Příkazy 1 kapitola

Služby jádra 2 kapitola

Knihovní funkce 3 kapitola

$ strace program 🡪 vypisuje služby jádra co program používá

CC - GNU C/C++ nebo LLVM/Clang

Spouští další programy – cpp(1), comp/cc1, as(1), ld(1).

Start kompilace se řídí příponou souboru.

\*.c \*.h **cpp** --> \*.i **cc1** --> \*.s **as** --> \*.o +lib\*.a **ld** --> a.out

cpp – preprocesor, obsahuje údaje začínající křížkem s čísly – čísla řádků kódu pro kompilátor, posílá data na standarní výstup, nedělá žádný soubor

cc1 – compiler, do asembleru, dělá optimalizace (nahrazení volání, rozvinutí cyklů, xor se sebou = nula)

as – z assembleru převod do strojového kódu

ld – linker (loader), nahrazení referencí za statické knihovny, tam kde je adresa neznámé funkce se dá její podpis a ten se linkerem pak narazuje a hledá (u statických knihoven)

o.soubory jsou závislé ne platformě ne jazyku – lze mixovat různé jazyky zkompilované do .o

-E – jen preprocesor

-S – do assembleru

-c – do strojového kódu

-o *jméno* – jméno výstupního souboru, bez implicitní koncovky

parametry preprocesoru, lze i programu cc

-D*makro*

-D*makro*=*hodnota* – nadefinuje makro pro preprocesor.

-U*makro* – ruší definici makra.

-I*adresář* – adresář pro hlavičkové soubory.

-I- – vypíná standardní adresáře (/usr/include).

kompilátor

-O[*číslo*] – zapíná optimalizaci, 0 - default, none, 2-normal

-g – zapíná generování ladících informací (názvy proměnných aby poznal místo adres apod)

-p – profilovací informace pro prof(1).

-pg – profilovací informace pro gprof(1).

optimalizace může např předpokládat, že proměnné se neaktualizují pokud poslány funkci, a tak to co má v registrech nemusí aktualizovat z paměti -> může rozbít program, pokud takovou optimalizaci povolíme ale nedodržíme toto v programu

Parametry linkeru

-L*adresář* – adresář pro knihovny.

-nostdlib – bez standardních knihoven.

-l*knihovna* – přidá soubor lib*knihovna*.a, případně .so.

--static – statické linkování.

--shared – sdílené knihovny.

-s – odstranit tabulku symbolů.

char znak;

int funkce(int argument) {

int cislo;

/\* ... \*/

}

Text – vlastní strojový kód (obvykle jen pro čtení/provádění, pevná délka). &funkce.

Data – čtení i zápis (globální a statické proměnné inicializované). &znak

Zásobník – čtení i zápis, zvětšuje se obvykle směrem k nižším adresám. &cislo, &argument

BSS – Data sekce pro neinicializované proměnné (a ty nastavené na nula) – pak jen řekne datové sekci o kolik se má zvětšit

Na zásobníku – automatické, deklarované uvnitř funkce.

V datové části – static nebo mimo funkce.

int jezek;

void funkce() {

int ptakopysk;

static int tucnak;

/\* ... \*/

Statické – static – pokud je použito mimo funkce tak pouze viditelnost v modulu (ne mimo soubory, to chce **extern**), jinak uvnitř funkce proměnná žije celou dobu a uchovává si hodnotu

Globální – mimo funkce a bez static – viditelné ze všech modulů.

MAKE (alternativně autotools, cmake, scons)

Soubor makefile nebo Makefile, řídí kompilaci více modulů

-f *Makefile* – soubor místo makefile nebo Makefile

-i – ignoruj chyby

-n – vypiš příkazy, ale neprováděj

-k – pokračuj i po chybě

PROMENNA=cc -s deklarace proměnné

CC = $(PROMENA) použití proměnné

**$@** - název souboru který se právě generuje

*cíl*: *zdroj…*

ze kterých zdrojů se vyrobí příslušný cíl, implicitní závislosti podle přípon

**Příkla**program.o: program.c program.h

Akce: odsazena tabulátorem, s mínusem na začátku – ignoruje chyby

$(CC) -c program.c

CFLAGS=-O2

LDFLAGS=-s

all: program

clean:

-rm \*.o a.out core

program: modul1.o modul2.o

$(CC) -o $@ modul1.o modul2.o $(LDFLAGS)

@echo ”Kompilace hotova.”

modul1.o: modul1.c program.h

modul2.o: modul2.c program.h

$(CC) -c $(CFLAGS) modul2.c

**nm(1) program/obj soubor**

Výpis tabulky symbolů programu (čísla funkcí atd, U -undefined, pro linker v obj souboru)

**strip(1) program**

Odstranit tabulku symbolů

**size(1)** objfile**.o**

Velikost objektového souboru

text data bss dec hex filename

20 3 0 23 17 x.o

**objdump(1) -h** objfile**.o pro ELFové soubory** (CUSTOM SEKCE)

Výpis informací nejen z objektového souboru (\*.o), u něj výpis všech sekcí a co v nich je – text, data…

Funguje i jako disassembler.

KNIHOVNY

Sada funkcí a proměnných s pevně definovaným rozhraním.

Definice rozhraní – *hlavičkový soubor*.

Umístění – adresáře /lib, /usr/lib.

Multiarch systémy – například lib64.

Statické versus sdílené – linkované za kompilace/běhu.

Statická knihovna – linker najde implementace a přihodí to do programu, cross-reference na adresy dodaných funkcí, větší – kopie všech funkcí, I v paměti

Statická knihovna – archív objektových souborů.

Linker – vytáhne z knihovny \*.o soubory.

Spustitelný soubor – obsahuje kopii \*.o z knihovny.

Staticky linkovaný program větší než dynamicky linkovaný, neumí sdílet kód s jinými programy, ale je v podstatě nezávislý.

**ar(1) rcs libSMTHING.a file1.o… komprese souborů (knihovna statická – kolekce .o souborů)**

**ar t /usr/lib/libc.a** rozbalit

umí se I nahradit jeden soubor v archive, ale nelze z proudu dat (tar umí proud, ale musí se celý rozbalit a zabalit)

**ranlib(1) soubor.a** index (tabulka) ve všech souborech v archive – rychlejší vyhledávání, totéž co **ar -s**

Dynamicky linkované knihovny/moduly

kód, přičleněný k programu až po spuštění

sdílené knihovny (při zavádění) nebo plug-iny (až když se volá)

Dynamický linker – /lib/ld.so

první načtená dynamická **knihovna**, stará se o přičlenění dalších knihoven, natvrdo zavede a po spuštění už dělá dynamičnost

Konfigurovatelný pomocí proměnných prostředí

LD\_LIBRARY\_PATH – seznam adresářů, oddělený dvojtečkami. Kde se budou hledat dynamicky linkované knihovny.

LD\_PRELOAD – objekt, který bude přilinkován jako první. Např. pro předefinování knihovní funkce. . Tedy lze podstrkovat vlastní implementace knihoven.

**U set-uid a set-gid programů dynamický linker ignoruje výše uvedené proměnné – díra v security**

/etc/ld.so.conf – globální konfigurace, obsahuje seznam adresářů s knihovnami

/etc/ld.so.conf.d/ – usnadnění práce správcům balíčků, tam si programy dávají vlastní cesty k adresářům s knihovnami (a mažou)

ldconfig(8) – generuje symlinky podle verzí a cache – které knihovny použít (místo toho aby to pokaždé hledal)

KOMPILACE

Linux libc4 (a.out), SunOS 4, SVr3

Umístění – na pevně dané adrese v adresním prostoru procesu. Při linkování se uvedlo na jaké adrese bude knihovna v paměti a její funkce, nevýhoda je že se musí dodržovat adresa, kde je funkce – problem compatibility + duplikace kódu (vice programů, každý to má jinde NEBO globální autority)

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.

ZA BĚHU

Extended Loadable Format

AT&T System V Release 4, Linux libc5+

Křížové odkazy – řešeny v době běhu. Rozhodne kde bude knihovna a pak ji teprve natáhne.

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 -více verzí stejné funkce

Nevýhody – pomalejší start programu, potenciálně pomalejší běh PIC kódu (je nutno alokovat jeden registr jako adresu začátku knihovny). Text procesu nemůže být read only.

Problém – nesdílitelné části kódu (křížové odkazy). viz též prelink(8) – spouští se periodicky a vybírá preferovanou adresu – když už tam knihovna je super, sdílí

ldd [-dr] program

formou: nazevknihovny => /path/to/lib

-d Provede doplnění křížových odkazů a ohlásí chybějící funkce.

-r Totéž, případné chyby hlásí nejen u funkcí, ale i u datových objektů.

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.

Ladění programu

Symbolické ladící informace – přepínač -g u kompilátoru.

Ladění na úrovni assembler – pouze adresy s hodnotami

Soubor core – obraz paměti procesu v době havárie. Nejčastěji v prac. adresáři soubor core-<pid>

Lze vytvořit i uměle například zasláním signálu SIGQUIT (Ctrl-\). Slouží k posmrtné analýze programu.

Pozor na **ulimit -c** může zrušit vygenerování souboru coredump pokud je moc velký

Pozor na **systemd-coredump(8)** (převezme generování core a spravuje to) a **coredumpctl(1)** (tahá jednotlivé core dumpy zpracované tím prvním)

Ladění běžícího procesu probíhá přes službu jádra ptrace(2) s pomocí souborového systému /proc.

**/proc** – adresáře procesů (process pid 1563 má adresár /proc/1563/ soubory **fd** (otevřené soubory), **mem** (obraz paměti)) **/proc/self** odkaz na sebe – kdo volá dosatne svoje číslo

gdb – GNU debugger. Nejrozšířenější možnosti (volání funkcí z programu, změna volací sekvence na

zásobníku, atd.). Le spouštět s core\* souborem nebo přímo and programme (přeloženým cpp -g)

b funkce() nastavení breakpointu

r a b c běh s hodnotami argv a b c

c krok

where kde spadnul

xxgdb – grafický front-end pro gdb(1).

Unix – normy jak se má co chovat

ANSCI C – základní knihovny pro c, bez konceptů procesu

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

Vlákna

Single Unix Specification – vyvinul se z POSIXu – toto je UNIX

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.

Normy:

X/Open XPG3,4: X/Open Portability Guide – rozšíření POSIX.1.

FIPS 151-1 a 151-2 – Federal Information Processing Standard; upřesnění normy POSIX.1.

SVID3 – System V Interface Description – norma AT&T (popisuje SVr4)

SVID4 – zahrnuje POSIX.1 1990.

BSD – označení pro extenze z 4.x BSD.

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?)

ANSCI C

Všechny při kompilaci, z hlavičkových knihoven stdlib

<limits.h>: INT\_MAX, UINT\_MAX, atd.

<float.h>: podobné limity pro reálnou aritmetiku.

<stdio.h>: – konstanta FOPEN\_MAX.

DETEKCE POSIX

#define \_POSIX\_SOURCE nebo (když se chci zeptat na verzi)

#define \_POSIX\_C\_SOURCE 199309 (když chci vynutit verzi)

#include <unistd.h> (hlavní knihovna POSIX)

Konstanta \_POSIX\_VERSION pak určuje verzi normy POSIX, kterou systém splňuje:

Nedefinováno – systém není POSIX.1.

198808 – POSIX.1 je podporován (FIPS 151-1) atd.

**sysconf(2) runtime limity mají podrtžítko \_NAZEV**

#include <unistd.h>

long sysconf(int name)

Globální limity.

Počet argumentů příkazové řádky. Počet dostupných procesorů. Velikost stránky. Frekvence časovače.

... a další.

**pathconf(2)**

long pathconf(char \*path, int name);

long fpathconf(int fd, int name);

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

**Limity: sysconf/pathconf**

**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.

Start programu

Linkování programu – crt0.o – spouští se, má definovaný main, 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 – dohledá závislosti

Inicializace – například konstruktory statických proměnných v C++. V GCC voláno z funkce \_\_main.

Nastavení globálních proměnných (environ).

Volání funkce main().

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

argc – počet argumentů programu + 1.

argv – pole argumentů.

Uložení stavu procesu do argv[] – nejčastěji přepsáním argv[0]. Nutné u programů, které

akceptují heslo na příkazové řádce. Zapíše tam co dělá – jde vidět z venku.

Platí argv[argc] == (char \*)0.

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

Při ukončení procesu je návratová hodnota vrácenarodičovskému procesu.

8-bitové číslo se znaménkem 0 – úspěšné ukončení.

Nenulová hodnota – chyba.

Ukončení procesu – návrat z main() (do crt0 a ten převezme hodnotu a skončí) nebo \_exit(2).

**exit(3)**

#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 pomocí \_exit(2)

**atexit(3)**

#include <stdlib.h>

int atexit(void (\*function)(void));

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

**\_exit(2)**

#include <unistd.h>

void \_exit(int status);

Služba jádra pro ukončení procesu. Je volána například z knihovní funkce exit(3). Opravdové ukončení co je v paměti, to zabije. Sdílené věci zůstanou, co měl otevřené to jádro pozná a uklidí.

**abort(3)**

#include <stdlib.h>

void abort(void);

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

Argumenty

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).

**getopt(3)**

#include <unistd.h>

int getopt(int argc, char \*\*argv, char \*optstring);

extern char \*optarg;

extern int optind, opterr, optopt;

while((c=getopt(argc, argv, ”ab:--”))!=-1){

switch (c) {

case ’a’:

opt\_a = 1;

break;

case ’b’:

option\_b(optarg);

break;

case ’?’:

usage();

}

}

b: akceptuje hodnotu -- akceptuje --

#include <getopt.h>

int getopt\_long(int argc, char \* const argv[], const char \*optstring, const struct option \*longopts, int \*longindex);

**Chybový stav služeb jádra**

Služba jádra – v případě chyby vrací -1 nebo NULL.

Důvod chyby – v globální proměnné errno:

**errno**

#include <errno.h>

extern int errno;

Hodnoty konstant v errno(3) nebo <sys/errno.h> (popř. <linux/errno.h> ). EAGAIN, EACCESS, EINVAL

**perror**

#include <stdio.h>

void perror(char \*msg);

if (somesyscall(args) == -1) {

perror(”somesyscall() failed”);

return -1;

}

Pro ENOENT vypíše: somesyscall() failed: No such file or directory

**Hodnota errno platná jen do příští chyby.**

Získání chyby

#include <string.h>

char \*strerror(int errnum);

int strerror\_r(int errnum, char \*buf, size\_t len); kopie hlášky aby nebyl sqfault

**Proměnné prostředí**

Environment variables. Předání argumentů programů co spouští programy co spouštím já

**Pole řetězců tvaru *jméno*=*hodnota*.**

Třetí argument funkce main() ... nebo přes globální proměnnou environ.

**Funkce musí mít makro před inclide \_POSIX\_C\_SOURCE 200112L**

**getenv(3)**

#include <stdlib.h>

char \*getenv(char \*name);

**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*.

**unsetenv(3), clearenv(3) Rušení proměnných**

#include <stdlib.h>

int unsetenv(char \*name);

int clearenv();

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

**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. Pokud by bylo nezarovnané na sběrnici tak buď spadne nebo pomalé – proto automaticky zarovnává, např struct (char, int) nemá velikost 1 + 4 ale je to zarovnané tak aby int nebyl na liché adrese

**calloc(3) Alokace pole**

#include <stdlib.h>

void \*calloc(size\_t nmemb, size\_t size);

Místo pro nmemb objektů velikosti size. Inicializováno nulami. (a zarovnáno – např pole prvků velikosti 3 na 4)

**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)**

#include <stdlib.h>

void free(void \*ptr);

Pozor: Některé systémy neakceptují free(NULL) – žádná akce

**alloca(3) Alokace na zásobníku**

#include <alloca.h>

void \*alloca(size\_t size);

Po ukončení funkce je automaticky uvolněno. Nesmí se použít pointer po návratu z funkce která to udělala. Specifické pro kompilátor. Nelze použít free(3).

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

#include <unistd.h>

int brk(void \*end\_of\_data\_segment);

void \*sbrk(int increment); relativní posun

Nastavení velikosti datového segmentu.

Používáno například funkcemi typu malloc(3).

Většina implementací malloc(3) neumí vracet uvolněnou paměť zpět operačnímu systému. Může tam na konci být nějaká hodnota – nelze uvolnit, nevadí – stránky se přastanou používat a vyswapují se.

Malloc ještě rozlišuje velké a male požadavky – velký udělá jako nový soubor v /proc verituální paměti, ten pak jde celý uvolnit

**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.Zabije ve chvíli kdy se dělá chyba, ne o x instrukcí dál (šlape do prázdna po free a pod) – zabít tam kde spadl, problem na vše nová stránka – nemusí doběhnout

Valgrind Vestavěné kontroly v GNU libc

**Nelokální skoky**

Podobné jako goto *(OMG, rychle pryč!* **^\_~***)*.

Ukončení vnořených funkcí. Například v případě fatálních chyb. Skok ven.

