z konstant:

Parametry open(2)

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

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.

Čtení souboru

read(2) — Čtení souboru

```
#include <unistd.h>

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

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

Zápis do souboru

write(2) Zápis do souboru

```
#include <unistd.h>

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

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

Příklad: kopírování souborů – I.

```
#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|O_TRUNC,
          0777)) == -1){
     perror("Opening output file");
     exit(2);
}
```

Příklad: kopírování souborů – II.

```
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);
```

Pozice v souboru

lseek(2) Nastavení pozice v souboru

```
#include <unistd.h>

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

Parametr odkud nabývá těchto hodnot:

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

- Viz též llseek(2).
- Na některé typy souborů nelze použít lseek(2).

Příznak O_APPEND

Úkol:

Jaký je rozdíl v chování nasledujících dvou úseků kódu?

Příznak O_APPEND

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

Příznak O_APPEND

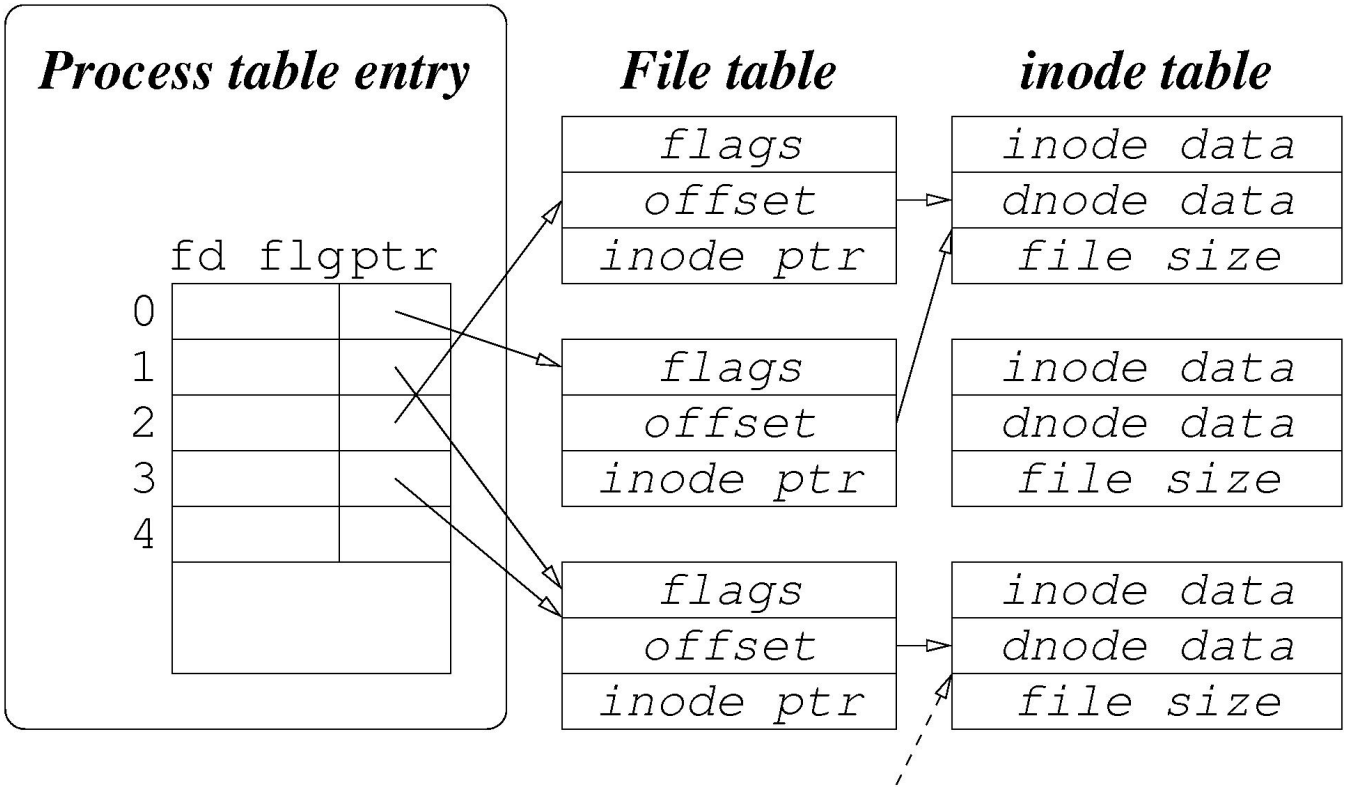
```
if ((fd = open(filename,O_WRONLY|O_APPEND))
          == -1) {
      perror("open");
      exit(1);
}
if (write(fd, buffer, size) == -1) {
      perror("write");
      exit(3);
}
```

O_APPEND a čtení ze souboru

Úkol:

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

Tabulka otevřených souborů



- Linux má navíc tabulku directory entry mezi strukturami `file` a `inode`
- BSD říká i-uzlům v paměti vnode.

Duplikování deskriptoru

dup(2),dup2(2) Duplikace deskriptoru

