

Programování v Unixu

Petr Šťastný, 15.2.2007

Vývojové nástroje

Popište činnost kompilátoru jazyka C a linkeru.

- **preprocesor** – provede includování, expanzi maker, vynechání komentářů
- **překladač** – přeloží zdrojový kód do objektových modulů, které místo volání externích funkcí obsahují tabulku jejich jmen
- **linker** – zkompletuje výsledný program, vyřeší externí odkazy mezi moduly navzájem a mezi moduly a knihovnami, vyprodukuje spustitelnou binárku (např. ve formátu ELF)

Vysvětlete princip utility make a napište příklad jednoduchého makefile pro překlad a slinkování programu v C

Pro překlad programu, skládajícího se z více zdrojových souborů, slouží utilita make, která rozpozná, které části se změnilo od posledního překladu a provede kompilaci jen změněných objektů a objektů, které na nich závisí.

Konfigurační soubor programu make – Makefile – se skládá ze seznamu cílů. U každého cíle lze uvést další objekty, na kterých je tento cíl závislý, a dále příkaz, který se provede v případě, že cílový objekt neexistuje nebo se některá ze závislostí změnila (tj. je mladší než cílový objekt).

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

Jádro a C API UNIXu

Vysvětlete význam parametrů funkce main() a proměnné environ. Jak se zpracovávají argumenty programu pomocí funkce getopt()?

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

- **argc** – počet argumentů příkazové řádky
- **argv** – pole ukazatelů na řetězce argumentů, argv[0] je název programu (započítává se do argc), pole ukončeno NULL pointerem
- **envp** nebo proměnná **extern char ** environ** – pole ukazatelů na řetězce ve tvaru „proměnná=hodnota“, ukončeno NULL pointerem

Proměnné prostředí lépe zpracovávat funkcemi getenv(), putenv(), setenv(), unsetenv(). Proměnné prostředí se při vzniku synovského procesu přenesou z otce, jinak je nelze mezi procesy měnit (každý má svou kopii).

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

- **argc** – počet argumentů příkazové řádky
- **argv** – pole řetězců argumentů příkazové řádky
- **optstring** – řetězec se seznamem povolených přepínačů a příp. označení očekávání hodnoty
- **optarg** – hodnota posledního přepínače
- **optind** – číslo prvního nezpracovaného argumentu po ukončení zpracování přepínačů v getopt()
- **optopt** – znak neznámého přepínače (getopt() vrací při chybě znak '?')
- **opterr** – příznak, zda má vypisovat chybová hlášení o chybných přepínačích

Vysvětlete význam pojmů proces, vlákno (thread), program. Čím se liší proces a vlákno?

- **proces** – systémový objekt charakterizovaný kontextem, identifikuje se pomocí PID; „kód a data v paměti“
- **vlákno (thread)** – systémový objekt uvnitř procesu, charakterizován stavem; „linie výpočtu“; „to co běží“; všechna vlákna jednoho procesu sdílí paměť, ale mají vlastní registry a zásobník
- **program** – spustitelný soubor na disku; soubor přesně definovaného formátu obsahující instrukce, data a služební informace nutné ke spuštění
- **kontext** – paměťový prostor procesu, registry a datové struktury jádra, týkající se procesu

Vysvětlete pojmy knihovní funkce a systémové volání. Popište konvenci, podle které systémová volání a knihovní funkce oznamují úspěšné ukončení nebo chybu.

- knihovní funkce – provádějí se v uživatelském režimu jako ostatní kód samotného programu
- systémová volání – zpracování argumentů a předání řízení jádru pomocí synchronního přerušení (procesor se přepne do privilegovaného režimu), poté úprava výsledků a jejich předání zpět volajícímu
- celočíselná návratová hodnota
 - ≥ 0 – operace úspěšně provedena
 - $= -1$ – chyba
- návratová hodnota typu ukazatel
 - $\neq \text{NULL}$ – operace úspěšně provedena
 - $= \text{NULL}$ – chyba
- `extern int errno` – kód chyby (nemaže se při úspěšném volání!)
- `void perror(char * s)` – výpis chybového hlášení

Co obsahují soubory `/etc/passwd` a `/etc/group`? Kdy a jak používá Unix informace z těchto souborů.

```
login:x:uid:ID primární skupiny:plné jméno:domácí adresář:login shell
název*:GID:členové skupiny
```

Jádro o těchto souborech vůbec neví, pracuje pouze s číselnými identifikátory uživatelů a skupin. S `/etc/passwd` a `/etc/group` se většinou pracuje pomocí knihovních funkcí.