**setjmp(3) Inicializace skoku**

#include <setjmp.h>

int setjmp(jmp\_buf env);

Inicializuje návratové místo, kam se budu chtít vrátit. Vrací nula (při longjmp goto sem a vrací retval)

**longjmp(3) Nelokální skok**

#include <setjmp.h>

void longjmp(jmp\_buf env, int retval);

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

Struktura jmp\_buf – návratová adresa, vrchol zásobníku, smyslné užití jen do doby než se funkce kde

se tvoří neoddělá ze zásobníku – tohle řeže zásobník a kdyby se to mělo udělat až po skončení funkce, takseřízne zásobník (skočí ven) špatně

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…, -> dll soubory můžou pocházet z jiných jazyků třeba c++ a potřebují volat konstruktory což C nezná)

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 programům / objektům.

Může to být oběma směry – program hledá funkce v objektu/object si zaregistruje funkce na zavolání při události (dojde přes HTTPS wavefront object – zavolej mě na zpracování)

**dlclose(3) Uzavření dynamické knihovny**

#include <dlfcn.h>

int dlclose(void \*handle);

Počítadlo použití – poslední dlclose opravdu uzavře.

Zavolá symbol \_fini (destruktory, ...).

**dlsym(3) Získání symbolu z knihovny**

#include <dlfcn.h>

void \*dlsym(void \*handle, char \*symbol);

chceme získat funkci / globální proměnnou z dl

**dlerror(3) Chybové hlášení libdl**

#include <dlfcn.h>

char \*dlerror();

poslední selhaná funkce

program dost osekaný, neobsahuje testování chybných proměnných:

#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);

}

**Lokalizace**

Přizpůsobení národnímu prostředí. Bez rekompilace programu.

Možnost nastavovat na úrovni uživatele. Možnost nastavovat různé kategorie – např: format papíru český, jazyk dánština, chybové hlášení systému anglicky

Dva kroky: internacionalizace I18N – aby uměl pracovat s vice lokalizacemi

lokalizace L10N – výběr jednoho lokalu který používat

**Kategorie lokalizace: environment variables locale.h**

LC\_COLLATE – třídění řetězců.

LC\_CTYPE – typy znaků (písmeno, číslice, nepísmenný znak, převod velká/malá písmena, zda 0 = 49, dotaz isnumeric(“a”)… ctype.h).

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

... a další.

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

Jazyk – dle ISO 639 (pro nás cs)

Teritorium – dle ISO 3166 (pro nás CZ)

Znaková sada – například (ISO8859-2 nebo UTF-8)

Modifikátor – například (EURO)

**Příklad: Názvy locales** cs\_CZ.UTF-8, cs, cs\_CZ, en\_GB, de@EURO

LANG – implicitní hodnota pro všechny kategorie, když chybí např LC\_TIME tak vezme z LANG

LC\_\* – nastavení jednotlivých kategorií.

LC\_ALL – přebíjí výše uvedená nastavení pro všechny kategorie.

**Konfigurace lokalizace**

**setlocale(3) Nastavení lokalizace**

#include <locale.h>

char \*setlocale(int category, char \*locale);

Pro locale == NULL jen vrátí stávající nastavení.

Pro locale == ”” nastaví hodnotu podle proměnných prostředí.

Po startu programu je nastaveno locale „C“ (angličtina, ASCII). Program by měl po startu volat následující funkci:

setlocale(LC\_ALL, ””); vezme všechny LANG, LC\_\*, a nastaví knihovnu podle nich

**strcoll(3) Porovnávání řetězců podle locale**

#include <string.h>

int strcoll(const char \*s1, const char \*s2);

Jako strcmp(3) – bajt po bajtu, ale bere ohled na LC\_COLLATE. Vrací jedna pokud je první větší.

**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). – převede do mezitvaru

Stačí porovnat byte po byte (jinak český by musel třeba procházet i 4x) Je-li třeba alespoň len znaků, je hodnota dest nedefinována.

**Katalogy zpráv – překladové soubory**

Pro kategorii LC\_MESSAGES.

GNU gettext – překladové tabulky, vyhledávání řetězců. Vytahá řetězce a pak se to kompiluje do binární podoby – aby to při běhu nemusel parsovat.

Zdrojové soubory: .po.

Zkompilované soubory: .mo.

#. ../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 nainstalované možnosti všech nastavení**

aa\_DJ

...

zu\_ZA.utf8

$ **locale charmap 🡪** UTF-8

$ **locale mon 🡪** leden;únor;březen;duben;květen;...

$ locale abmon 🡪 Led, úno, bře…

**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 – ořezat diakritiku nebo //IGNORE. Podobná funkce pro shell: iconv(1).

echo ježeček | iconv -f utf-8 -t us-ascii//TRANSLIT (ascii nemá diakritiku by vyhodil err)

**localedef(8) Definice locale**

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

Vytvoří binární podobu locale (souboru s překlady) pro přímé použití v aplikacích.

UNIX – vlastnosti jádra

**Start systému – firmware.**

Uloženo v paměti ROM.

Na PC odpovídá BIOSu.

Test hardware - funguje paměť? Ukládá data? Jak velká? Test sběrnice jaká tam jsou zařízení, testy cache, ram…

Zavedení systému z vnějšího media, někdy I grafické nebo přes sériovou linku, nicméně většinou formware natáhne pouze primární zavaděč – rozhlédne se a rozhodne ze kterého disku boot, natáhne malý kus programu – zavaděč.

Často poskytuje příkazový řádek (PROM monitor).

Sériová konzola?

**Primární zavaděč systému**

Program v boot bloku disku – sedí v prvním bloku / diskové oblasti

Pevná délka, malý, natáhne do paměti a spustí

Zavádí sekundární zavaděč (který umí číst disky a souborové systémy, RAID pole)

Na disku: v master boot record – kam se musí vlízt spolu s tabulkou oblastí.

**Sekundární zavaděč systému**

Načítá jádro, nainicializovat a předat řízení

Předává jádru parametry – má soubor kde to má uloženo aby se to nemuselo dělat pořád ručně.

Některé poskytují příkazový řádek. Některé umí číst souborový systém (možnost bootovat libovolný soubor).

Používá firmware k zavedení jádra – jeho služby (BIOSU..) – musíme bootovat z disku kterému ten firmware rozumí a umí ho zpřístupnit / kterému firmware rozumí ten zavaděč

GRUP – má různé custom pluginy pro různé souborové systémy, jako malý OS, umožňuje vybrat odkud natáhnout jádro atd..

Jádro má main, lze mu předat argumenty “z cmd”

**Start jádra: parametry jádra**

Jádro převezme řízení a zruší paměť zavaděčům

Musí vědět kde je Systémová konzola – kam psat hlášení

Musí vědět kde je Kořenový disk – který disk připojit jako kořen, odkud spustit init

Parametry pro ovladače zařízení.

Ostatní parametry předány do uživatelského prostoru.

bootparam(7) (-> manuálová stránka)

když nerozumí paramteru, předá to init() -> první userspace program

**Průběh inicializace jádra**

Virtuální paměť – co nejdříve (Linux < 2.0 to neměl – jádro nepotřebuje hned mít virtuálku vidí přímo ty fyzické adresy, každý driver má inicializační funkci – zavolá se, rozhlídne zda umí zpracovat nějakej HW když ne jedem dál / řekne vidím nový zařízení, v linuxu 2.0 měly parameter na konec dosud obsazeného jádra -takto se zabíral prostor - vs. moduly problem ty by pak musely mít dvě inicializační funkce, jednu jako to bývalo dřív pro start a druhou v runtime 🡪 nepraktické, jede se standardně prvně virtuálka a každý žádá o paměť).

Inicializace konzoly (někdy dvoufázová: early\_printk() -> Do určité faze používá firmware zařízení, než si doinstaluje ovladače)

Inicializace CPU – režimy napájení, ochrany, pak se jede na sběrnici

Inicializace sběrnic (PCI již podpora autokonfigurovace – sama ví co má za zařízení, dříve musel poslat příkaz -zapiš a ozvi se - a pokud odpověděl tak ano, asi tam je to zařízení – typické porty pro určité typy zařízení, toto zařízení mohlo zkusit použít svoje porty tam už sedí jiné - zasekat pc).

Inicializace zařízení. – od sběrnice se inicializují zařízení

Vytvoření procesu číslo 0 (má názvy: idle task, swapper – přes něj se i swapovaly procesy,scheduler), většinou už existuje – už něco dělal - jen se nějak nastaví struktury pro procesy jádra a přejmenuje – řekne toto je process 0 - idle,

Vice procesorů – vice idle tasků pro kadžý jeden, v linuxu je taky od něj požadováno aby běžel když nic jiného neběží

Start vláken jádra (kflushd, kswapd, ...). Vlákna jádra jsou samostané jednotky celou dobu běží v provilegovaném režímu, nainicalizuje si tolik vláken pro jeden typ /kflush../ kolik má procesorů

Inicializace ostatních CPU a start jejich idle procesů.

Připojení kořenového systému souborů.

Start procesu číslo 1 – obvykle /sbin/init.

... dále řízeno přes uživatelský prostor.

Jádro vidí paměť lineárně, nepotřebuje překlad stránek, linux 2.0 – jádro se někam natáhlo, každá další funkce dostala kde zatím končí jádro, a každý driver dostal adresu – od té adresy dál to může být jeho, musí vrátit kde skončí – ale problem – když se začaly zavádět moduly – zavádení za běhu > musel by mít dvě inicializační funkce – jedna s pamětí, jedna by musela žádat ->> virtualizace jako první, a driver žádají už standardně všechny o paměť

**Inicializace zařízení**

UNIX v7 – bloková/znaková zařízení, statické tabulky (bdevsw[], cdevsw[]). Seznam podporovaných zařízení – obsahuje pointer na funkce k obsluze (open, read, close)

Linux – bloková/znaková/SCSI/síťová zařízení, dynamické tabulky – jsou tam mezivrstvy už je to složitější není tam jen funkce read/write, např karta má jen 2 stavy a pokud dojde paket tak to oznámí jádru, write – karta řekne mohu posílat a předají se jí pakety, pak řekne done – mohou se pakety uvolnit z paměti 🡪 realizace pomocí init funkce driveru, ten si zaregistruje v těchto dynam. Tabulkách typy zařízení co umí a které mu má jádro předat když objeví

Obsluha zařízení – funkce pro otevření, čtení, zápis, řídící operace, atd. **Privátní data zařízení** = předává se jako první parametru funkcím zařízení, nainicializuje datovou strukturu pro každé zařízení když po něm něco chce, tak pracuje s jednou strukturou pro jedno zařízení – posílá privátní data jako parameter aby se vědělo s čím se pracuje)

Rekonfigurace za běhu – hot-plug/hot-unplug (USB apod.). drivery musí byt připraveny že může zařízení kdykoliv zmizet

**Bootování s ramdiskem – dočasný kořenový disk**

Obsah ramdisku načten sekundárním zavaděčem- před tím než se připojí svazek kořenový a svazek a spustí init- do paměti spolu s jádrem.

Jádro nemusí mít v sobě žádné ovladače kromě konzoly a souborového systému, který je na ramdisku. Dneska je plno různých zařízení – minimalisticky potřebujeme jen ovladač konzoly, kořenového disku a jeho filesystému.

Inicializace a přilinkování modulů – jádro nemusí umět hledat moduly (proč by mělo, pak poběží pc a jádro je zbytečně velký protože obsahuje složitosti s hledáním modlů co použije jen při startu), nechá se to na ramdisku který musí umět různé typy disků ([RAID](https://cs.wikipedia.org/wiki/RAID), [LVM](https://cs.wikipedia.org/wiki/Logical_Volume_Manager), [NFS](https://cs.wikipedia.org/wiki/Network_File_System)), disk může být šifrovaný… obsahuje složité funkce pro hledání modulů, detekce modelů v userspace

Případné odmontování ramdisku (dočasného kořenového disku)

Dále pokračuje start systému připojením kořenového souborového systému a spuštěním initu.

**Ramdisk v Linuxu**

Komprimovaný soubor – bývá to Obraz souborového systému nebo cpio(1) archív, jednoduše parsovatelné

Na něm je Startovací skript /linuxrc např shellový – podle toho co interpretuje ramdisk,

určení kořenového svazku…

Po ukončení – přemontování jako /initrd nebo zrušení.

**Bootovací zprávy jádra**

Příkaz dmesg(8) kruhový buffer kam zapisuje zprávy z bootu systému – přepisuje nejstarší

Uloženo ve /var/log/dmesg, /var/log/boot.msg nebo podobně

* Mapování virtuálky – reserver/usable, v kterých stránkách je initial ramDisk, které stránky nesmí swapovat, detekce NUMA – která paměť kterému procesoru, které stránky jsou v DMA zone (ne všude můře DMA sahat), rezervování ramky pro gpu, řafiče přerušení…

**Výroba a Konfigurace jádra**

System V – konfigurace jádra v /etc/system /etc/conf - dívá se na poslední verzi v tomto souboru a když je modifikováno tak při dalším bootu překompiluje na novou verzi (update)

jádro poskytováno jako .o soubory a pár malých .c souborů (např. soubor = definice jednoho pole – tabulky procesů s možností zvolit si konstantu - velikost, dalo se modifikovat na míru, pak zkompilovat, přilinkovat zbytek a máme jádro)

BSD - /sbin/config konfigurace jádra v souboru, pak vygeneruje BSD make do specifické složky (k dispozici i zdrojáky narozdíl od komerčního System V)

Linux – prostě používá make, (make config – interaktivní pgoram na ptaní parametrů), make all (kompilace jádra)), .config (volby se kterými byl zkompilován) /proc/config.gz (kompilace s volnou že to kernel umí o sobě říct – ale nutnost zapnout + zabírá místo)

Monolitické jádro

Jeden soubor na disku, všechny ovladače součástí jádra.

Často bez autodetekce zařízení.

Paměť dostupná všem částem jádra stejně. Každá funkce může volat jiné. Rychlé, ale bezpečnostní díry – dostanu pointer a přeteču, tak přepíšu paměť – v kernel horší.

Jedna část jádra (ovladač) může rozbít druhou.

Mikrojaderné systémy

CMU Mach, OSF Mach, L4, minix, Windows NT HAL, QNX, ...

Co nemusí být v jádře, dát mimo něj. Mikrojádro moc nedělá – ani swapování, ani zpráva zařízení…

Procesy (servery) pro správu virtuální paměti, ovládání zařízení, disků a podobně. Každý má vlastní kontext. Jádro prostě jen přepíná mezi servery a je komunikační kanál.

Dobře definovatelné podmínky činnosti (lépe se dokazuje korektnost, vhodné pro kritické SW jen teoreticky, protože: DMA, SMI, IOMMU a další problémy).

Předávání zpráv – malá propustnost, velká latence. Chci IO – přepnout kontext do kernelu, pak serveru IO, zpět kernel a spět přepnout do user space. Kopírování paměti – jádro musí kopírovat paměť mezi servery, navzájem k sobě nemohou,

Problém : může být malý OS (intel management interface) na sběrnici odchytává některé porty přemapovává vstupy – mapovat USB klávesnici na jiný typ – kompatibilita.. ALE sahá na to bez vědomí jádra (zpomaluje přístup, a jádro s tím nic neudělá)

Další problemy – DMA může přistupovat bez kontroly, existují driver co umí číst a přepsat paměť připojený ve spánku pc, nebo ovladač může přes sběrnici zapisovat data do paměti – přepsat někomu (není to úplně oddělené, proto se podepisují ovladače – je to největší díra jádra)

**Modulární jádro**

Části (moduly), přidávané do jádra za běhu (odpovídá dynamicky linkovaným knihovnám

v uživatelském prostoru). Pořád je to monolitické jádro ale už se dá rozhodnout co přidat a co ne, může se I rozhodnout zda bude modul v jádře nebo userspace.

Ovladače, souborové systémy, protokoly, …

Přidávání ovladačů pouze při startu systému – AIX, Solaris < 10.

**Definovaná rozhraní, nikoliv adresní prostor.**

Závislosti mezi moduly (depmod(8)).

Linux: umí dynamicky přidávat ovladače, pokud má sběrnice autokonfiguraci tak dokonce ovladač nemusí ani aktivně hledat zařízení, jen zaregistrovat čísla co umí zpracovávat u jádra a to když objeví zařízení tak to předá:

Dynamická registrace ovladačů: register\_chrdev(), register\_blkdev(), register\_netdev(), register\_fs(), register\_binfmt() a podobně.

Dohledávání pomocí identifikátorů sběrnice (např. PCI ID).

Procesy

Při startu – kontext procesu číslo 0 – později idle task.

Idle task nemůže být zablokován uvnitř čekací rutiny. v momentě kdy dostane řízení zavolá HALT do doby než procesor dostane přerušení - koukne kdo by mohl běžet – idle čeká a šetří baterku

**KONTEXT Stav systému, příslušný běhu jednoho procesu/vlákna**

(co je na registrech, čítač instrukcí, zásobník, nastavení procesoru…)

Přepnutí kontextu – výměna právě běžícího procesu za jiný.