```
#include <unistd.h>

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

- Duplikování – nový odkaz na tutéž strukturu `file`.
- Použití: přesměrování v shellu.

Úkol:

Jak se liší funkce následujících dvou úseků kódu?

Varianta 1:

```
fd1=open(”file”,O_WRONLY|O_CREAT,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);
```

Změna vlastností deskriptoru

fcntl(2) Změna vlastností deskriptoru

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

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

Příkazy pro fcntl(2)

- F_DUPFD – duplikuje deskriptor fd do arg, podobně jako dup2(2).
- F_GETFD – čte flagy deskriptoru (pouze FD_CLOEXEC).
- F_SETFD – nastavuje flagy deskriptoru (FD_CLOEXEC).
- F_GETFL – čte flagy struktury file. Viz druhý parametr open(2).
- F_SETFL – nastavuje flagy struktury file. Lze nastavovat např O_APPEND, O_NONBLOCK, O_ASYNC a O_SYNC. Nikoliv měnit čtení na zápis a naopak.
- F_GETLK, F_SETLK – zamykání souboru (viz dále).

Práce s I/O zařízeními

ioctl(2) — Práce s I/O zařízením

```
#include <unistd.h>

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

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

Příklad: ioctl(2)

Nastavení signálu DTR na sériové lince na logickou 1:

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

Práce se soubory

i-uzel

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

Atributy i-uzlu

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

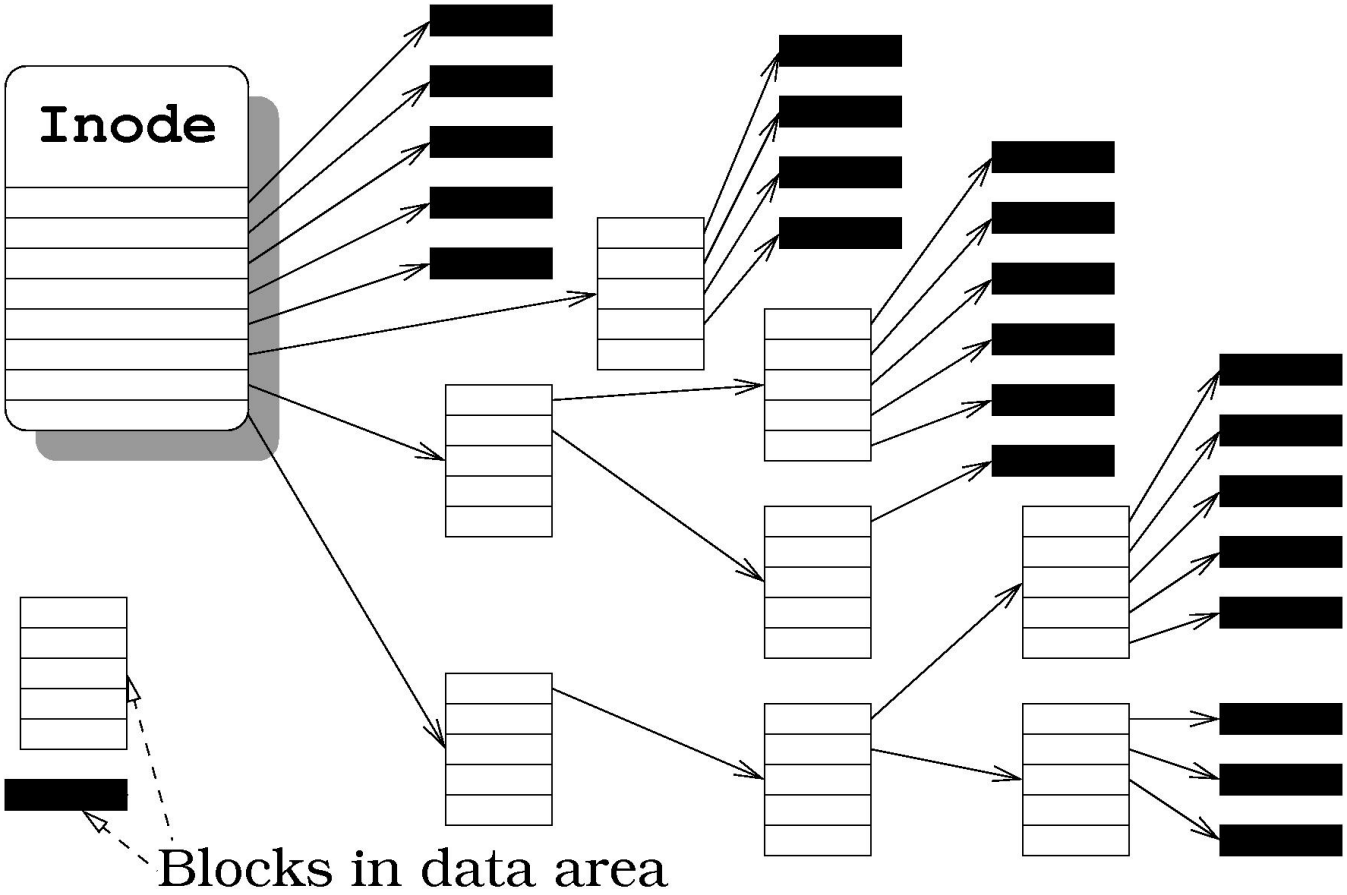
Odkazy na datové bloky

Tradiční přístup:

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

Novější souborové systémy: B-stromy nebo varianty.

i-uzel a datové bloky



i-uzel a datové bloky – vlastnosti

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

Úkol:

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

Čtení atributů i-uzlu

stat(2) — Informace o i-uzlu

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

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

- Služba `lstat(2)` neprochází symbolické linky.

Struktura stat

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

Typy souborů

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

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

Přístupová práva souboru