Jak se používá identifikace vlastníka a skupiny pro testování přístupových práv? Jaký je rozdíl mezi reálným a efektivním vlastníkem procesu?

<code>if (UID_p == 0)</code>	... všechna práva
<code>else if (UID_p == UID_s)</code>	... práva vlastníka
<code>else if (GID_p == GID_s GID_s je v SUPG)</code>	... práva člena skupiny
<code>else</code>	... práva ostatních

- UID_p – ID uživatele procesu
- GID_p – ID skupiny procesu
- SUPG – supplementary GIDs (seznam skupin, ve kterých je uživatel, s výjimkou primární skupiny)
- UID_s – ID vlastníka souboru
- GID_s – ID skupiny souboru
- **RUID** – reálné UID – skutečný vlastník procesu
- **EUID** – efektivní UID – uživatel, jehož práva proces používá
- **SUID** – saved UID – původní EUID, uschovávání propůjčeného uživatele

Systém souborů

Jaké objekty jsou v UNIXu přístupné pomocí rozhraní systému souborů? Jaký je rozdíl mezi znakovými a blokovými zařízeními?

- skutečné soubory
- periferní zařízení (hlavní a vedlejší číslo ovladače)
- pojmenované roury
- sokety (v doméně UNIX)
- procesy (`/proc`)
- paměť (`/dev/mem/`, `/dev/kmem`)
- pseudosoubory (`/dev/tty`, ...)
- **znakové** – data se přenáší přímo mezi procesem a ovladačem zařízení, např. sériové porty
- **blokové** – data prochází bufferem, čtení dat z ovladače po blocích, např. disky

Popište strukturu svazku typu `s5` a jeho vylepšení (`ufs`).

- **boot block** - zaváděcí blok, uložení zavaděče OS
- **superblok** – základní informace o svazku: počet bloků pro i-nody, počet bloků svazku, seznam volných bloků, seznam volných i-nodů, informace o zařízení, ...
- oblast i-nodů
- oblast datových bloků
- **i-node** – obsahuje informace o souboru s výjimkou jeho jména – typ, práva, vlastník, skupina, velikost, počet odkazů, 10 odkazů na datové bloky, 3 odkazy na nepřímé bloky, ...
- jména souborů max. 14 znaků

UFS z důvodu snížení fragmentace souborů a omezení pohybu hlav na disku zavádí členění svazku na skupiny cylindrů, každá skupina obsahuje kopii superbloku, řídicí blok skupiny, tabulku i-nodů, bitmapy volných i-nodů a datových bloků. Jména dlouhá až 255 znaků.

Vysvětlete princip navigace ve struktuře adresářů, tj. jak jádro najde podle cesty k souboru příslušné datové bloky. Vysvětlete rozdíl mezi pevnými (hardlink) a symbolickými (symlink) odkazy na soubory.

Pokud cesta začíná znakem '/', začíná navigace v kořenovém adresáři (většinou i-node 2), jinak v pracovním adresáři procesu. Datový blok adresáře obsahuje seznam souborů (a podadresářů), čísla jejich i-nodů a také odkaz na sebe sama (.) a na nadřazený adresář (..).

- **hardlink** – druhé jméno souboru, odkaz na stejný i-node, není rozdíl mezi originálem a linkem, lze vytvářet jen v rámci jednoho svazku, nelze vytvářet pro adresáře (na některých systémech ano, hrozí zacyklení cesty)
- **symbolický link** – pouze odkaz na skutečnou cestu, speciální typ souboru, obsahující absolutní či relativní cestu cíle, odlišné chování pro originál a link, může odkazovat na adresář a také na neexistující soubor, lze vytvářet mezi svazky

Jaká jsou přístupová práva k souborům? Co je to propůjčování práv (set UID)?