Linux – struct task\_struct, k dispozici globální proměnná current – lze nastavit parametry a zavolat schedule() a tím třeba uspat

Kontext, pod kterým běží služba jádra:

UNIX: Běh pod kontextem volajícího procesu – má dva režimy: běží v režimu userspace/kernelspace. Zvýší oprávnění procesu a skočí do jádra a vykoná akci.

Mikrokernel : předá se řízení jinému procesu (serveru).

Nutno vyřešit přístup jádrem do user-space (např. pro write(2)). Chci po jádře aby zapsalo – musí umět načíst data co zapsat – uloženo v userspace

Přerušení

Např žádost o pozornost hardwaru – přerušení může přijít od HW že je ready data (paket) / něco se stalo (stisk klávesy). Má být přerušení přerušitelné?

NE – jednoduché, ale špatná odezva, (existují samozřejmě přerušení která nemůže nic blokovat – dělení nulou atd) ANO – problem složitosti

Obsluha – nepřerušitelná (rozdělit na 2 typy, viz níže) nebo přerušitelná s prioritami (bsd unix měl priority kdo koho může přerušit, obsluha nemůže přerušit sama sebe, nemůže přerušit příchod paketu příchod jiným – tohle bychom chtěli synchronně).

Horní polovina – co nejkratší, nepřerušitelná. Např. přijetí packetu ze sítě, nastavení vyslání dalšího packetu. Interrupt time. Zpracuje jen to co je nutné – řekne přišel paket a kam, řekne hardwaru příště mi zase oznam příchod a konec.

Spodní polovina – náročnější úkoly, přerušitelné. Obvykle se spouští před/místo předání řízení do uživatelského prostoru. Paket – tlusté zpracování – rozbalení, checksum, šifrování, firewall, jestli se má paket jen přeposlat…

Preemptivní (user space je vždy)/nepreemptivní jádro – může dojít k přepnutí kontextu kdekoli v jádře nebo se musíme vrátit do vykonávání přerušeného procesu? – tedy když nastane přerušení, asi nastalo protože se na něco čekalo, nemůže náhodou běžet jiný process který má vyšší prioritu než ten přerušený atd.. – nemusí nutně vrátit řízení po skončení obsluhy a může změnit kontext

**Nepreemtivní jádro** – nepovoluje přepnutí kontextu jádra – prostě se vrať do userspace a tam si to už pohlídá správa procesů, nemusíme řešit bezpečnost, ale zpoždění – přepnout userspace a pak zjistit že musí stejně měnit kontext na jiný process s vyšší prioritou co dostal paket

**Hybrid (fedora)** – lze si říct kde se dá přerušit a kde ne

**Preemitivní** – může dojít k přepnutí kdekoliv, ale musí se hlídat datové struktury aby se nerozbil

Pod jakým kontextem lze provádět přerušení?

Zvláštní kontext – nutnost přepnutí kontextu *🡪* zvýšení doby odezvy (latence) přerušení. Navíc jenutno případně mít více kontextů pro možnáparalelně běžící přerušení.

UNIX (ve většině implementací): Přerušení se provádí pod kontextem právě běžícího procesu podobně jako volání služeb jádra. Problémem je že nesmí se zapořítávat využitý čas obsluhou tomu procesu a hlavně: obsluha přerušení nesmí zablokovat proces. (tedy nemůže čekat, proces nemá s obsluhou logicky nic společného a jeho blokování je tedy chybou, např takto by mohl být zablokován idle task!)

Linux bez samostatného kontextu, uvažuje se o threaded handlers – vlastní vlákno

Realizace přerušení

v registru je adresa tabulky přerušení X – adresa do pole pointerů, vezme Xtý záznam a skočí na rutinu obsluhy, uloží stav zásobníku právě běžícího programu, může mít něco rozpracovaného – musí mu uložit registry (je v jeho kontextu) – přidá je na zásobník, tohle ještě ani nemusí být úplně jednoduché – může mít rozpracovaných vice instrukcí v mikrokódu – někde to rozsekne, něco dodělá a něco zahodí, další problem nemusí stačit zásobník pro přerušený program + přerušení, nemůžeme si dovolit swapovat pro zvětšení zásobníku 🡪 mívá vlastní malej zásobník

**Odložené vykonání kódu**

Funkce, vykonaná později (po návratu z přerušení,při volání scheduleru, atd.) Například odložené vykonávání spodní poloviny přerušení – nekritický přerušitelný kód. Jakým způsobem se dá odložit 🡪 tasklety, workqueues (přeskočil to)

**Virtuální paměť**

Virtuální adresa – adresa z hlediska instrukcí CPU.

Překlad mezi virtuální a fyzickou adresou – **stránková tabulka** (virtuální adresy procesu mají daný prefix který se přičte – mapa adres a k nim daný prefix pro přištění).

Každý proces má svoji virtuální paměť: **každý proces má svoji stránkovou tabulku**., i další údaje - špinavé stránky, zda je v paměti..

Výpadek stránky (page fault) – stránka není v paměti, stránkový adresář neexistuje, stránka je jen pro čtení a podobně. vyolá **výjimku**, podobné přerušení prostě mezi instrukce vsune skok na obsluhu page fault, tyto **jsou synchronní** - ví se která instrukce výpadek způsobila, přerušení je asynchronní může dojít vždy

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. Při úspěchu vzbudí process a zopakuje tu instrukci co to způsobila – process nic nepozná (až na uběhnutý čas).

Provedený překlad adresy se cachuje: TLB – asociativní paměť několika posledních

použitých párů *(virtuální adresa, fyzická adresa) a třeba I přístupová práva (read only code..)*. pro celý OS (není pro každý proces). Přepnutí kontextu – vyžaduje vyprázdnění TLB, v případě virtuálně adresované cache take vyprázdnění cache.

Přepnutí kontextu – vyžaduje vyprázdnění TLB, v případě virtuálně adresované cache také

vyprázdnění cache (to už je spíš historie). – znovu jít a načíst si překladová pravidla ze stránkové tabulky nového procesu

Přepnutí mezi vlákny je rychlejší

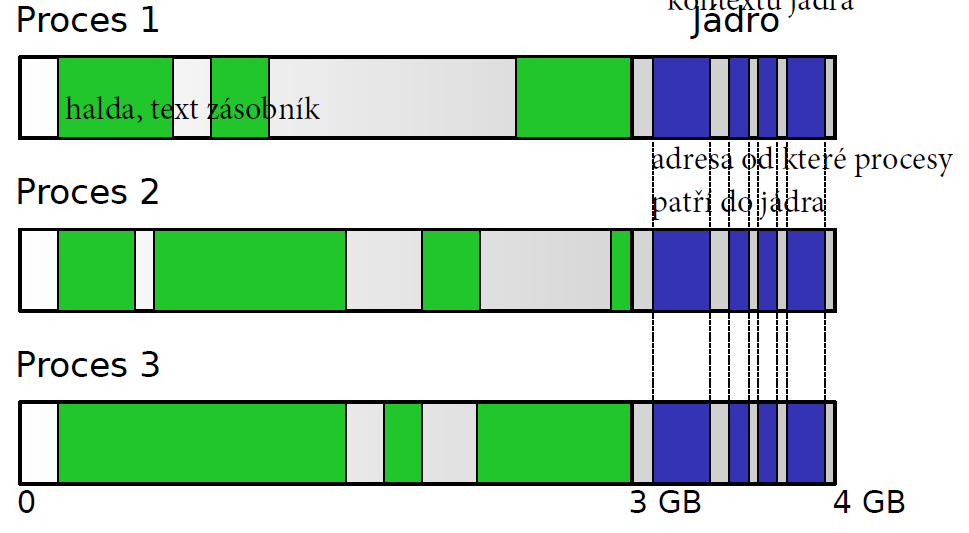
Softwarový TLB – OS-specifický formát stránkových tabulek. – vice tabulek, možnost pamatovat si pro pár procesů naráz

Lazy TLB switch – uvnitř jádra lze ušetřit – teprve při příchodu do userspace se musím podívat zda sem předtím nebyl jinde, jádro to nepotřebuje vidí fyzické

**Prostor jádra a uživatelský prostor**

do všech procesů je namapováno jádrom přepnutí do jádra znamená že začně vidět modré stránky

může být i instrukce pro zneplatnení rozsahu adres v TLB, když proces chce namapovat novou knihovnu měl by vylít TLB, ale nemusí - stačí zneplatnit jen sekci ne celé (I při přepnutí kontextu může vylít jen 0-3GB)

****

Virtuální paměť jádra – obvykle mapována na nejvyšších adresách.

Paměť jádra – mapována do všech procesů stejně.

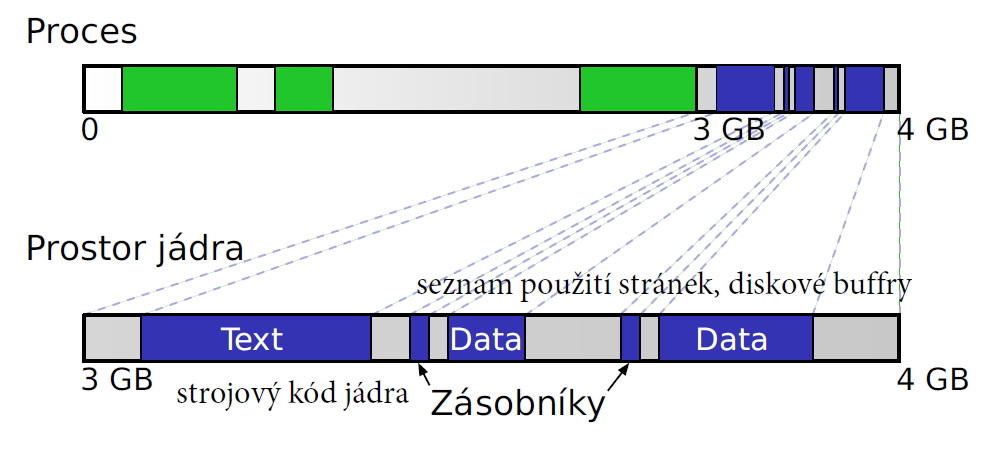
Přepnutí do režimu jádra – zpřístupnění horních (virtuálních) adres.

Alternativa – jádro má samostatnou VM – lze využít pro userspace 4GB (ale: TLB flush při volání jádra nebo přerušení); 🡪 4:4 GB split. (je to obrana proti meltdown)

Meltdown

ostranní kanaál úniku niformací - spekulární vykonávání – nemusí čekat na podmínku kam skočit a rovnou může vykonávat jedno větev, když se zjistí že šel špatně, vrátí se - což nejde úplně lehce, a hlavně pomocí měření délky vykonávání lze zjistit co stihl načíst do cache – lze vidět kudy se vydal a dokonce lze se rozhodnout co načíst z jaké adresy a uniknout ze spekulárního vykonávání 🡪 intel si dovolil odložit ověřování práv, dovolí si odložit ověřování práv na později – načte něco do cache co by nemělo být vidět, ve spekulárním vykonávání adresa do jádra nespadne a mohu stihnout to přečíst a pak se vydat jinudy (tedy nechat ho spekulárně vykonávat to co vím že pak stejně zahodí 🡪 normálně by to spadlo) z userspace vyčtu paměť jádra

Reálně se mapuje do všech procesů pouze kousek jádra, zavoláním služby jádra se domapuje zbytek – tradeoff mezi bezpečností a nutností přemapovávat celý jádro – stačí si přimapovat jen to co reálně potřebuje.



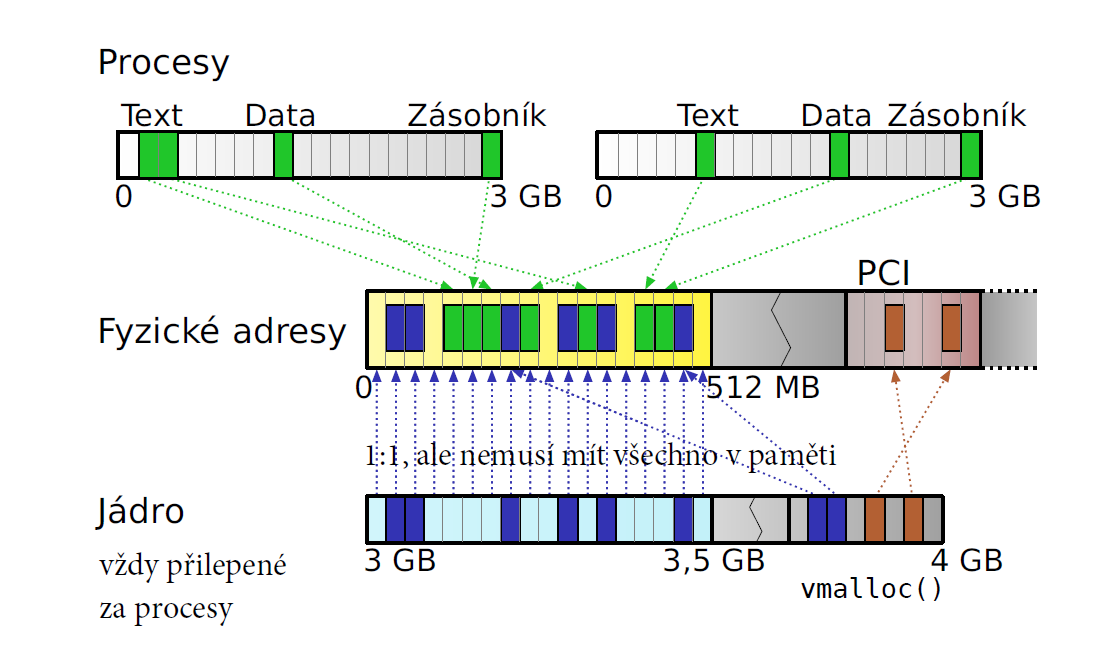
Zásobník v jádře – pro každý thread/kontext.

Linux – 1 stránka (4kb)/thread, nastavitelné 2 stránky(8kb)/thread

aby každý proces mohl přenout do jádra, víc procesorů - více kontextů naráz v jádře (nebo I s jedním jádrem může být zablokován a může běžet jiný -> každý kontext má vlastní zásobník)

Fyzická paměť – mapována také 1:1 do paměťové oblasti jádra (Linux bez CONFIG\_HIGHMEM).

Virtuální alokace – mohu chtít pro jádro virtuální alokaci: dočasné zpřístupnění fyzické paměti zařízení uvnitř jádra. Virtuální mapování paměti ram přes sběrnici do zařízení (hnědá), nebo kvůli tomu že chci spustit program se zásobníkem co je přes n stránek – chci n stránek vedle sebe (je to fyzické mapování), se nemusí povést – virtuálně si je tedy dám vedle sebe a mapuju je na víc fyzických adres



někde má rozsah virtuálních adres kde může dávat věci tak že se mu jeví souvisle -> n**áročnější práce, zneplatnění TLB na všech procesorech!, musí zarazit všechny procesory a přinutit je dogenerovat TLB – režie**

**Taky kvůli virtuálním adresám nemusí být schopnej 1:1 namapovat (a využít) všechny fyzické adresy, takže tam je omezení – některé se mohou např používat jen jako userspace storage apod**

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. Firewall překladu adres které vidí zařízení na reálné adresy – ochrana či nutnost, musíme zesouvislit pamět třeba pro GPU aby si mohla přečíst texturu..

Přístup do user-space: proces předá jádru ukazatel (např. buffer pro read(2)). A délku a čti – ale je to nepřátelské prostředí – jádro by mohlo použít neplatnou adresu (Robustnost – user-space nesmí způsobit pád jádra.) a nebo pracovat s adresou k sobě a uložit svoje data – si vyčíst paměť

Nutnost kontroly pointeru – projít všechny tabulky – lze pokud není vice procesorů – mezitím co hledáme se může změnit mapování a pak se to pokazí.

Validace před použitím? Problémy ve vícevláknových programech (přístup versus změna

mapování v jiném vlákně).

Přístup do uživatelského prostoru není možný uvnitř ovladače přerušení – nelze kopírovat data na adresu kam si program řekl že je chce – mezitím se mohlo přepnout a sedí tam jiný process / nemusí ta stránka ani být v ramce

Ošetřování validity pointeru z userspace

Normální běh – cca 10 instrukcí bez skoku. Používá ELF sekce – pro generování druhého toku instrukcí, specifikuje si oddělit hlavní tok a opravné, takže pokud je vše ok ani neskáče a projede to rychle.