Přístupová práva lze z `st_mode` získat těmito maskami:

- `S_ISUID, S_ISGID, S_ISVTX` – set-uid bit, set-gid bit a sticky bit.
- `S_IRUSR, S_IWUSR, S_IXUSR` – práva vlastníka souboru.
- `S_IRGRP, S_IWGRP, S_IXGRP` – práva skupiny.
- `S_IROTH, S_IWOTH, S_IXOTH` – práva ostatního světa.

Ověření přístupových práv

access(2) Ověření přístupových práv

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

- Ověřuje proti reálnému UID/GID.
- Parametr mode – typ přístupu: log. součet F_OK, R_OK, W_OK nebo X_OK.
- POZOR: hrozí časová závislost a bezpečnostní problém!

Úkol:

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

Nově vytvářené soubory

- Vlastník – podle efektivního UID vytvářejícího procesu.
- Skupina – více možností.

- Podle efektivního GID procesu, který soubor vytvořil.
- 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.

Nově vytvářené soubory

umask(2) Maska přístupových práv

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

int umask(int newmask);
```

- Vrací předchozí nastavení 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 vlastníka souboru

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

Změna skupiny souboru

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

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

Poznámka

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

Změna vlastníka a skupiny

chown(2) Změna vlastníka/skupiny souboru

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

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

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

Změna přístupových práv souboru

chmod(2) Změna přístupových práv

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

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

Úkol:

Jakým parametrem nastavíte práva `rwsr-xr--` ?

Změna velikosti souboru

truncate(2) Nastavení velikosti souboru

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

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

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

Změna velikosti souboru

Úkol:

Napište program, který vytvoří soubor s dírou. Vyzkoušejte, které UN*Xové programy (např. `cp(1)`, `tar(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)`.