U každého souboru jsou uvedena práva zvlášť pro vlastníka, skupinu a ostatní pro zápis, čtení a spuštění. Další trojice bitů je suid, sgid a sticky.

- SGID pro soubor bez práva spuštění skupinou: kontrola zámků při každém přístupu
- sticky bit pro adresáře: právo mazat a přejmenovávat soubory mají jen vlastníci souborů (např. /tmp)
- SGID pro adresář: nové soubory budou mít stejnou skupinu jako adresář

Příznaky SUID a SGID u spustitelných souborů umožňují propůjčit uživateli, který je spouští, práva vlastníka nebo skupiny spouštěného souboru. Uživatelé tak mohou provádět některé operace s oprávněními vyššími, než mají oni sami (např. passwd).

Jak procesy přistupují k otevřeným souborům? Jak se liší deskriptor a otevření souboru?

Každý proces má svoji tabulku souborových deskriptorů. Z ní vede odkaz na systémovou tabulku otevřených souborů (jedna pro celý systém), v níž je mód otevření souboru a aktuální pozice v souboru. Procesy přistupují k otevřeným souborům podle deskriptoru, získaném při jejich otevření. Z tabulky otevřených souborů vede odkaz do tabulky i-nodů. Tento způsob umožňuje, aby procesy sdílely pozici v souboru.

Co je to virtual file system? K čemu slouží a jaký je princip jeho fungování?

VFS je mechanismus pro jednotný přístup jádra k souborům bez rozdílu použitého filesystému. Umožňuje na jednom stroji pracovat se svazky různých typů. Struktura **vnode** (virtual node), na kterou ukazuje otevření souboru, obsahuje část nezávislou na konkrétním fs a strukturu inode, specifickou pro každý typ souborového systému. Každý typ fs definuje tabulku funkcí pro jednotlivé operace, na kterou se odkazují všechny vnodes odpovídající souborům na všech svazcích daného typu.

Jaký je vztah diskového oddílu a svazku? Jak probíhá vytvoření, kontrola konzistence, připojení a odpojení svazku?

Systém souborů (svazek, filesystem) lze vytvořit na:

- **oddílu disku** (partition) – část disku, na jednom disku může být více oddílů
- **logickém oddílu** (logical volume) – spojení více oddílů do jednoho svazku (oddíly mohou být na různých discích)

Svazek se vytváří nad oddílem pomocí příkazu **mkfs**, připojuje se příkazem **mount** k adresáři některého již připojeného svazku a odpojuje se příkazem **umount**.

Pokud není svazek před zastavením systému korektně odpojen, mohou být data v nekonzistentním stavu. Ke kontrole slouží příkaz **fsck**. Žurnálovací systémy nepotřebují kontrolu celého svazku, v žurnálu je vidět, jaké operace se před zastavením systému prováděly a v jakém byly stádiu.

Problém s výkonem dělá zápis metadat, řeší se asynchronním zápisem (nejprve do paměti, později na disk). Možná nekonzistence při asynchronním zápisu metadat se řeší žurnálováním (atomický zápis závislých operací do žurnálu a teprve poté provedení samotných změn).

Procesy a roury

Popište paměťový prostor procesu v uživatelském režimu a v režimu jádra.

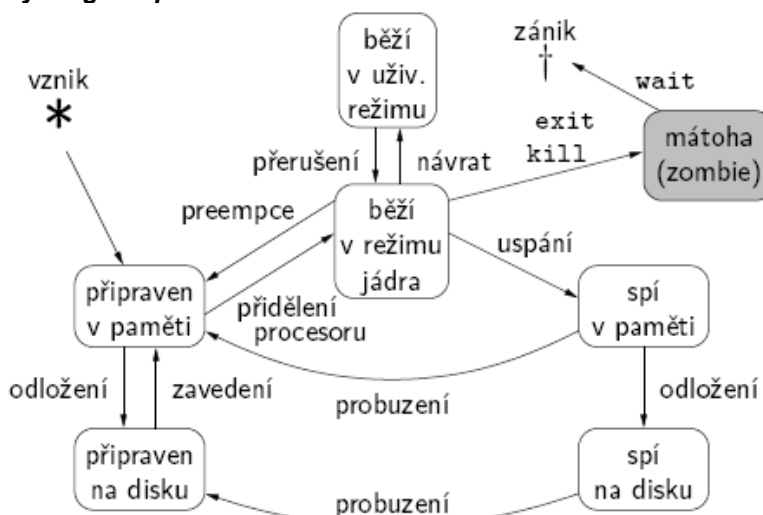
uživatelský režim:

- text – kód programu
- data – inicializované proměnné
- bss – neinicializované proměnné
- uživatelský zásobník – automaticky roste podle potřeby
- oblast user (u-area) – informace o procesu používané jádrem, které nejsou potřeba při odložení procesu na disk

režim jádra:

- text jádra – v paměti je pouze jedna kopie, sdílená všemi procesy
- data a bss jádra – tabulky, proměnné, ..., součástí je také u-area aktuálně běžícího procesu
- zásobník jádra – samostatný pro každý proces, není přístupný v uživatelském režimu

Nakreslete a popište stavový diagram procesu



Popište základy mechanismu plánování procesů. Jaké jsou prioritní třídy?

- **preemptivní plánování** – procesor je procesu odebrán po uplynutí časového kvanta (pokud se dříve procesor nezdá sám) – základem jsou pravidelná přerušení od časovače
- **nepreemptivní (kooperativní) plánování** – přeplánování pouze v okamžiku, kdy se proces sám vzdá procesoru
- procesy řazené do **fronty podle priority**, procesor přidělen vždy prvnímu procesu z fronty s nejvyšší prioritou

Prioritní třídy:

- **real-time** (100-159) – pevná priorita, definováno časové kvantum, garance prodlevy mezi běhy procesu
- **systémová** (60-99) – pro systémové procesy, pevná priorita
- **sdílení času** (0-59) – dvousložková priorita: pevná uživatelská a proměnná systémová část; pokud procesor hodně využívá CPU, je mu snižována priorita a zvětšováno časové kvantum

Popište mechanismus mapování souborů do paměti. Popište, jak lze za běhu programu získat přístup k proměnným a funkcím definovaným v nějaké dynamické sdílené knihovně.

Při mapování souborů do paměti se pro popis úseku sdílené paměti používá soubor, se kterým se pak pracuje pomocí instrukcí práce s pamětí a odpadá velká režie systémových volání `read()` a `write()`.

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

Funkce od adresy `addr` (0=systém zvolí) namapuje úsek délky `len` začínající na pozici `off` souboru s deskriptorem `fildes`. Vrací adresu namapovaného úseku nebo `MAP_FAILED`.

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

Zpřístupní knihovnu v souboru `file`, vrátí handle nebo `NULL`

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

Vrátí adresu symbolu daného jména z knihovny

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

Signály

Co jsou to signály? Jak může být pro proces vygenerován signál? Jaký je rozdíl mezi posláním signálu procesu a vláknu?

Signál je zpráva předaná procesu, informující o vzniku nějaké události. Nese informaci o vzniku signálu a jeho číslo, nemůže přenášet žádné další informace. Signály se zpracovávají asynchronně – příchod signálu přeruší běh procesu a vyvolá handler.

- chybové události – nepovolený přístup do paměti, použití nepovolené instrukce, ...
- asynchronní události – signál poslaný jiným procesem, vypršení timeoutu, odpojení terminálu, stisk Ctrl-C, ...

Signál určený pro proces obslouží právě jedno vlákno, které nemá tento signál zablokovaný. Masku blokových signálů mají vlákna vlastní, handlers jsou společné pro všechny. Signály pro konkrétní vlákno se posílá pomocí `pthread_kill()`.

Jaké jsou možnosti ošetření signálů? Jak se nastavuje ošetření signálů (handlers, blokování) pro proces a pro vlákna?

Možnosti ošetření signálu:

- implicitní akce
- ignorování (systém signál zahodí)
- ošetření handlerem (uživatelsky definovanou funkcí), po návratu z handleru proces pokračuje od místa přerušení
- blokování signálů pomocí masky – zablokované signály se zpracují až po jejich odblokování (ve frontě)

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

Nastaví handler signálu, vrátí původní nastavení. Struktura `sigaction` obsahuje informace o handleru, masce blokových signálů a další příznaky.