Jádro musí být připraveno že dostalo špatný pointer – dělá se překlad adres pomocí právě existující TLB na fyzickou

**Tabulka výjimek** – adresa instrukce, která může způsobit chybu 🡪 skok na opravný kód (volání get data z pointer uloží do té tabulky ten pointer a co se má udělat pokud selže předtím, než se čte), když nenajde 🡪 spadl na adrese která NEPOCHÁZÍ z user space - error: interní chyba jádra

1. Porovnání s PAGE\_OFFSET (3 GB na 32-bitovémsystému). Je-li větší, chyba (odkaz do jádra)
2. Použití ukazatele – není-li platný, výjímka CPU.
   1. Obsluha výjimky – je adresa instrukce v tabulce výjimek? Ano: zavolat opravný kód.
   2. Jinak: interní chyba jádra (kernel oops).

Jak tedy poznat, že system spadl při volání funkce jádra z jádra (kernel sám sobě pošle socket) ? Nutnost oznámir pomocí set\_fs(KERNEL\_DS) 🡪 nemá se provádět kontrola 1) Porovnání s PAGE\_OFFSET

MULTIPROCESOROVÉ SYSTÉMY

**SMP** – symetrický multiprocesing první typy multiprocesorů – byla frontside sběrnice a každý processor si vždy vzal pro sebe, jednoduché ale synchronní omezení na práci s pamětí

**NUMA** – zapojení procesorů a sběrnic do nějaké hustší topologie, propojené numa topologií (ne sběrnicemi)

hierarchická paměť – z určitých CPU rychlejší přístup než z jiných, ALE principiálně to same jako SMP, každý může všude přistupovat (cc-NUMA – cache coherent, vždy zapíše do cache a když chce jiný processor nejdřív vylije a pak povolí zápis, prostě se to chová jako by tam byla jen jedna).

**Multipočítače (cluster)** – na částech systému běží zvláštní kopie jádra (clustery a podobně). Není možný přímý přístup do paměti všude – pomocí nějakých rozhraní/služeb.

Problémy – vlákna mají společnou paměť, zamykání a integrita sdílených dat, problem komu posílat přerušení z těch procesorů a cache ping-pong – jeden načte a druhý už řve vylít chci taky číst a pořád se tak hádají

Problém paralelismu (jak vice procesorů, tak jeden processor)

Na jednom CPU – v kterémkoli okamžiku může být proces přerušen a tentýž kód může provádět i jiný proces 🡪 manipulace s globálními datovými strukturami (alokace paměti, seznam volných i-uzlů, atd.).

JEDNO CPU – stačí zakázat přerušení (ale horší odezva) - instrukce cli() a sti(), sti tam nebývá aby funkce náhodou přerušení nepovolila když nemá (vnořené cli) – prostě si uložím flag, pak zakážu, a pak znovu anstavím flag ať už má povolené/zakázané

MULTICPU SYSTÉMY

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). Např zámek pro každý adresář s manipulací souborů

Zamykání kostrukce v SMP – nutnost atomických instrukcí (test-and-set) nebo detekce změny nastavené hodnoty (MIPS) na zámku. Zamčení sběrnice (prefix lock na i386)

SEMAFORY

Exkluzivní přístup ke kritické sekci

Určeno i pro dlouhodobé čekání, je uspán pokud je zamčeno

Lze volat pouze s platným uživatelským kontextem

Linux – up(), down(), down\_interruptible(). dva typy čekání - přerušitelné a ne - zámek generuje jedno z nich, (např u disků - je jednodušší počkat pár MS než ztrávit ten čas při context switch– čeká nepřerušitelně)

SPINLOCK

Krátkodobé zamykání používá se když nemůžeme blokovat (takto bychom zablokovali někoho kdo blokuje🡪 deadlock - např. přerušení, věci s právy jádra)

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.

spinlock - busywait čekání, atomický zámek, před vstupem zkusím zamčít, na konci odemču, aktivně zkouší zamčít - na kratke čekání, **busy wait spinloky musí v přerušení dělat zákaz přerušení**, jinak by jiní přerušení přerušilo přerušení držící lock -> může způsobit deadlock

R/W zámky

Paralelní čtení – exkluzivní zápis Linux – struct rwlock, struct rwsem. – má oba spinlock i semafor

Problémy – priority (nejčastěji se dává priorita na čtení -> by se zápis nedostal na řadu -> takže zase priority pro čtení – pokud dlouho nebylo, počkám až všichni čtenáři vypadnou a nové zablokuju) a upgrade r-zámku na w-zámek (nemůže, deadlock pokud se to stane vícekrát). – řešení: pustím zámek na čtení a dostanu zápis, znovu musím zkontrolovat že tam je něco co chci přepsat (mezitím se mohlo stát plno věcí)

Moc update málo čtení

Např počítadla na cpu počtu paketů – místo test&set atomického inkrementu počítadla na každém procesoru zvlášť, pak chce číst - součet

Read-copy-update (moc čtení málo update)

Atomické instrukce pro zámek jsou pomalé (stovky taktů; přístup do hlavní paměti).

Obvyklá cesta (např. čtení) by měla být rychlá. Proto je čtení bez zámku a kdo chce zapisovat tak musí **kopírovat strukturu, změnit, atomicky přehodit pointer a uvolnit starou (ale lze to udělat, když to stale někdo může ještě číst?..** kdy uvolnit starou verzi? Dá omezení na čtenáře – lze číst jen bez přepnutí kontextu, pak si musí aktualizovat pointer 🡪 po přepnutí na všech procesorech. Odložené vykonání funkce po splnění podmínky – řekne jádru tohle uvolnit po X přepnutích kontextu**)**

**Další problem je s přeuspořádáváním (**Slabě uspořádané architektury) – může nejdříve přehodit pointer a pak zapsat – uvnitř programu to logicky nemění nic, NAVEDEK ANO: Nutnost explicitních paměťových bariér (speciální instrukce CPU nebo direktivy kompilátoru, vice verzí).

**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 🡪 vice alokátorů

Alokace během přerušení – nesmí uspat process zákaz swapování (v přerušení) – vždy má bokem nějaké místo pro tento typ alokací (ale může dojít)

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 opakované alokace po jednom objektu:

Stejné zarovnání v cache (v cache se dávají obejkty na místo podle své adresy, podobná zarovnání = podobné adresy, vyhazují se z cache)

Zbytečné inicializace (inicializace a hned write)

Studené (cache-cold) objekty – alokovat před chvílí uvolněné objekty, ne nějakou random paměť

Zamykání při alokaci. - pomalé

Reakce na tlak ve virtuální paměti – čast dat co nevadí když se smažou – možnost uvolnit paměť pokud dochází (cache-hot) 🡪 jak poznat které lze smazat?

Řešení: SLAB Alokátor

Slab – struktura uvnitř stránky: metadata, objekty. Při alokaci nové stránky si ku zabere na metadata zbytek rozkouskuje na alokaci objektů

Volné místo – využito pro cache coloring. Rozprostře volné místo tak že mezi jednotlivými objekty je kousek prázný, objekty se do cache dávají na místo podle adres – podobné VM adresy se vyhazují, takto docílí toho že se uloží jinam, volné místo to posune o kousek vedle – změní adresu

Stav slabu – obsazený, částečně obsazený, volný. Procházíme ty volné když hledáme prostor – rychlejší.

VM pressure – uvolnění volných slabů má poznamenáno který slab alokuje cache – lze uvolnit

Paralelizace – částečně volný slab pro každý procesor. každý procesor má svůj slab, když vyčerpá lokální zásobu tak s globálním zámkem jde a získá nový slab

Cache-cold/hot objekty – lze specifikovat při alokaci i uvolnění. Nemusí se vymazat objekty, lze znovu použít, ušetřít inicializaci (u nových slabů lze předinicializovat - Konstruktor, destruktor (volitelné))

Linux – /proc/slabinfo - seznam všech slabů a informace o nich

lze detekovat i utíkání paměti počítá alokace dealokacepro jeden typ objektu

**Časovače**

Časovač – nutnost vyvolat přerušení po určité době -chci za 5 minut něco zpracovat a ještě si nastavím časovač (např 6 minut) abych poznal že se to nestalo (pokud se to stalo tak ho ukončím)

Funkce v Linuxu – add\_timer(), del\_timer().

Atributy – čas a funkce, která se vyvolá po vypršení času.

Zablokování procesu – current->timeout. (nastavení glob struktury vlastního procesu, bylo výše)

**Čekací fronty**

Wait queues – zřetězený 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* – někdy není vhodné probouzet všechny – přijde paket a mám X procesů co ho mohou zpracovat -> stačí probudit jeden ne aby se na tom všichni pobili a většina zase usnula 🡪 použití wake\_one() ALE musí být schopen paket zpracovat, ne že se probudí a zjistí že má ještě jinou událost a dá jí přednost

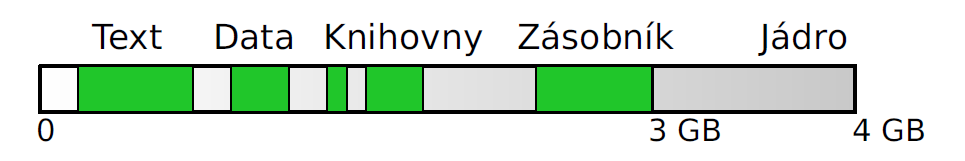
**Přepnutí kontextu – funkce schedule().**

**Procesy**

Proces – běžící program, kontext procesoru se samostatnou VM.

Vlákna (threads) – kontexty sdílející VM. vyrobením dalšího vlákna se stanou dvě vlákna, všechna jsou si rovna - ne jeden velký proces a kolem malá vlákna

**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 🡪 někdo přetypovával pointery na integer a zpět, u 32bit OK ale ne 64bit – první 4GB -32bit- adres je prázdných aby odchytili tyto chyby nic nás to nestojí adres je dost

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 atributů – např. programem ps(1).

Implementace – nad virtuálním souborovým systémem /proc nebo nad /dev/mem.

Atributy procesu jsou:

Stav procesu – viz graf, runnable, userspace, kernelspace, zombie…

Program counter – čítač instrukcí; místo, kde je proces zablokován (WCHAN).

Číslo procesu – PID. unikátní

Rodič procesu – PPID (rovno 1, pokud sirotek).

Priorita procesu – dvě verze

Vlastník procesu – (real) UID.

Skupina procesu – (real) GID – procesi co spolu souvisí (spojené rourou ls /dev/null | cat)

Skupina procesů, session – seskupování procesů do logických celků – všechny procesy na jednom terminálu

Reakce na signály, Čekající signály ps s

Časy běhu kolik spotřeboval, jak dlouho běží, jak dlouho v jádře..

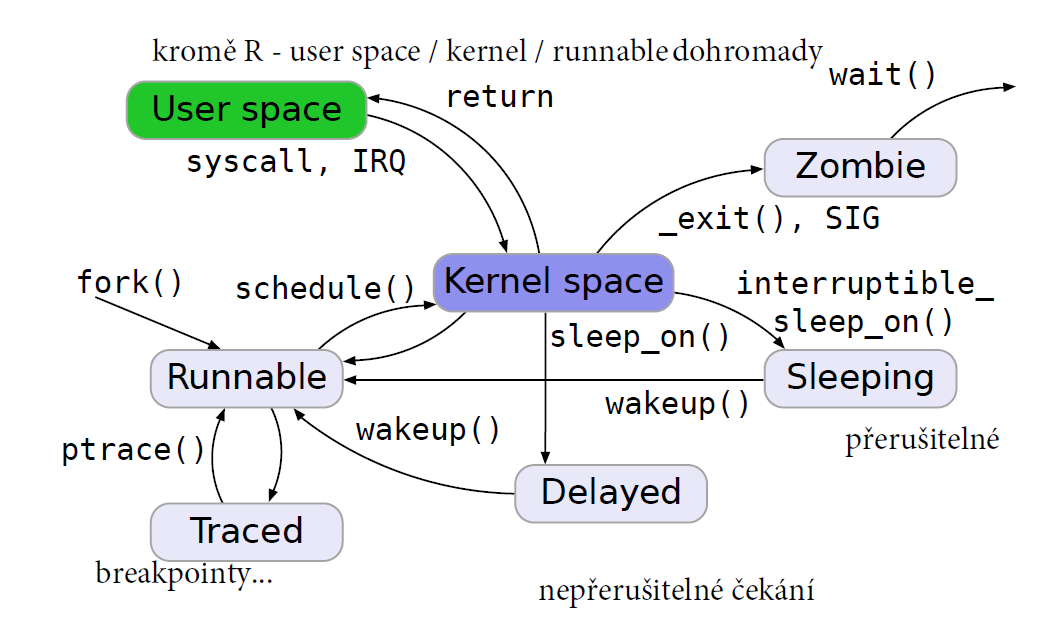
Pracovní a kořenový adresář /proc/self/root /proc/self/cwd

Tabulka otevřených souborů /proc/self/fd

Odkazy na potomky jak vyrovit sirotky (nezávislé programy) z rodiče? – nastavit rodiče na 1, proces má odkaz na následující potomky - > takhle projde všechny

Limity – na velikost souboru, max. spotřebovaný čas, max. počet otevřených souborů atd (setrlimit(2)).

Stavy procesu většinou zobie neuvidí - čeká se na dokončení a hned se předá a hodnota, pokud tam jsou tak je to spíše chyba



ps l seznam procesů, první písmeno sloupce je stav tohohle graf

**Služba jádra**

Kód definován v jádře, není možné tuto funkcionalitu si naprogramovat

Přepnutí oprávnění CPU – jen začne vidět i stránky jádru

Charkaterizována svým číslem – většinou je daná volací sekvence – do registru dej číslo služby, do registru jiného argument (pokud víc tak na zásobník), přepni se do režimu jádra – to je závislé na procesoru a operačním systému

Mechanismus – software interrupt, call gate – různé principy přepnutí se, nověji jsou dostupné I minimalistické sysenter()/sysreturn() funkce se snahou udělat to levněji než přes interrupt

Nastavení errno při chybě

Přerušitelné(po poslání signálu služba skončí jakoby neproběhla – chová se “atomicky” 🡪 do errno je EINTR) /nepřerušitelné služby jádra (stav delayed, signály a přerušení jsou odloženy).

Druhá kapitola referenční příručky

Glue funkce v knihovně – syscall se volají jako funkce – ve standardní knihovně jsou obálky (GLUE funkce) které přeskládají parametry – prostě wrapper na volání služeb jádra

Makra **syscall\_1..5** (číslo = počet argumentů volání) s číslem služby – lze volat libovolnou

**Knihovní funkce**

Kód definován v adresním prostoru procesu, není privilegované, nejsou atomické – Možnost příchodu signálu během provádění, po přerušení může doběhnout

Lze předefinovat (napsat vlastní funkci)

Nemusí být reentrantní (nemusí fungovat: přijde signal který by danou funkci zavolal znova)

Třetí kapitola referenční příručky

**Vznik procesu**

kromě procesu idle 0 vznikají procesy pomocí fork (clone), vyrobí kopii až na pár detailů (pid, co vrátil fork) přesnou kopii potomek sdílí paměť, stejný zásobník, stejné otevřené soubory

**fork(2) Vytvoření procesu**

#include <sys/types.h>

#include <unistd.h>

pid\_t fork();

Vytvoří potomka procesu, oba jsou stejné – musí mu zkopírovat paměť aby nebyla kolize. Rodiči vrátí číslo potomka.

Potomkovi vrátí nulu. (switch fork() - lze se větvit, nula – potomek)

Potomek dědí téměř vše (priorita, zásobník, otevřené soubory..) od rodiče. Výjimky jsou:

PID číslo procesu

PPID číslo rodiče

Zámky na souborech.

Návratová hodnota fork(2).

Signál od časovače. ještě nic nedělal

Čekající signály.

Hodnoty spotřebovaného strojového času.

**vfork(2) Virtuální fork()**

#include <sys/types.h>

#include <vfork.h>

pid\_t vfork();

Vytvoří potomka bez kopírování adresového prostoru (škoda kopírovat pokud to chci hned zahodit a pustit jiný)

Rodič je pozastaven dokud potomek nevyvolá exec(2) nebo \_exit(2). nad stejnou pamětí - jen aby potomek mohl zavolat exec, a potomek si zavolá exec a získá vlastní paměť 🡪 aby si nepřepisovali data

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, vrací číslo potomka který skončil (pid\_t).

Pokud je status nenulový ukazatel, uloží do něj informace o změně stavu potomka (flags, a makra na jeho parsování)

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).

Některé unixy definují makro WCOREDUMP(status) pro zjištění zda byl core soubor

WIFSTOPPED(status) – proces byl pozastaven.

Důvod pozastavení zjistíme makrem WSTOPSIG(status).

Parametr options je nula nebo logický součet následujících:

WNOHANG – nezablokouje se čekáním.

WUNTRACED – i při pozastavení nebo ladění potomka.

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 – struktura rusage (popsána u getrusage(2))

switch (pid = fork()) {

case 0:

potomek();

break;

case -1:

perror(”fork() failed”);

exit(1);

default:

rodic(pid);

break;

}

potomek() {

…

exit(status);

}

rodic(pid) {

int status;

waitpid(pid, &status, 0);

...

}

**Spuštění jiného programu**

Parametrem je spustitelný soubor. Začne vykonávat nový program.