Pevné linky

link(2) Vytvoření odkazu na i-uzel

```
#include <unistd.h>

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

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

Smazání souboru

unlink(2) Zrušení odkazu na i-uzel

```
#include <unistd.h>

int unlink(char *path);
```

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

Příklad: anonymní dočasný soubor

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

Pozor: jen příklad; nutno lépe zvolit jméno souboru!

Smazání souboru nebo adresáře

remove(3) Zrušení souboru/adresáře

```
#include <stdio.h>

int remove(char *path);
```

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

Přejmenování souboru

rename(2) Přejmenování souboru/adresáře

```
#include <unistd.h>

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

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

Úkol:

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

Časy souboru

utime(2) Nastavení časů souboru

```
#include <sys/types.h>
#include <utime.h>
int utime(char *path, struct utimbuf *times);
struct utimbuf {
    time_t actime;
    time_t modtime;
}
#include <sys/time.h>
int utimes(char *path, strut timeval times[2]);
```

- Nastavení *atime* a *mtime*.
- Je-li parametr `times` NULL, nastaví na aktuální čas.
- Nastavovat smí pouze vlastník souboru (nebo superuživatel).

Nastavení časů souboru

Úkol:

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

Symbolické linky

Příklad

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

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

Vytvoření symbolicého linku

symlink(2) Vytvoření symbolického linku

```
#include <unistd.h>

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

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

Čtení obsahu symlinku

readlink(2) Čtení symbolického linku

```
#include <unistd.h>

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

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

Symbolické linky a přístup k souborům

Služby 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`.

Dočasný soubor z C

mkstemp(3) Vytvoření dočasného souboru

```
#include <stdlib.h>

int mkstemp(char *template);

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

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

Dočasný soubor ze shellu

Špatně

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

Správně

```
TMPFILE=‘mktemp /tmp/mujprogram.XXXXXX‘
ls > $TMPFILE
```

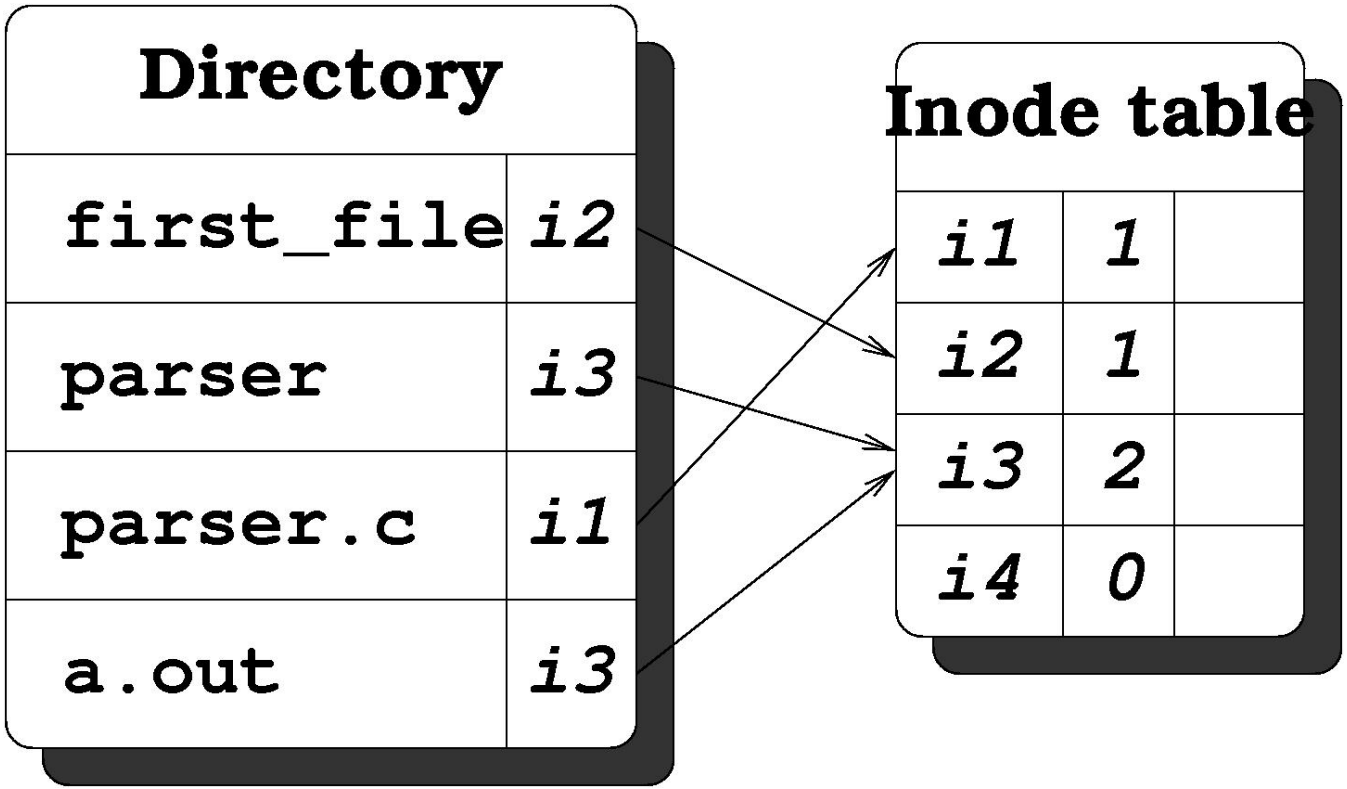
Adresáře

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

Adresáře – pokračování

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

Struktura adresáře



Nový adresář

mkdir(2) Vytvoření adresáře

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

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

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

Zrušení adresáře

rmdir(2) Smazání adresáře

```
#include <unistd.h>

int rmdir(char *path);
```

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

Čtení adresáře

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

opendir(3) Otevření adresáře

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

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

Čtení obsahu adresáře

readdir(3) Čtení adresářové položky

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

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

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

Úkol: vlastnosti čtení adresáře

Úkol:

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

Adresáře procesu

getcwd(3) Jméno pracovního adresáře

```
#include <unistd.h>

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

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

Změna pracovního adresáře

chdir(2) Změna pracovního adresáře

```
#include <unistd.h>

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

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

Kořenový adresář procesu

chroot(2) Změna kořenového adresáře procesu

```
#include <unistd,h>

int chroot(char *path);
```

- Změní kořenový adresář procesu.
- Povoleno pouze superuživateli.

Úkol:

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

Synchronizace disků

sync(2) Synchronizování disků

```
#include <unistd.h>

void sync(void);
```

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

Synchronizace deskriptoru

fsync(2), fdatasync(2)

```
#include <unistd.h>

int fdatasync(int fd);
int fsync(int fd);
```

- Zapíše všechny modifikované části souboru na disk.
- `fdatasync(2)` nezapisuje metadata souboru (čas modifikace, ...).
- Problém – patří nadřazený adresář pod „metadata souboru“?
- Problém – jak atomicky přepsat soubor? (`O_PONIES` :–)

Vytvoření speciálního souboru

mknod(2) — Vytvoření souboru

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

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