Nahradí text (a VM) procesu jiným textem vezme se proces, zahodí se jeho paměť a zkonstruuje se nová na základě jiného programu

Nevzniká nový proces! – na to je fork()

**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); 🡨 služba jádra, ostatní funkce

\*l – předávání argumentů proměnný počt - jako funkce

\*v – předávání v poli

\*p umí spustit nový program aniž by se zadávala cesta - hledá v PATH

\*e -> předávají ENV 🡪 můžu pustit program a dát mu jiné proměnné prostředí, pokud neuvedu tak se kopíruje to co má rodič, tyto proměnné zcela nahradí staré (ovlivní getenv, setenv..)

argumenty příkazové řádky se předávájí všechny včetně nultého argumentu - tedy často path = argv[0] – jakobych dvakrát zadával cestu k programu (ne nutně - může se pomocí toho rozhodovat co spustit např busybox je software co se chová jako shell, ls, ps a další utility nebo jeho programy - rozhoduje se pomocí toho co je nultý argument, pokud je tam ls chová se jako ls 🡪 jeden tlustý prográmček, šetří místo; dalším příkladeb bash pozná zda je přihlašovací shell pomocí mínusu před názvem shellu)

Uzavře deskriptory s příznakem FD\_CLOEXEC (POSIX.1 vyžaduje např. u adresářů). Nutnost atomické operace spusť nový program s uzavři soubor – program chce mít možnost spustit nový a mít při tom otevřený log (selže – zapsat hlášku, podaří se – zavřít log pro potomka)

**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, vrátí exit status. Lze používat i shellové příkazy. Pokud mi jde o bezpečnost, pak použít execve – omezit environment variables aby někdo nepodstrčil proměnné pro shell které by pokazily vykonávání \_string\_

**fork(2) bez stránkování**

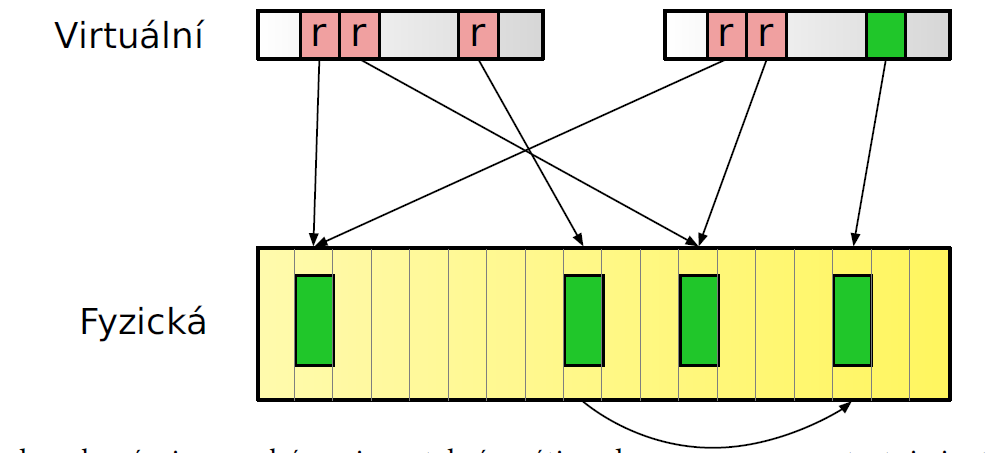
Systém bez stránkování – multitasking pomocí odswapování celého procesu a přiswapování jiného – zavolání fork rovnou odsunulo starý proces a nový zabral celou paměť – hotovo, mělo smysl kopírovat.

Následuje-li exec(2), nový adresní prostor se nahradí.

**fork(2) se stránkováním**

“Kopírování” pomocí forku - ponechání toho co tam je a označení za read only (kopie stránkové tabulky – ta je dokonce hierarchická stačí kopírovat kořen)

zde zůstalo R, rodič pořád nemůže psát, ale má tam počítadlo, takže najde tam 1 a jen změní r na zelenou, nedělá se zo hned protože ten druhý proces může běžet kdekoliv, a těžko to zjistit, ten proces by musel mít odkaz nahoru zpět ze stránky by musel být schopen zjistit komu všemu patří



Pokus o zápis do read only – přepnutí do jádra, zjistí se že by tam teoreticky psát mohl – pokud by se nesdílela, a udělá kopii která už je zapisovatelná (tomu druhému zůstane read only – to by se nedalo dohledat neumíme z fyzické dělat virtuální 🡪 až se pokusí číst tak jádro uvidí že to není sdílené a nastaví na writeable)

**Systémy se stránkováním -** Unifikovaný systém diskových bufferů a virtuální

paměti – sdílení stránek s diskovými buffery – stránka má svůj *obraz v souboru* (ty části programu co jsou read only – na disku je to co v paměti)

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). mohu si stránky mapovat do více procesů pokud mám knihovnu sdílenou tak že mám odkaz do paměti na její umístění - tedy "tato stránka patří tamtomu souboru" pozná že je tototéž a nekopíruje to 2x,

Stránkování na žádost (demand paging)

Text procesu se nenačítá do paměti, pouze se označí, odkud se má načíst. (pokud je v cache od jiného programu jehož nová kopie běží tak se namapuje z cache, jinak na disku)

Přístup k textu: výpadek stránky; stránka se načte ze souboru.

Při nedostatku paměti lze přímo zrušit z paměti (bez swapování, pokud není dirty – modifikovaná – což text programu nikdy není), později lze novu načíst. **Text file busy – nedovoluje zápis do souboru and kterým běží program (aby se mu nepodstrkoval jiný kód)**

Výhoda – nenačítá se celý text, který se možná ani nevyužije (např. chybné parametry na příkazové řádce, obrovský program se ani nespustí nic se nemusí kopírovat).

**Optimalizace** **I/O operace**

mmap(2) – namapuje do virtuálky soubor, nemusí se opravdu načítat do paměti, načtou se jen jednotlivé stránky v případě potřeby – až to teprvě někdo přistupuje (tedy open() nenačte soubor, tepve read() – pouze ten kus co je potřeba )

read(2) – v případě, že čteme do buffer zarovnaného s velikostí stránky, může systém

pouze namapovat (copy-on-write) stránku z buffer cache a náš buffer zahodí (většina OS to nedělá)

Co je rychlejší – hned kopírovat nebo dělat copy-on-write mapování? – záleží, pokud nebudu číst tak rozhodně mapování, ale pokud hodněkrát potřebuji zapisovat a musím hned měnit mapování při zápisu 🡪 nutím vylívat TLB (pokud vice procesorů dokonce musím zastavi vše a říct vylít teď) 🡪 levnější by to bylo na začátku prostě jednou zkopírovat

**Optimalizace Alokace paměti**

Služba sbrk(2) pouze posune konec dat, nealokuje nové stránky. (pouze poznačí adresy za platné)

Přístup k nově alokovanému prostoru – výpadek stránky, obsluha přidělí novou stránku.

Výhody – paměť se přiděluje až v okamžiku použití.

**Memory overcommitment**

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

Ano: nenastane situace, kdy OS nemůže dostát svým slibům a musí násilně ukončit proces. Ale je to náročné na výkon a hlavně může říct došla paměť I když je ramka prázdná – pouze sliboval ale nic se nepoužívá.

Ne: nedojde tak brzo k vyčerpání zdrojů. Některé systémy mají možnost nastavit míru

overcommitmentu.

Středí cesta : nastavit limit na přidání paměti, může např o 1GB překročit sliby u každého procesu

/proc/sys/vm/ovrcommit\_mem nastavit 0/1

Jak operační systém pozná, že došla paměť? Nepozná v momentu kdy “nenajde prostor na swapování” tak už dávno nejede - víc swapování spomaluje víc a víc, křivka využitelnosti - procesy se probouzí a swapují (každý se probudí, zjistí že nemá stránku - požadavek, uspí se probudí jiný a znovu), pomalý pc a dlouhá odezva -> kernel dělá heuristiky - co zabít / vůbec zabíjet? třeba procesy s memory leaky, může zabít nevinné, zabití největšího procesu – často Xserver, grafika 🡪 to uživatel chce

**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, řeší fragmentaci

BTW: když malloc vrátí NULL, tak to neznamená že došla paměť - problém by nastal až když tu paměť budu chtít použít A NAVÍC to už by program dávno nejel když dojde paměť 🡪 NULL se vrací když nejde zvětšit datovou oblast - např. limit na paměť ze systémových limitů

**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. kernel umí jen čísla typů souborů které podporuje (ne videa a obrázky..)

Vznik magických čísel - dříve: vzal se sboubor skočilo se na adresu 0 a začal se vykonávat, časem chceme sekce text, data, pro zachování kompatiblity se udělala hlavička a před ní instrukce skoku – narazil na skok pokračoval vykonávat, novější system uviděl skok aha skok kvůli hlavičce a přečetl si hlavičku a zjistil kde jsou data, text.. instrukce skoku byla první magické číslo

**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.

Na základě tohoto souboru se teda netvoří paměť ale nahraje se interpret a dá se mu tento script.

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 (formát, ne soubor po kompilaci), 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 společný formát pro bináry, textové, knihovny

Stejný formát pro \*.o soubory i pro spustitelné programy.

Sekce – mají textová jména jsou tam 4 základní (text..), lze přidávat další sekce a jejich atributy (Lze specifikovat, kam se která sekce má instalovat do paměti, je spustitelná, nemá se vůbec načítat…)

Možná rozšíření – několik sekcí pro kód, z jednoho sekvenčního assemblerového textu lze generovat několik sekvencí kódu – viz generování opravných kódu mimo main flow – viz kontrola userspace-pointerů jádrem. Ikona spustitelného souboru, a podobně.

**Přístupová práva procesu**

Pro UID a GID platí podobná pravidla.

Reálné U/GID vždy uživatele který sputil, efektivní U/GID – to co se reálně používá k ověřování práv, Saved(uložené) U/GID – uložení pro možnost přepínání (pokud je \_POSIX\_SAVED\_IDS).

Většina přístupových práv se prověřuje proti efektivnímu UID. Typy uid\_t a gid\_t, 16 nebo 32 bitů.

**getuid(2), getgid(2) Čtení UID/GID**

#include <sys/types.h>

#include <unistd.h>

uid\_t getuid();

uid\_t geteuid();

gid\_t getgid();

gid\_t getegid();

**setuid(2) Změna efektivního UID**

#include <sys/types.h>

#include <unistd.h>

int setuid(uid\_t uid);

int setgid(gid\_t gid);

euid == 0 (má práva root): nastaví reálné, efektivní i uložené UID na hodnotu uid.

Jinak je-li uid rovno reálnému nebo uloženému UID, změní pouze efektivní UID na uid.

Jinak končí s chybou EPERM

**setreuid(2) Prohodit reálné a efektivní 🡪 když neměl uložené u/gid**

#include <sys/types.h>

#include <unistd.h>

int setreuid(uid\_t ruid, uid\_t euid);

int setregid(gid\_t rgid, gid\_t egid);

umožní měnit reálné tímto prohozením

**Uložené ID**

Pokud je definováno \_POSIX\_SAVED\_IDS

FIPS 151–1 vyžaduje tuto vlastnost.

Změna reálného UID: pouze superuživatel.

Efektivní (a uložené) UID je nastaveno službou exec(2) na UID vlastníka souboru, pokud má příslušný program nastavený set-uid bit. Jinak efektivní = reálné.

Uložené UID: při exec(2) se kopíruje z efektivního UID

Start programu 8086 1337 1337

setuid(8086) 8086 8086 1337 *nebo:* setuid(8086) 8086 8086 1337

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

exec() 8086 1337 1337

* Volní exec nic nemění, může obě měnit volání exec s reálné = efektivní způsobí pouze 8086 🡪 při startu uložené se kopíruje z efektivního

**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** (setuid() rootu mění všechny)

Vyžaduje systém podporující uložená UID (reálné nelze změnit, jen s právy root)

programy jsou ovlivnitelné proměnými prostředí, program se set-uid/gid bitem musí být malý(=málo chyb, co nejméně knihoven) program co nepoužívá proměnné prostředí které dostane od exec (typicky volání system() je pak nebezpečné)

**Doplňková GID (seznam)**

Starší verze UNIXu – při přihlášení uživatele: UID a GID podle souboru /etc/passwd, změna GID pomocí newgrp(1) 🡪 pokud chtěl použít, musel pouštět nové shelly s jinými právy, muselo se to pořád přepínat, otravné

Novější systémy – doplňková (supplementary) GID. (Zavedeno v 4.2 BSD)

Inicializace – při přihlášení podle /etc/group.

Přístupová práva – kontroluje efektivní U/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=0, 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. Používá se při přihlašování.

Navíc do seznamu skupin přidá skupinu group (primární v /etc/passwd)

Knihovní funkce (katalog 3) –> volá setgroups(2).

**Čísla 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. Getppid se používalo na měření trvání přepnutí se do jádra (pid se může cachovat, pro ppid musí do jádra, mohlo se měnit – kdyby umřel, dostanu ppid=1)

**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.

uživatelský + systémový <= reálný \* počet procesorů (může běže paralelně)

**times(2) Získání časových informací**

#include <sys/times.h>

clock\_t times(struct tms \*buf);

struct tms {

time\_t tms\_utime; 🡨 uživatelský čas

time\_t tms\_stime; 🡨 systémový čas

time\_t tms\_cutime; 🡨 + potomci

time\_t tms\_cstime; 🡨 + potomci

}

Vrací reálný čas od nějakého okamžiku v minulosti. (opakované volání = interval)

Poslední dva údaje jsou včetně potomků.

Vrací počet tiků systémového časovače. (na sekundy \* frekvence časovače – pomocí sysconf)

kernel se rozhoduje zda použít systémový časovač, pokud mám nečinný PC a všechny procesy čekají na událost, tak se může vypnout přerušení časovače - zbytečně by se budil a jišťoval že nikdo nechce jet (stejně bude probuzen tou událostí) …(podobně když běží pouze jeden aktivní proces)

**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.

RUSAGE

**struct timeval ru\_utime; /\* user time used \*/ náhrada times(2) – v sekundách, ne tiky**

**struct timeval ru\_stime; /\* system time used \*/**

long ru\_maxrss; /\* maximum resident set size kolik stránek paměti měl max stránek v rámcích - zaráz

long ru\_ixrss; /\* integral shared memory size \*/

long ru\_idrss; /\* integral unshared data size \*/

long ru\_isrss; /\* integral unshared stk size \*/

**long ru\_minflt; /\* page reclaims page faults - vyřeší bez uspání a načtení stránky z IO \*/**

**long ru\_majflt; /\* page faults – musí uspat (popřípadě hledat oběť) a načítat z disku \*/**

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 – zavolal jádro a to ho uspalo \*/**

**long ru\_nivcsw; /\* involuntary ctxt switches – přerušen časovačem / událostí \*/**

LIMITY

**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 \*/ 🡪 měnitelné až do hard

rlim\_t rlim\_max; /\* Hard limit \*/ 🡪 hard strop, mění akorát root

};

Běžný uživatel – změny soft limitu až do výše hard limitu. **Limity se dědí na potomky.**

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 malloc/setbrk selže kvůli tomuto

RLIMIT\_STACK : velikost zásobníku

RLIMIT\_RSS : resident set size většinou neimplementováno – použití nic neudělá (náročné na zjištění)

RLIMIT\_NPROC : počet procesů daného uživatele (limit pouze jednoho procesu -ne globální- jiný proces stejného uživatele to může mít jinak)

RLIMIT\_NOFILE : max. počet otevřených souborů

RLIMIT\_MEMLOCK : uzamčená paměť (pro citlivý data - klíče neswapovat)

RLIMIT\_AS : velikost virtuální paměti – limit na všechny adresy – na veškerou paměť, tento se používá nejvíc

**Priorita procesu**

**nice(2) Změna priority procesu**

#include <unistd.h>

int nice(int inc);

Přičte (nenastavuje) inc k prioritě volajícího procesu. Pouze superuživatel může uvést negativní inkrement. Menší číslo – vyšší priorita. Méně prioritní dostane taky nějaký čas, jen méně než vice prioritní (aby měli šanci pustit zámek na který může čekat vyšší priorita).

omezený na malý rozsha čísel -20 až 19, běžné procesy na 0

**sched\_yield(2) Kooperativní multitasking**

#include <sched.h>

int sched\_yield();

Předá řízení jinému procesu, pokud je takový proces k dispozici. Nepoužívat! (používal se místo uspání procesu jako řečení že nechce teď běžet, ale reálně pokud jiný proces nemůže běžet tak akorát pálí čas voláním tohohle 🡪 aktivně čeká)

Dispečerská priorita (interní hodnota, neměnitelné zvenku) je ta která se počítá na základě priority a chování procesu (ten který bere vstup myši a hned se uspí asi bude mít méně než ten co aktivně počítá 🡪 mění se s chováním procesu) a teprve podle ní se rozhoduje, má fronty pro procesy s různým číslem (všechny se setjným v jedné frontě)

**Priorita libovolného procesu**

**get/setpriority(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 jednoho procesu.

PRIO\_PGRP – priorita celé skupiny procesů.