- Vytvoří soubor daného jména.
- Parametr `mode` specifikuje přístupová práva a typ souboru (`S_IFREG`, `S_IFCHR`, `S_IFBLK` nebo `S_IFIFO`, viz `stat(2)`).
- Linux – nelze takto vytvořit adresář.

Access Control Lists

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

Příklad: ACL

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

ACL – vlastnosti

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

Souborové systémy

Vlastnosti 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.

Svazky

Svazek (systém souborů, *volume*) 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

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.
- Komplikovaná implementace.

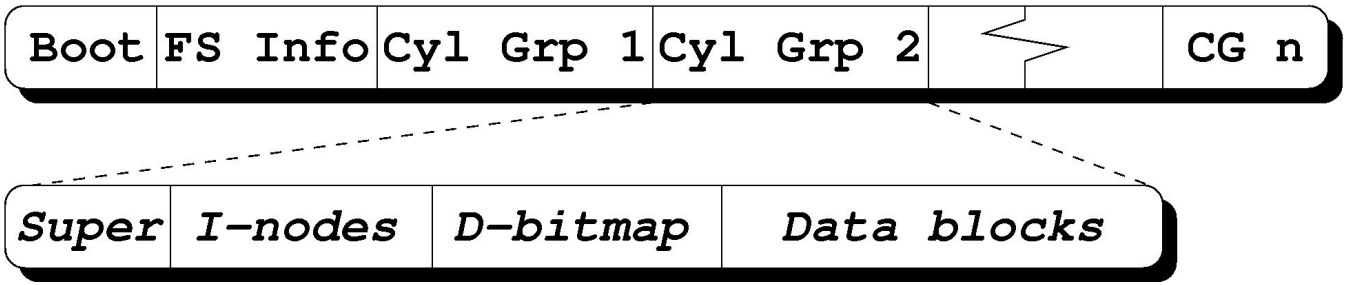
Žurnálované souborové systémy

- Transakční přístup.
- Změny nejprve zapsány do logu (žurnálu) a pak provedeny.
- Po havárii – přehrání celých transakcí.
- Některé operace – i rychlejší než nežurnálovaný FS.
- Celkově o něco pomalejší.
- Žurnál jen metadat nebo i dat.
- Chyba OS nebo HW se řeší pomocí `fsck(8)`.
- Jen transakce z jádra (ne user-space).

FAT

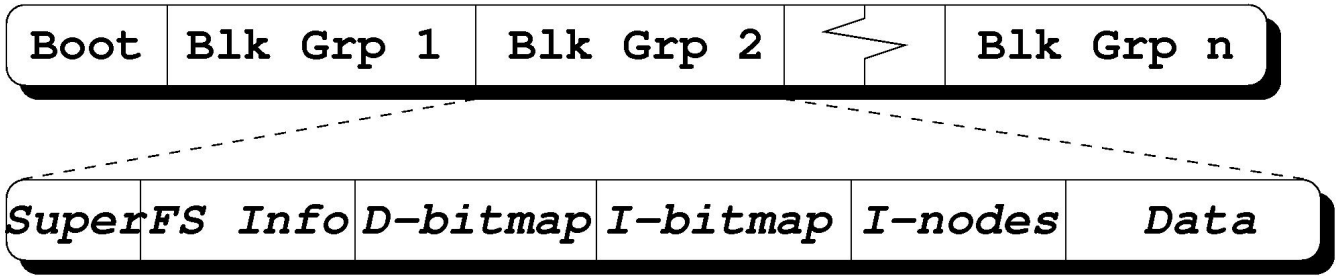
- Nemá i-uzly (nelze mít soubor ve více adresářích, nemá UNIXová přístupová práva).
- Pomalý přímý přístup k souboru (sekvenční procházení přes FAT).
- Fragmentace už při současném zápisu do dvou souborů.
- Fragmentace při rušení souboru.
- Na větších FS velká délka bloku špatné využití místa.
- Výhody – 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
- Původně: synchronní zápis metadat.
- FreeBSD: soft updates.
- Kontrola disku na pozadí.
- *BSD, Solaris (+ žurnálování), Linux.

Ext2 filesystem



- Skupiny bloků (block groups). Není nutná znalost geometrie disku. Jednodušší implementace, využití celých bloků.
- Alokační strategie: Předalokované bloky, alokace dat poblíž příslušných metadat, zamezení zaplnění jedné skupiny bloků.
- Obvykle 1 KB (až 4 KB) bloky – rychlejší než FFS s 4 KB bloky.
- Bitmapa volných i-uzlů.

Ext2FS – pokračování

- Asynchronní zápis metadat; na požádání umí i synchronní.
- Velikost až do 4 TB dat. Velká odolnost proti havárii.
- Rychlé symbolické linky.
- No-atime, relatime.
- Maximum mount count. `tune2fs(8)`.
- Možnosti při chybě – panic, remount r-only, ignore.
- `libe2fs` – knihovna pro přístup k e2fs. `e2defrag`.

Ext3FS

- 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.

ReiserFS

- Všechna data v jednom B+ stromu.
- Alokace místa – i menší kousky než jeden sektor.
- l-uzly – alokace podle potřeby.
- Efektivní i při velkém množství souborů v adresáři nebo velkém množství malých souborů.

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

- Rozdělení svazku – allocation groups velikosti 0.5 až 4 GB.
- Organizace dat – B+ strom
- DMAPI – data manipulation API – zpřístupnění vlastností B-stromu (vkládání/rušení dat uprostřed souboru).
- 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).

Sun ZFS

- Zettabyte File System
- Interně podobný jako Slab alokátor v paměti.
- RAID-Z – jednotlivé slaby s různou úrovní redundance.
- Kontrolní součty dat
- Self-healing (automatické opravy chyb).
- Copy-on-write: sjednocení duplicitních bloků

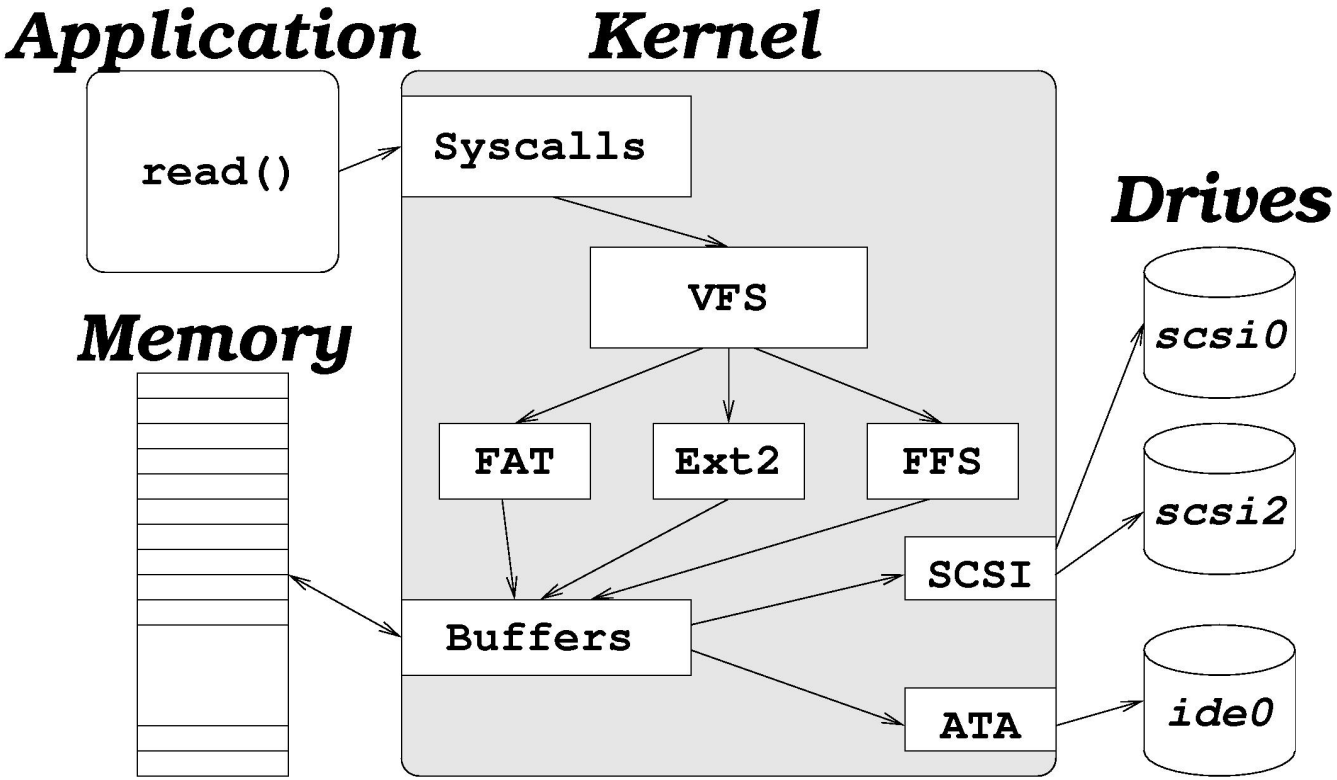
BTRFS

- Copy-on-write B-stromy
- Zapisovatelné snímky FS.
- Kontrolní součty metadat (volitelně i dat).

Další služby FS

- Komprese dat – celý FS nebo jen určité soubory.
- Obnova smazaných souborů.
- Bezpečné mazání souborů – ext[234]fs.
- Nepřemistitelné soubory – ext[234]fs.
- Soubory, umožňující pouze přidávat data – append-only.
- Změna velikosti svazku za běhu – AIX jfs, Tru64 advfs, ext[34]fs ...

Virtual file system



Správa logických svazků

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

Struktura LVM

- 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

Roura

- Datový kanál – zasílání proudu dat mezi procesy.
- Implementace – kruhový buffer velikosti `PIPE_BUF`.
- Čtecí konec, zápisový konec (deskriptory).

Nepojmenovaná roura

pipe(2) — Vytvoření roury