PRIO\_USER – priorita všech 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**.

proces se může prohlásit za nového vedoucího, jeho podskupina se utrhne a on se stane novým vedoucím - např úloha ls | grep | cat... -> skupina procesů

**setpgid(2), setpgrp(2) nastav procesu pid číslo skupiny pgid**

#include <unistd.h>

int setpgid(pid\_t pid, pid\_t pgid);

pid\_t setpgrp(void);

pid nebo pgid 0 (pid=0 moje pid, pgid=0 skupina vedená mnou)

setpgrp() je totéž co setpgid(0, 0).

**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); 🡨 získá id session procesu pid

pid\_t setsid(void); 🡨 nastaví sebe jako vedoucího session

Selže, je-li proces vedoucím procesem skupiny.

**Démoni**

Démon – proces běžící na pozadí bez řídícího terminálu. když skončím tak všechny procesy dostanou SIGAB, démon ne

Služba co tvoří démona (byla vytvořena pomocí fork ze shellu, dělá se to takto aby exit služby vrátilo prompt shellu co nejdříve) dělá:

* fork(2)
* Rodič: \_exit(2)
* setsid(2)
* Pracovní adresář – změnit na /.
* Otevřené soubory – uzavřít.
* Std. deskriptory 0, 1 a 2 – otevřít na /dev/null.

**daemon(3)**

#include <unistd.h>

int daemon(int nochdir, int noclose);

zda má nastavovat pracovní adresář na / a zda má uzavírat otevřené soubory

**I/O operace**

Všechno je soubor.

Soubor – základní jednotka při zpracování I/O operací z pohledu služeb jádra.

File 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);

v unixu může mít víc souborů otevřený soubor pro zápis, je na nich aby si to spravovali sami

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

O\_CREAT – vytvoření souboru, pokud neexistuje.

O\_EXCL – chyba, pokud soubor existuje.

O\_TRUNC – zarovnání souboru na nulovou délku. ls > /tmp/soubor zarovná na 0 a začne přepisovat

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, čtení může vrátit chybu ale později tam data mohou přibýt (roura, síťové spojení) jinak se zablokuje pokud není co číst a je to roura apod.

O\_SYNC – synchronní výstup (když zavoláme zápis, tak se hned dokončí a nezůtane v cache)

**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.

**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 (může načíst míň a není to chyba)

pokud požádám o neblokující operace, a žádná data nejsou nečeká a vrátí -1 🡪 ale může příště mít

**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.

může zapsat míň, pokud nezapíše nic chyba vrátí -1

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

#include <unistd.h>

off\_t lseek(int fd, off\_t offset, int odkud); offset od hodnoty 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 (roura) nelze použít lseek(2).

Rozdíl mezi otevření souboru s O\_APPEND nebo bez a seek na konec – append je atomický, seek ne – může např předtím než zapíšeme někdo něco změnit, můžeme přepsat.

Pokud se lseek kompiluje na 64bit systému tak se lseek nastaví s 64bit offsetem, jinak 32bit (max 4GB soubor)

Při otevření s O\_APPEND lze se pohybovat kdekoliv ale zápis nás posune vždy na konec

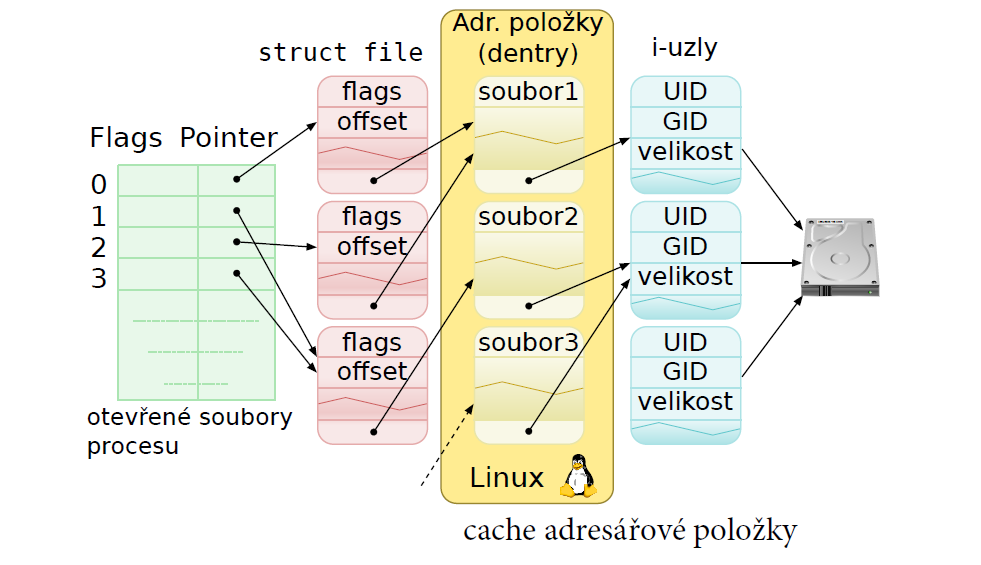
**Tabulka otevřených souborů**

Každý soubor má odkaz číslo deskriptoru na struct file (flags otevření, offset pozice), a tyto ukazují na inode (práva, velikost a mimojiné vlastní adresy záznamů), lze zkopírovat fd (a tedy různá čísla fd mohou ukazovat na stejnou struct file), lze otevřít soubor vícekrát (různé file struktury na stejný inode)

V paměti jsou iuzly (BSD vnode) jen těch nedávno používaných souborů

Nelze zjistit lehce jméno souboru (to je ve složce) 🡪 proto má linux dentry

dentry primární vzhled struktury souborového systému pro jádro (ne vše ale dost toho co se používá), pokud během práce se souborem přejmenuji složku tak to pozná



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

#include <unistd.h>

int dup(int oldfd);

int dup2(int oldfd, int newfd); 🡨 dame mu descriptor který má otevřít (uzavře pokud je

používaný)

Duplikování – nový odkaz na **tutéž** strukturu file.

Použití: přesměrování v shellu (programům může nastavit jako stdout fd otevřený soubor: echo > text.txt)

Zápis dvakrát do stejné struktury file (tedy fd ukazují na stejnou) způsobí dva zápisy, ale když mám dva fd co ukazují na různé file (2x open bez O\_APPEND), tak druhý zápis přepíše ten první (struct file má pozici)

**fcntl(2) Změna vlastností deskriptoru**

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

int fcntl(int fd, int cmd); cmd – číslo co se má udělat 🡪 ARG1 | ARG2

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

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. (protože kontrola práv se provádí při otevření)

F\_GETLK, F\_SETLK – zamykání souboru, synchronizace(viz dále).

**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, cmd záleží na daném zařízení

Některé operace: nelze převést na čtení/zápis dat (např. SMART informace disku).

je lehké vše považovat za file a mít to jako proud byte - tiskárny, klávesy klávesnice, ale někdy chci mít řídící operace (terminál měnit vstup zasílání klávesa po klávese, nebo po blocích, disk - načíst SMART informace nebo načíst novou DISK partition table pro kernel která se změnila)

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);

**i-uzel**

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

metadata souboru. Čtení atributů – služba stat(2), příkaz ls(1).

Délka souboru (poslední platný bajt, může mít díry atd..)

Typ souboru (roura, terminál, soubor..)

UID a GID vlastníka

Časy – přístupu, modifikace, a změny stavu.

Přístupová práva rwxr-x-r--

Počet odkazů – klesne-li na nulu, je i-uzel uvolněn a jeho datové bloky také.

Odkazy na datové bloky

**Odkazy na datové bloky**

Tradiční (starší) 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.

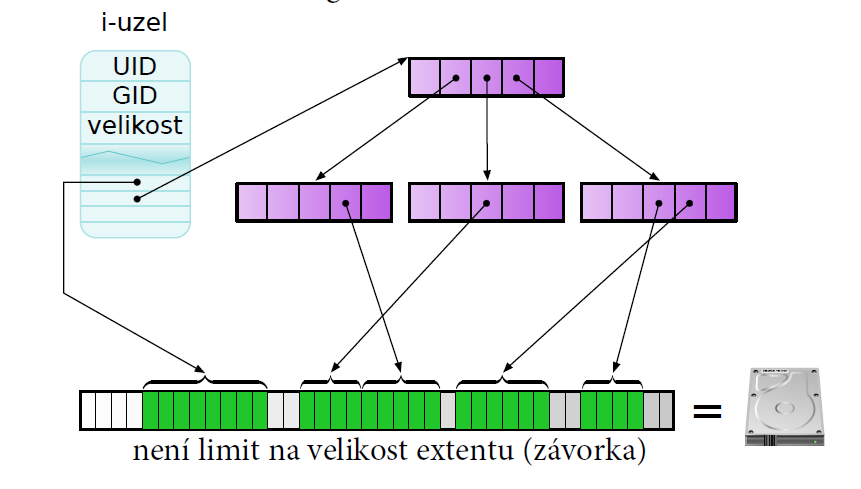
Přímý přístup ke kterémukoli místu souboru.

Díry v souborech – /var/log/lastlog (číslo uživatele = adresa v souboru, hodně velkých děr 🡪 ale je to skoro zadarmo, zbytek není prostě naalokován), core.

**Extent-based souborové systémy**

novější systém, problematičtější když spadne při reorganizaci stormu, extent = adresa + délka

efektivnější pro celé bloky dat (kopírovaní 600MB obraz cd) – lze jedním extentem popsat celá data pokud se zadaří a najdu souvislou oblast pro záznam



**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); 🡨 neprochází symlinky

int fstat(int fd, struct stat \*st);

STRUKTURA stat

st\_dev – zařízení, na kterém se i-uzel nachází

st\_ino – číslo i-uzlu (to v izulu není)

st\_mode – typ souboru a přístupová práva (parsuje se makry)

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.

snaha moc neměnit atime -> pokus o ČTENÍ může selhat out of memory (s systém může dělat kopie změny -> read generuje write, snaha nedělat -> optimalizace neaktualizuje atime pokud je menší než mtime – šetří

Typ souboru lze z položky st\_mode získat těmito (bool) 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 lze z st\_mode získat těmito maskami:

S\_ISUID, S\_ISGID, S\_ISVTX – set-uid bit, set-gid bit a sticky bit (mazat v adresáři se sticky bitem smí vlastník souboru nebo adresáře)

S\_IRUSR, S\_IWUSR, S\_IXUSR – práva vlastníka souboru.

S\_IRGRP, S\_IWGRP, S\_IXGRP – práva skupiny.

S\_IROTH, S\_IWOTH, S\_IXOTH – práva ostatního světa.

**access(2) Ověření přístupových práv**

#include <unistd.h>

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

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 problem - nepoužívat!

Symlink ukazující do prázdna má reálné U/GID všechna práva, stejná práva jako neexistující soubor

Symlink ukazující na soubor má pouze X\_OK

**Nově vytvářené soubory – podle čeho nastavit práva?**

Vlastník – podle efektivního UID vytvářejícího procesu.

Skupina – více možností.

Podle efektivního GID procesu, který soubo vytvořil

Podle GID adresáře, ve kterém je soubor vytvářen (u systémů s doplňkovým gid 🡪 chci spíše práva té složky, aby tam mohli všichni kdo smí ke složce – pracuji tam pod skupinou, mám efektivní GID jiné než doplňková vůče kterým sem se autentizoval)

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

Dnes povolují unixy obě dvě varianty, když adresáři nastavíme setGID bit adresáři, tak nové soubory dědí GID adresáře

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

#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.

Typicky pak používám 3 typy práv u nových souborů - temp - 0600, executable - 0777, newfile = 0666 a nechám na umask aby vymazal z toho to co nechci nastaveno

**Změna vlastníka souboru**

dříve mohli měnit všichni, pak se vymyslelo omezování místa pro uživatele, kteří by si mohli registrovat soubory pro jiné uživatele aby jim to nezvyšovalo kvotu – zakázáno

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 (spouští jej vlastník nebo má set-uid bit)

Nemění se zároveň s GID také UID vlastníka souboru.

Nové GID je totožné s efektivním GID procesu nebo s některým z dodatkových GID procesu. (tedy mohu dát do skupiny ve které jsem členem)

* Při změně vlastníka/skupiny se vždy nulují set-UID/set-GID bity.
* BSD (a I jiné) nulují u souboru tyto bity I při zápisu do něj (změním soubor = musí se opět rozhodnout zda chci povolit aby šel I nadále spouštět)

**chown(2) Změna vlastníka/skupiny souboru**

#include <sys/types.h>

#include <unistd.h>

int chown(char \*path,uid\_t owner,gid\_t grp); 🡨 změna symlinku (kromě SVR4, lchown())

int lchown(char \*path,uid\_t owner,gid\_t grp); 🡨 změna symlinku ne souboru kam ukazuje

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! (tedy není lchown() 🡪 chown mění symlink)

**neexistuje chgrp narozdíl od shellu, vše je v této funkci**

**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);

práva: setuid: 4, setgid: 2, sticky: 1 (první bit), read: 4, write: 2, exec: 1 (druhý, třetí, čtvrtý – ugo)

7777 🡪 rwsrwsrwt

7666 🡪 rwSrwSrwT velké písmeno 🡪 pod ním není právo (-)

0765 🡪 rwxrw-r-x

3210 🡪 -w---s—T

**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.

Pokud truncate změní celikost souboru, pak změní st\_ctime (poslední doba změny statusu), st\_mtime (doba modifikace)

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

#include <unistd.h>

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

Vytvoří další jméno i-uzlu. V shellu jako **ln existujici\_soubor nový\_link**

Může skončit s chybou, pokud path a newpath nejsou na tomtéž svazku. (na to je symlink)

**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 (tedy unlink na jedinný existující pevný odkaz nemusí nutně smazat soubor, pokud je otevřen programem – smaže se až ho program pustí)

* Použití: ANONYMNÍ DOČASNÝ SOUBOR: otevřít soubor s flagem O\_CREAT|O\_EXCL a pak ho zrušit pomocí unlink (bacha, není to atomické, mezi otevřením souboru a jeho unlinkem s tím mohl někdo něco udělat)

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

#include <stdio.h>

int remove(char \*path);

Smaže soubor nebo adresář. Je součástí normy ANSI C (což je takovej artefakt, ANSI nepracuje s pojmem adresář, ale funguje to taky)

**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.

**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;

}

Nastavení *atime* a *mtime*.

Je-li parametr times NULL, nastaví na aktuální čas.

Nastavovat smí pouze vlastník souboru (nebo superuživatel).

**utimes(2) Nastavení časů souboru**

#include <sys/time.h>

int utimes(char \*path, struct timeval times[2]);

**Symbolické linky**

**Příklad: Symbolický link**

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

Symbolický odkaz na soubor pomocí cesty.

Relativní (vůči místu uložní symlinku) versus absolutní symbolické linky (podpora obou týpů cest)

Uloženo v datovém bloku souboru.

Přístupová práva – obvykle nemají význam (přečte adresu kam odkazuje a opět musí kontrolovat práva)

**symlink(2) Vytvoření symbolického linku**

#include <unistd.h>

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

Vytvoří symbolický link path, obsahující řetězec sympath. Ve shellu **ln -s stary novy**

**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). –volání open by neotevřelo symlink ale soubor kam ukazuje. Vrací délku symlinku.

Obsah bufferu **není ukončen nulovým znakem**.

**Služby, které neprocházejí symlinky: chown(2) (pokud v systému neexistuje lchown(2)),**

**lchown(2), lstat(2), readlink(2), rename(2) a unlink(2).**

Tvrdý link na symlink udělá odkaz na symlink, ne na soubor kam vede

ln -s myfile symlink && ln symlink hardlink 🡪 hardlink zvýší číslo odkazů pro symlink

**Vytváření dočasných souborů**

Adresáře /tmp, /var/tmp. Mají sticky bit

Exkluzivita

Bezpečnostní problém se symbolickými linky. (vytvoření souboru se shodným jménem již existujícího symlinku bez O\_EXCL mi může podstrčit LIBOVOLNÝ SOUBOR)

Linux – O\_CREAT|O\_EXCL. 🡪 atomické, selže pokud už tam symlink je

FreeBSD – O\_NOFOLLOW.

**mkstemp(3) Vytvoření dočasného souboru v C**

#include <stdlib.h>

int mkstemp(char \*template);

**Příklad: Použití mkstemp(3)**

char \*tmpfile = strdup(”/tmp/mail.XXXXXX”);

int fd = mkstemp(tmpfile);

Vytvoří dočasný soubor podle dané masky = písmena X – na konci, aspoň 6.

Vrátí deskriptor, do parametru zapíše jeho skutečné jméno (není const)

**Nepoužívat: mktemp(3), tmpnam(3), tempnam(3).**

**Vytvoření dočasného souboru ve SHELLU**

**Špatně:**

TMPFILE=/tmp/mujprogram.$$ 🡨 nepoužívat číslo procesu, dá se uhodnout

ls > $TMPFILE

**Správně: bacha, narozdíl od Cčka ve shellu je to dobrá implementace**

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 – nelze hardlink u adresáře (nejasný význam „..“ v adresáři)

Délka jména – záleží na FS. Původní UNIX: 14, dnes většinou aspoň 252.