```
#include <unistd.h>

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

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

Pojmenovaná roura

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

Vlastnosti roury

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

Příklad použití roury – I.

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

Příklad použití roury – II.

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

Signály

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

Reakce na signál

signal(2) Nastavení reakce na signál

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

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

Zaslální signálu

kill(2), raise(2) — Zaslání signálu

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

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

- `pid > 0` zaslán procesu s číslem `pid`.
- `pid == 0` zaslán procesům ze stejné skupiny.
- `pid < 0` zaslán procesům ze skupiny `abs(pid)`.
- `pid == -1` nespecifikovaný výsledek (obvykle všem procesům).
- `signo == 0` – jen testuje zaslání signálu (viz EPERM vs. ESRCH).

Čekání na signál

pause(2) Čekání na signál

```
#include <unistd.h>

int pause();
```

Úkol:

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

Dostupné signály – I.

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

Jméno	Popis	Std.	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

Dostupné signály – II.

Jméno	Popis	Std.	Akce
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

Dostupné signály – III.

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

Vlastnosti signálů

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

Spolehlivé signály

- Vygenerování signálu – v okamžiku volání `kill(2)`.
- Doručení signálu (delivery) – vykonání reakce na signál.
- Čekající signál (pending) – stav mezi vygenerováním a doručením.
- Blokování signálu – odložení doručení. Signál zůstává ve stavu pending dokud proces nezruší blokování nebo nenastaví reakci na ignorování.
- Signál vygenerován vícekrát – v původním rozhraní se mohl doručit jednou nebo vícekrát. Novější systémy: fronta signálů (queued signals).
- Restartování služeb jádra – místo `EINTR` (přerušitelné služby).

Množiny signálů

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

sigsetops(3) Operace nad množinou signálů

```
#include <signal.h>

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

Zablokování signálu

sigprocmask(2) Blokování signálů

```
#include <signal.h>

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

Hodnota parametru how:

- SIG_BLOCK – sjednocení původní množiny a set.
- SIG_UNBLOCK – průnikk původní množiny a doplňku set.
- SIG_SETMASK – nastavení na set.

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

Dotaz na čekající signály

sigpending(2) Dotaz na čekající signály

```
#include <signal.h>

int sigpending(sigset_t *set);
```

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

Čekání na signál

sigsuspend(2) Čekání na signál

```
#include <signal.h>

int sigsuspend(sigset_t *set);
```

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

Reakce na signál

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;
}
```

- `sa_handler` může být i `SIG_IGN` nebo `SIG_DFL`.
- `sa_mask` – signály, které mají být zablokovány během provádění handleru.

Příznaky struktury sigaction

- SA_NOCLDSTOP – pro SIGCHLD: ne při pozastavení, jen při ukončení.
- SA_ONESHOT (nebo SA_RESETHAND) – jednorázová instalace ovladače. Pak zpět na SIG_DFL.
- SA_ONSTACK – použít alternativní zásobník (viz sigaltstack(2)).
- SA_NOCLDWAIT – pro SIGCHLD: proces nečeká na potomky a potomci nevytvářejí zombie.
- SA_NODEFER (nebo SA_NOMASK) – během provádění ovladače není zablokováno doručení stejného signálu.
- SA_RESTART – restartuj případnou přerušitelnou službu jádra namísto chyby EINTR.