Délka struktury – pevná nebo proměnná.

Organizace adresáře – seznam, pole, strom.

Každý adresář má aspoň dva odkazy – sám na sebe a jeho rodič na něj 🡪 nelze smazat adresář pomocí **unlink**

Sémantika konstrukce „//“ 🡪 vždy interpretováno jako jedno lomítko (snaha o adresave vzdálených pc… nepoužívá se)

**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 ořezaná pomocí umask(2).

**rmdir(2) Smazání adresáře**

#include <unistd.h>

int rmdir(char \*path);

Smaže prázdný adresář.

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

**opendir(3) Otevření adresáře**

#include <sys/types.h>

#include <dirent.h>

DIR \*opendir(char \*path);

int closedir(DIR \*dp);

V některých systémech je možné adresář číst přímo pomocí služby read(2).

Linux: O\_DIRECTORY.

POSIX.1 definuje přístup k adresáři pomocí opendir(3)

**readdir(3) Čtení adresářové položky**

#include <sys/types.h>

#include <dirent.h>

struct dirent \*readdir(DIR \*dp);

void rewinddir(DIR \*dp); 🡨 vrácení ukazatele čtení na začátek (při modifikaci složky)

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).

Veškerá funkcionalita programu ls(1) je programovaná 🡪 readdir neřadí soubory, neskrývá soubory s tečkou na začátku atd..

**Adresáře procesu**

**getcwd(3) Jméno pracovního adresáře**

#include <unistd.h>

char \*getcwd(char \*buf, size\_t sz);

Vrátí cestu k pracovnímu adresáři. Údaj vyplňuje OD KONCE bufferu, vrací pointer na první platný znak (prochází to směrem ke kořeni, plní reverzně)

Je-li sz příliš malé, skončí s chybou. V linuxu implementace knihovní funkce nad getcwd(2).

Pozor na rozdíl mezi pwd (shellovský buldin co si pamatuje odkud se uživatel dostal do cwd, což nemusí být nutně parent directory – symlinky apod) a /bin/pwd (reálně počítá pwd z cwd/..)

**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 při přechodu.

Místo getcwd(3) a po čase chdir(2) zpět lepší použít open() a po čase fchdir(2). (zatímco cesta se mohla změnit, fd bude vždy platný)

Proč neexistuje cd(1)? 🡪 spustil by se jako nový proces, změnil by sobě cwd a skončil – změna nepropagována do rodiče 🡪 shell todo musí dělat sám

**Kořenový adresář procesu**

**chroot(2) Změna kořenového adresáře procesu**

#include <unistd,h>

int chroot(char \*path);

Změní kořenový adresář procesu. Povoleno pouze superuživateli – jedinný způsob jak z něj utéct výše pokud není “/” (jinak by šlo udělat hardlink na program se setuid bitem (pokud vlastníkem root 🡪 běží pod rootem), v této složce si vytvořit usr/lib a změnit mu root dir na moji složku 🡪 podstrčím mu svoji implementaci knihoven, takže vlastně získám root práva na vykonání libovolné funkce)

Pracovní adresář pouze jeden 🡪 vlákna nebo funkce v knihovně nemohou mít svůj vlastní, musí modifikovat ten sdílený 🡪 kvůli tomu se přidávají služby \*at()

**\*at(2) relativní cesty ve služnách**

#include <unistd.h>

int openat(int dirfd, path, flags, mode);

int mkdirat(int dirfd, path, mode);

int unlinkat(int dirfd, path, flags);

… renameat() .. a další.

**Adresářový deskriptor** pro otevření souboru relativně vůči němu.

Konstakta AT\_CWD pro pracovní adresář procesu 🡪 openat(AT\_CWD, path ..) == open(path ..)

**sync(2) Synchronizování disků**

#include <unistd.h>

void sync(void);

Unix nikdy negarantoval že po write() ta data opravdu budou zapsána (disk může selhat, write() to dává do cache)

Zařadí buffery které se mají ukládat na disk do fronty pro okamžitý zápis.

Nečeká na dokončení zápisu – pouze znamení pro start zápisu. Nelze dělat transakce (db)

Staré unixy jádro ani toto neřešily, reálný zápis se děl

Pokud došla paměť, pak tead zavolal sync() aby mohl do cache dávat další data

Jinak: Byl tam malý deamon co každých 30 sec volal sync() 🡪 když spadl pc, čekalo se 30 sec aby pokud běží měl šanci to zapsat.

**fsync(2), fdatasync(2)**

#include <unistd.h>

int fdatasync(int fd); 🡨 zápis dat

int fsync(int fd); 🡨 zápis dat i metadat (iuzel)

Zapíše všechny modifikované části souboru na disk. Počká na dokončení – máme garanci, že je to na disku. Lze implementovat transakce db – udržuje žurnál změn, musí vědět že se povedl zápis.

fdatasync(2) nezapisuje metadata souboru (čas modifikace, ...).

Norma neříká co jsou metadata souboru

Problém – patří nadřazený adresář pod „metadata souboru“? – v linuxu nutnost volat fsync i na adresář pokud chci zajistit uložení změny adresáře (fsync na souboru to neuloží, adresář není součástí metadat)

Problém – jak atomicky přepsat soubor? (např config soubory uloží nový nastavení vedle a pak udělá atomicky rename – když spadne, tak nepřijdu o starý 🡪 ALE co když se udělá rename předtím než se skutečně uloží ta data na disk? – dostanu prázdný soubor, a nechci používat fsync protože to brzdí uživatele…) 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); 🡨 dle typu

int mkfifo(char \*path, mode\_t mode); 🡨 pouze rouru

Vytvoří soubor daného jména. I verze \*at()

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 nez spolupráce superuživatele. 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.

s jednoduchými právy lze jen uživatele vlastníka nebo skupinu práv, ale běžný uživatel skupinu neudělá, pokud si teda uživatel chce definovat lidi co smí k souboru tak to moc nejde

u::rwx,g::r-x,o::r-- 🡺 rwxr-xr—

u::rwx,g::r-x,o:---, u:bob:rwx,g:wheel:rw-,m:r-x 🡺 uživatel bob dostane pravo rwx, skupina wheel dostane rw-, maska ořeže na r-x (poslední povinný údaj m je maska, bob dostane pouze r-x)

Vyhodnocování – hledá se shoda efektivního UID procesu, pokud se nenalezne, tak efektivní GID a doplňková GID, pokud se ani tady nenalezne, použije se položka o:. U nepovinných položek logický součin s maskou. **Použije se první nalezené právo co lze použít, dál se nejde** (tedy takto lze někoho vylouči ze skupiny, dát ho před ni s menšími právy)

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:: (ne g!); není-li, pak g::.**

Implicitní ACL (druhý ACL kromě vlastního ACL adresáře – má 2) – u adresářů. Použije se pro nově vytvářené soubory.

Programy – getfacl(1), setfacl(1), chacl(1). Též acl(5).

**Komunikace mezi procesy**

**Roura**

Datový kanál – zasílání proudu dat mezi procesy. FIFO struktura, dá se udělat pomocí kruhového bufferu. Velikost lze zjistit pomocí [f]pathconf()

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. Jejich uzavřením zanikne.

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 (nemusí mít zprostředkovatele – rodiče – co by to vytvořil)

**Vlastnosti roury**

Zápis až do velikosti PIPE\_BUF je atomický. (problem, pokud víc, pak nemám garantováno že příště zapíšu já, pokud píše víc procesů zároveň)

Otevření (pojmenované) roury pro zápis se zablokuje do doby, než některý jiný proces otevře rouru pro čtení. (u nepojmenované se to neděje tam už je otevřeno)

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 🡪 pozná že už nikdo nezapisuje.

Zápis do roury způsobí zaslání SIGPIPE, nemá-li žádný proces rouru otevřenou pro čtení 🡪 pozná že už nikdo nečte.

#include <unistd.h>

...

int r, fd[2];

int buf[PIPE\_BUF];

...

if (pipe(fd) == -1) {

perror(”pipe()”);

exit(1);

}

switch (fork()) {

case -1:

perror(”fork()”);

exit(1);

case 0: /\* Potomek \*/

close(fd[0]);

write(fd[1], ”Manipulační svěrka\n”, 19);

exit(0);

default: /\* Rodič \*/

close(fd[1]);

while ((r = read(fd[0], buf, PIPE\_BUF)) > 0)

write(1, buf, r);

wait(NULL);

exit(0);

}

pokud potomek neudělá close tak nedostane SIGPIPE pokud rodič uzavře deskriptor, write by se zabokoval po čase 🡪 došel by mu buffer a jediný kdo ho může vyprázdnit=číst je on sám

kdyby nebyl čevený close tak se rodič zablokuje, 0 vrací až když je vše uzavřeno pro zápis – čekal by sám na sebe až zapíše 🡪 blokoval by sám sebe

**Signály**

Signál – asynchronní událost. zaslán jiným procesem, ten kdo ho dostane ho obdrží \*\*někdy\*\*

Reakce – **ignorovat**, **zachytit ovladačem** (*handler*) máme dopředu nadefinováno co se má dělat po obdržení signálu, **implicitní** **akce** - pokud nedefinujeme je tam nějaké default chování

Zachycení signálu – proces začne vykonávat handler – přestane tok instrukcí, po obsluze se pokračuje v místě přerušení.

Ukončení handleru – pokračování od místa přerušení.

Zaslání signálu procesem – naše **efektivní UID musí dopovídat reálnému UID procesu, kterému zasíláme signál**

Zaslání signálu jádrem – obvykle synchronní odpověď na akci procesu. Proces použije nezarovnaný přístup na sběrnici – v jádře výjimka, obsluha => doručit procesu signál SIGBUS 🡪 z pohledu procesu je to synchronní – požádal o IO a dostal signal

**Reakce na signál**

**signal(2) Nastavení reakce na signál**

#include <signal.h>

void (\*signal(int sig, void (\*hndlr)(int)))(int); 🡨 funkce co na signal sig zavolá hndlr a vrátí ho

nebo jinak (to samé s typedef – musím si udělat vlastní typ):

typedef void MujHandler(int); 🡨 při zavolání signálu dostane jeho číslo - argument

MujHandler \*signal(int sig, MujHandler \*hndlr);

Nainstaluje ovladač signálu.

Vrátí jeho předešlou hodnotu.

Speciální hodnoty handleru: SIG\_IGN (ignore), SIG\_DFL (default, implicitní reakce)

Parametrem ovladače (hndlr) je číslo signálu – lze mít jednu funkci na obsluhu všech a podle čísla signálu měnit chování

execve(2) ruší všechny handlery (protože se odstranily z paměti, už nejsou platné)

**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); 🡨 sám sobě

pid > 0 zaslán procesu s číslem pid.

pid == 0 zaslán všem procesům ze stejné skupiny ke které patříme

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

pid == -1 nespecifikovaný výsledek (obvykle všem procesům 🡺 v kontextu uživatele znamená všem svým procesům, k jiným nemám právo).

signo == 0 – jen testuje práva k zaslání signálu (test na existence procesu 🡪 errno dostanu EPERM = nemám právo=existuje , ESRCH == neexistuje).

**Čekání na signál**

**pause(2) Čekání na signál**

#include <unistd.h>

int pause();

Zablokuje proces dokud nedostane signál. Není dostatečné na synchronizaci. Pokud pokud proces zavolá pause() ale signal na který chce čekat dojde předtím, tak se uspí navždy.

**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. Default akce**

SIGABRT Abnormální ukončení APSB core (= ukončení)

SIGALRM Časovač PSB ukončení

Lze říct pošli mi alarm za x sekund a zavolat pause()

SIGBUS Hardwarová chyba SB core

Nezarovnaný přístup na sběrnici

SIGCHLD Změna stavu potomka JSB ignorování

Asynchronní notifikace o skončení potomka, pokud nemá čas na něj čekat – může si definovat co udělat při obdržení, může to být I pozastavení

SIGCONT Pokračování po STOP JSB znovuspuštění

Pokračování po SIGSTOP, nelze ignorovat

SIGFPE Chyba reálné aritmetiky APSB core

Pokud dojde k chybě např záporný logaritmus, dělení nulou

SIGHUP Zavěšení linky PSB ukončení

Původně ukončení sériové linky – poslán procesům které běžely a jejichž terminal je

uzavřen, nebo příkaz nohup (nastaví ignorování SIGHUP)

Deamon reaguje na tento signal znovunačtením konfigurace (recyklace významu)

SIGILL Neplatná instrukce APSB core

Procesor má vykonat číslo které není platnou instrukcí

SIGINT Přerušení z terminálu APSB ukončení

Je interrupt, po zmáčknutí ^C

SIGKILL Ukončení procesu PSB ukončení

Číslo 9, nelze ignorovat – nepodmíněné ukončení

SIGPIPE Rouru nikdo nečte PSB ukončení

SIGQUIT Znak Quit na terminálu PSB core

Jako SIGINT ale vygeneruje core

SIGSEGV Chyba segmentace APSB core

Použití neplatné adresy

SIGSTOP Pozastavení procesu JSB pozastavení

Proces je pozastaven, nelze zachytit ani ignorovat

SIGTERM Výzva k ukončení APSB ukončení

Takové “hezké” ukonči se – dá mu možnost něco si uložit – např restart pc – dostane sigterm a po limitu pošle SIGKILL

SIGTSTP Znak Stop na terminálu JSB pozastavení

Jako SIGSTOP ale lze ignorovat

SIGTTIN Pokus o čtení z terminálu JSB pozastavení

Dostane proces který není na popředí při pokusu o čtení

SIGTTOU Pokus o zápis na terminál JSB pozastavení

Dostane proces který není na popředí při pokusu o zápis

SIGUSR1 Uživatelský signál 1 PSB ukončení

Proces si je může použít k čemu chce

SIGUSR2 Uživatelský signál 2 PSB ukončení

**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 signal (bitová maska 0/1 takže dvojí zaslání způsobí jedno vyhodnocení)

**Spolehlivé signály**

Vygenerování signálu – v okamžiku volání kill(2).

Doručení signálu (*delivery*) – vykonání reakce na signál. Mohou být doručeny I když zrovna volal jádro - před návratem z režimu jádra do userspace kontrola zda nedošel signál a případné doručení.

Čekající signál (*pending*) – stav mezi vygenerováním a doručením. Může čekat proto že příjemce ještě neběžel / zablokoval signál.

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); prázdná množina

int sigfillset(sigset\_t \*set); všechny prvky

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. – přidat ty co jsou v set

SIG\_UNBLOCK – průnik původní množiny a doplňku set. – odebrat ty co jsou v set

SIG\_SETMASK – nastavení na set. – přesně nastavit na set

Jsou-li odblokovány čekající signály, je aspoň jeden doručen před návratem ze sigprocmask(2). Do Old mi uloží verzi před změnou.

**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. Lze synchronně zpracovávat signály – přijímat kdy chci.

**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 – vše atomicky. Lze realizovat synchronizace na rozdíl od pause() – může si zablokovat signál na který pak bude čekat, pustí potomka, a pak naráz povolí signál a zablokuje se.

**sigaction(2) Změna reakce na signál (novější než signal())**

#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 (ignore) nebo SIG\_DFL (default).

sa\_mask – signály, které mají být zablokovány během provádění handleru.

Do Old ukládá stav před změnou.

act = NULL 🡪 zjištění stavu bezezměny

**Příznaky struktury sigaction**

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

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

SA\_ONSTACK – použít alternativní zásobník pro zpracování signálu(viz sigaltstack(2)).

SA\_NOCLDWAIT – pro SIGCHLD: proces nečeká na potomky (nemůže volat wait()) a potomci nevytvářejí zombie. Takto funguje pid 1 (init) aby nemusel zpracovávat sirotky

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

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

**I/O multiplexing**

Jak kopírovat data mezi dvěma procesy? Pomocí read/write není moc super, jde to -> musí se použít v neblokujícím režimu ale i tak je to aktivní čekání – pálí čas:

while (!done) {

if ((n = read(fd1, buf, bufsiz)) <= 0)

break;

if (write(fd2, buf, n) < 0)

error\_message(”write to fd2”);

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

break;

if (write(fd1, buf, n) < 0)

error\_message(”write to fd1”);

}

-Polling – viz výše, pořád se dívám zda nejsou data, není to efektivní

-asynchronní I/O -přehození IO uvnitř jádra do jiných vláken pro read/write

-vlákna -co vlákno to descriptor – může se zablokovat, ale pro tisíce deksriptorů…

-selektivní čekání (událostně řízené programy) – definuji si které dekriptory mě zajímají a řeknu jádru ať mě vzbudí až na některém budou data a pak se uspím

**Rozhraní pro multiplexing**

select(2) – původně z BSD, první podporoval TCP IP, byla nutnost komunikace s vice uzly

poll(2) – System V, nedělá poll, ale podobně jak select

kqueue(2) – FreeBSD - mechanismus jak do procesů pomocí jedné fronty přidat události, ne jen čekání na FD, ale třeba i signály, je to buffer kde mi jádro ukládá data a zasílá mi je, když dojdou 🡪 volání kqueue blokuje

/dev/poll – Solaris jako kqueue ale přes soubor

/dev/epoll, eventfd(2) – Linux

**Množiny deskriptorů**

**FD\_\*(3)**

#include <sys/select.h>

fd\_set set;

FD\_CLR(int fd, fd\_set \*pset);

FD\_SET(int fd, fd\_set \*pset);

FD\_ISSET(int fd, fd\_set \*pset);

FD\_ZERO(fd\_set \*pset);

Není zde natstavi na vše – není to omezené

**select(2)**

#include <sys/select.h>

struct timeval {

long tv\_sec; /\* sekundy \*/

long tv\_usec; /\* mikrosekundy \*/

};

int select(int n, fd\_set \*rdset, fd\_set \*wrset, fd\_set \*exset, struct timeval \*timeout);

wrset – množina dekriptorů u kterých mě zajímá write, ponechá ty na kterých událost nastala

rdset – množina u kterých mě zajímá čtení, kernel tam ponechá ty na kterých událost nastala

exset – množina u kterých mě zajíma nějaká výjimka, ponechá ty u kterých nastala – EOF, IO\_ERROR..

Parametr n – horní limit velikosti množiny dekriptorů (nejvyšší číslo dekriptoru aby kernel věděl jak daleko má dekriptory procházet)

Vrací ty deskriptory, kde nastala očekávaná událost.

Návratová hodnota: počet připravenýchdeskriptorů.

Poslední parameter je limit jak dlouho čekat, může se do něj ukládat kolik ještě zbývá než by timeoutnul.

**poll(2)**

#include <stropts.h>

#include <poll.h>

int poll(struct pollfd fdarray[], unsigned long nfds, int timeout\_ms);

struct pollfd {

int fd;

short events; 🡨 maska událostí na které čeká

short revents; 🡨 maska událostí které nastaly

};

Samostatné vstupní a výstupní parametry (events, revents). Efektivnější pro vice fd, nemusím v každé smyčce definovat které FD chci – select se chová destruktivně

Parametr events a revents je logický součet některých

z následujících hodnot:

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

POLLOUT – data normální priority mohou být zapsána bez blokování.

POLLWRNORM – totéž jako POLLOUT.

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

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

POLLERR – došlo k chybě na příslušném deskriptoru.

POLLHUP – došlo k zavěšení linky.

POLLNVAL – deskriptor tohoto čísla není otevřen.

**Pokročilé I/O operace**

**Zamykání souborů**

Zamykání pro čtení nebo pro zápis – ve skutečnosti sdílený a výlučný zámek.

Nepovinné (implicitně, informativní - nebrání mi v operaci) a povinné zamykání (není všude, platí u souborů které nejsou spustitelné a má setgid bit)

Nepovinné – jen vzhledem k dalším zámkům. Povinné – i vzhledem k I/O operacím.

**Vlastnosti zámků**

Zámek přísluší souboru (i-uzlu): přes víc procesů, přes případné hard linky. Zámek přetrvá přes volání exec(2).

Metody zamykání – fcntl(2) (POSIX.1, SysV, 4.4BSD), lockf(2) (SysV), flock(2) (BSD).

**Zamykání přes fcntl(2)**

**fcntl(2)**

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

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

struct flock {

short l\_type; typ výlučný/sdílený

off\_t l\_start; zamykým od bajtu X

short l\_whence; SEEK\_CUR/SET/END – start je offsetem k tomuto

off\_t l\_len; jak dlouhý blok zamykám

pid\_t l\_pid; kdo zamyká

};

Argumenty cmd – co chci dělat:

F\_GETLK – zjistí, jestli je vytvoření zámku blokováno nějakým jiným zámkem. Pokud takový

zámek existuje, je struktura flock naplněna popisem tohoto zámku. Pokud neexistuje, je

l\_type změněno na F\_UNLCK.

F\_SETLK – nastaví nebo zruší zámek. Není-li možno zámek vytvořit, vrátí –1 s errno rovno

EACCESS nebo EAGAIN.

F\_SETLKW – blokující verze F\_SETLK. Pokusí se nastavit zámek. Pokud to není možné, zablokuje se do doby, než bude možné zámek vytvořit nebo než přijde signál.

cmd F\_GETLK, F\_SETLK nebo F\_SETLKW

arg je ukazatel na strukturu flock.

l\_type – typ zámku:

F\_RDLCK – zámek pro čtení,

F\_WRLCK – zámek pro zápis,

F\_UNLCK – pro zrušení zámku.

l\_start – offset prvního bajtu zamykaného regionu.

l\_whence – jedno z SEEK\_SET, SEEK\_CUR nebo

SEEK\_END (viz lseek(2)).

l\_len – délka zamykaného regionu. Je-li nulové, značí zámek od l\_start do konce souboru.

l\_pid – PID procesu, který drží zámek (vrací F\_GETLK).

**Scatter-gather I/O**

Čtení do nesouvislého datového prostoru

Zápis z nesouvislého datového prostoru

Jedna služba jádra – ušetří se přepnutí do jádra a zpět (nebo kopírování dat do souvislého bufferu).

Moderní hardware – umí scatter/gather přímo.

**readv(2) scatter read**

#include <sys/types.h>

#include <sys/uio.h>

ssize\_t readv(int fd, struct iovec iov[], int iovcount);

struct iovec {

void \*iov\_base;

size\_t iov\_len;

};

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

**writev(2) gather write**

#include <sys/types.h>

#include <sys/uio.h>

ssize\_t writev(int fd, struct iovec iov[], int iovcount);

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

**Memory-mapped I/O**

Mapování (části) souboru do paměti

Sdílená paměť mezi programy – knihovny atd – soubor je obrazem paměti

Urychlení I/O – ušetří se kopírování dat, lze namapovat celý soubor naráz

Alokace paměti – namapování /dev/zero.

Nevýhoda – zneplatnění TLB při změně mapování.

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

#include <unistd.h>

#include <sys/mman.h>

#ifdef \_POSIX\_MAPPED\_FILES

void \*mmap(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset);

#endif

Start - kam chci mapovat, length – kolik namapovat, prot – práva přístupu k namapované paměti, fd – který soubor mapovat, offset – odkud mapovat

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

Parametr prot:

PROT\_EXEC – stránky mohou být prováděny.

PROT\_READ – stránky jsou přístupné pro čtení.

PROT\_WRITE – do stránek lze zapisovat.

PROT\_NONE – ke stránkám nelze přistupovat.

FLAGS:

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

MAP\_SHARED – zápis do mapované oblasti se projeví v namapovaném souboru i v paměti dalších procesů, které si tento úsek souboru namapovaly. Časem uvidí všechny procesy.

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

**munmap(2) Odmapování souboru**

#include <unistd.h>

#include <sys/mman.h>

#ifdef \_POSIX\_MAPPED\_FILES

int munmap(void \*start, size\_t length);

#endif

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

**msync(2) Synchronizace regionu**

#include <unistd.h>

#include <sys/mman.h>

#ifdef \_POSIX\_MAPPED\_FILES

#ifdef \_POSIX\_SYNCHRONIZED\_IO

int msync(const void \*start, size\_t length, int flags);

#endif

#endif

Aktualizace zápisu do sdílené paměti – aby to uviděli všichni (i když jsou to ty stejné stránky takže je to vždy viditelné hned, jen se předpokládá že by to mohlo běžet někde kde je např pro vice procesorů více pamětí 🡪 reálně to funguje I bez msync())

MS\_SYNC – služba počká na dokončení zápisu na disk.

MS\_ASYNC – služba nastartuje zápis, ale skončí bez čekání na dokončení diskových operací.

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

Právě jedno z MS\_SYNC a MS\_ASYNC musí být nastaveno.

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

#include <sys/mman.h>

int mprotect(const void \*addr, size\_t len, int prot);

Parametr prot stejný jako u mmap(2).

Parametr addr musí být zarovnán na velikost stránky.

POSIX.4 (1b) říká, že mprotect(2) může být použito pouze na regiony získané pomocí mmap(2).

**Zákaz swapování**

**mlock(2), munlock(2)**

#include <sys/mman.h>

int mlock(const void \*addr, size\_t len);

int munlock(const void \*addr, size\_t len);

Nedědí se přes fork(2) a exec(2). Využití – real-time aplikace (musí běžet hned, ne swapovat do paměti), kryptografické aplikace (uložení klíče)

Setrlimit 🡪 lze omezit velikost takto blokované paměti

**mlockall(2), munlockall(2) zamčení celé VM 🡪 jen superuser**

#include <sys/mman.h>

int mlockall(int flags);

int munlockall();

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

MCL\_FUTURE – i regiony mapované v budoucnu budou zamčeny.

Jak si namapovat zásobník (real time aplikace ho chtělí hned v paměti a zamčený) 🡪 stroj ho naslibuje ale nenaalokuje, nechci v realtime appce swapovat 🡪 zavolat alloca() na velkém bufferu – donutím ho naalokovat – nahrát do paměti a pak lze zamknout.

**Vlákna**

Vlákno – thread – *light-weight process*.

Kontexty činnosti procesoru pracující and sdílenými prostředky (paměť, fd..)

Vlákna jednoho procesu

stejná VM

jiný zásobník, čítač instrukcí, registry

stejné deskriptory

stejný pracovní adresář

a další sdílené atributy

Vlákna v UNIXu – IEEE POSIX 1003.1c (POSIX threads).

Proč vlákna?

využití více procesorů

paralelizace diskových operací

Kdy ne vlákna – tam kde lze použít událostně řízené programování (GUI aplikace) nebo samostatné procesy (síťové servery).

1:N (user-level threads) – například balík pthreads. Vlákna neexistují – jen se to podle knihovny chová vícevláknově – pokud jedno zastaví může pustit jiné, nevyužije více procesorů.

1:1 (kernel-level threads) – LinuxThreads a NPTL – new posix thread library. Reálná vlákna – jádro to dělá samo a my se nemusíme o nic starat – vlastně jsou to procesy se sdílenými zdroji

M:N – kombinace obojího – např. scheduler activations ve FreeBSD; IRIX. Problém atomicity, ale lze si udělat X procesů pro každý processor a pak tisíce procesů na userspace co nezatěžují jádro, implementace signálem od jádra pokud chce uspat vlákno dá vědět knihovně zda nechce pustit jiné z těch userspace)

Linux – clone() kde se dá nadefinovat co se má sdílet (vše – vlákno, nic – nový proces, či mix)

**pthread\_create(3) vytvoření vlákna**

#include <pthread.h>

int pthread\_create(pthread\_t \*thread, pthread\_attr\_t \*attr, void (\*start\_routine)(void \*), void \*arg);

Vytvoření vlákna. Identifikace vlákna je uložena do thread. Parametr attr určuje další vlastnosti vlákna. start\_routine je funkce a arg její argument co se jí posílají.

**pthread\_exit(3) zrušení vlákna**

#include <pthread.h>

void pthread\_exit(void \*retval);

Ukončí vlákno, zavolá registrované funkce pro dobu ukončení a uvolní lokální data vlákna.

**pthread\_join(3) počkat na ukončení – verze wait() pro procesy**

#include <pthread.h>

int pthread\_join(pthread\_t tid, void \*\*retval);

Počká na ukončení vlákna (blokující funkce) a získá návratovou hodnotu.

**Synchronizace vláken**

Mutex – mutual exclusion – vzájemné vyloučení vláken – v podstatě semafor, pokud pracuje s nějakým zdrojem tak může pracovat sám a jiná vlákna co by chtěla modifikovat musí čekat

Stavy – odemčený zámek/zamčený zámek.

Vzájemné vyloučení – v jednu chvíli může držet zámek zamčený nejvýše jedno vlákno.

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

pthread\_mutex\_t fmutex = PTHREAD\_MUTEX\_INITIALIZER; (běžný)

pthread\_mutex\_t rmutex = PTHREAD\_RECURSIVE\_MUTEX\_INITIALIZER; (pamatuje si kdo blokuje –

projde dvakrát zamčení od toho samého procesu)

pthread\_mutex\_t emutex = PTHREAD\_ERRORCHECK\_MUTEX\_INITIALIZER; (kontrola chyb – pozná že

je zamčený mutex I když nikdo nedrží a pod)

int pthread\_mutex\_lock(pthread\_mutex\_t \*mtx); 🡨 zablokuje nebo pustí

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mtx); 🡨 získám zámek nebo chyba - neblokuje

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mtx);

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mtx); 🡨 opak phread\_mutex\_init() / inicializace

konstantou PTHREAD\_MUTEX\_INITIALIZER

**Podmínkové proměnné**

Podmínková proměnná – hlášení o události jinému vláknu –> pomocí mutexu je to neohrabané (musel by to aktivně zkoušet nemůže se uspat – problem jako u signálů mohl by se mutex odemčít dříve než ho stihnu zamčít a tak ho potom zamknu a odemčení už nedojde), proměnná má v sobě zámek

Strany komunikace – vlákno čeká na podmínku, vlákno signalizuje podmínku.

Podmínková proměnná má asociovaný zámek.

Čekání na podmínku – atomické odemčení mutexu a zablokování vlákna. Mutex musí být předem zamčený. Při ukončování funkce se mutex opět zamče.

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

pthread\_cond\_t c = PTHREAD\_COND\_INITIALZIER; 🡨 zamknutí mutexu (jako synchroznizace signály)

int pthread\_cond\_wait(pthread\_cond\_t \*c, pthread\_mutex\_t \*mutex); 🡨 odemkne mutex a

atomicky současně uspí proces

int pthread\_cond\_timedwait(pthread\_cond\_t \*c, pthread\_mutex\_t \*mutex, struct timespec \*abstime); uspí se na timeout

int pthread\_cond\_signal(pthread\_cond\_t \*c); 🡨 jako wait\_one()

int pthread\_cond\_broadcast(pthread\_cond\_t \*c); 🡨 probudí všechny kdo čekají

int pthread\_cond\_destroy(pthread\_cond\_t \*c);

**Soukromá data vlákna**

Globální proměnná – ale v každém vlákně s jinou hodnotou.

Důvod použití – není nutno předávat jako argument do všech funkcí.

Klíč – konkrétní kus dat, v každém vlákně vrací jinou hodnotu.

Destruktor – při ukončení vlákna se volá pro jeho nenulové klíče.

**pthread\_\*specific(3)**

pthread\_key\_t list\_key; 🡨 globální proměnná

extern void\* cleanup\_list(void\*);

pthread\_key\_create(&list\_key, cleanup\_list);

int\* p\_num = (int \*)malloc(sizeof(int));

(\*p\_num) = 4; /\* Nejaka hodnota \*/

pthread\_setspecific(list\_key, (void \*)p\_num);

/\* Nekde uplne jinde \*/

int\* p\_keyval = (int\*)pthread\_getspecific(list\_key);

/\* a nakonec \*/

pthread\_key\_delete(list\_key);

**Vlákna – další vlastnosti**

Zrušení vlákna – pthread\_cancel(3). Vlákno může být zrušitelné jen v některých bodech. (může si nastavit kde se může zrušit – vypnout si možnost zrušení a definovat body kde se dá zrušit)

Odpojení vlákna – pthread\_detach(3). Není pak možno/nutno vlákno připojovat přes

pthread\_join(3). Lze specifikvoat při tvoření nebo později – tímto.

Jednorázová inicializace – pthread\_once(3). Zavolání pouze při prvním použití. Inicializace – spuštění funkce – pouze jednou, zbytek vláken neprovede.

Identifikace vlákna – pthread\_self(3). V /proc/[pid]/task/[thread id]

**Vlákna a signály**

Zaslání signálu – pthread\_kill(3) – parametrem číslo vlákna – zasílání mezi sebou

Cekání na signál – sigwait(3) – může čekat, signál je doručen přednostně tomu kdo čeká

Synchronní signál – doručen vláknu které signál vygenerovalo.

Asynchronní signál – doručen některému vláknu, které signál neblokuje.

Maska blokovaných signálů – pro každé vlákno zvlášť – viz pthread\_sigmask(3). Doručí se tomu kdo nemá zablokované, když mají všichni tak ho odloží a počká až někdo odblokuje.

**Vlákna a soubory**

Vlákna sdílí file descriptory 🡪 sdílí ukazovátko souboru, by se mlátili

**pread(3) čti ze souboru**

ssize\_t pread(int fd, void \* buf, size\_t count, off\_t offset);

**pwrite(3) zapiš do souboru**

ssize\_t pwrite(int fd, void \* buf, size\_t count, off\_t offset);

pouze abolutní adresy zápisu (ne SEEK\_CUR atd..), nemodifikují posouvátko – pro vlákna