Handlers jsou společné pro všechna vlákna, signál určený pro proces obslouží právě jedno vlákno, které nemá tento signál zablokovaný. Masky pro blokování signálů má každé vlákno vlastní.

Vlákna

Co je to vlákno, jak se liší od procesu? Které atributy jsou společné pro proces, které jsou privátní pro každé vlákno? Jak je možné vytvořit globální proměnnou privátní pro jedno vlákno?

Vlákno je linie výpočtu v procesu. V jednom procesu může běžet více vláken najednou. Vlákna sdílejí globální proměnné procesu, ale mají vlastní čítač instrukcí, zásobník, plánovací prioritu a politiku, hodnotu `errno` a klíčované hodnoty.

Privátní globální proměnné lze vytvořit pomocí klíčovaných hodnot. Klíče jsou viditelné v celém procesu, ale hodnotu má každé vlákno vlastní.

- `pthread_key_create()`, `pthread_key_delete()`,
- `pthread_setspecific()`, `pthread_getspecific()`

Popište postup při vytvoření a zrušení vlákna. Jak fungují destruktory klíčovaných hodnot a zásobník úklidových handlerů? `fork()` a vlákna.

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

Vytvoří nové vlákno a v něm spustí funkci `start_fun(arg)`

```
void pthread_exit(void *value_ptr)
```

Ukončí volající vlákno, obdoba `exit()` pro proces.

```
int pthread_join(pthread_t thread, void **value_ptr)
```

Počká na ukončení vlákna `thread` a ve `value_ptr` vrátí hodnotu ukazatele z volání `pthread_exit()` nebo návratovou hodnotu hlavní funkce vlákna.

Vlákno má zásobník úklidových handlerů, které se volají při ukončení nebo zrušení vlákna. Jako první se spouští naposledy vložený handler. Práce se zásobníkem:

```
void pthread_cleanup_push()  
void pthread_cleanup_pop()
```

Po provedení handlerů se volají destruktory na klíčované hodnoty pro všechny klíče s hodnotou různou od NULL. Aktuální hodnota je předána destruktoru. Volání destruktora probíhá stále dokola, dokud nejsou všechny hodnoty NULL (destruktor ji musí vynulovat).

Při **fork()** procesu obsahuje synovský proces přesnou kopii volajícího vlákna včetně stavů mutexů a dalších zdrojů. Ostatní vlákna v synovském procesu neexistují, jimi naalokovaná paměť zůstane ztracená.

Uveďte nástroje pro synchronizaci vláken.

- **Mutexy** – zámky pro ohraničení kritických sekcí. Je to forma binárního semaforu, rozdíl je ten, že mutex může odemknout pouze to vlákno, které ho zamklo. Mutexy jsou určeny ke krátkodobému držení a měly by fungovat rychle.
- **Podmínkové proměnné** – slouží k signalizaci změn v datech mezi vlákny. Každá podmínková proměnná je vždy asociována s jedním mutexem. Při použití p.p. vlákno testuje nějakou podmínku nad sdílenými daty. Pokud není splněna, uspí se použitím podmínkové proměnné. Jakmile jiné vlákno sdílená data změní, signalizuje ostatním, že se tak stalo, a vlákno (nebo vlákna) se probouzí a testují, zda je podmínka již splněna.
- **Read-write zámky** – najednou může mít zámek buď několik vláken zamčeno pro čtení nebo maximálně jedno vlákno pro zápis (a nikdo pro čtení).
- **Bariéry** – čeká se, až se všechna vlákna sejdou na bariéře a pak jsou hromadně puštěna dál. Lze implementovat pomocí mutexů a podmínkových proměnných.
- **Semaforey** – zobecnění mutexů

Synchronizace a zamykání

Vysvětlete vznik konfliktu při přístupu ke sdíleným datům a jeho řešení pomocí zamykání. Jak může vzniknout deadlock? Popište zamykání souborů pomocí fcntl().

Konflikt vzniká, pokud více procesů provádí současně operace (čtení, zápis) nad sdílenými daty, a tyto operace nejsou atomické. V takové situaci může dojít k nekonzistenci dat.

Řešením je ohraničit tzv. kritickou sekci (část programu, kterou může v jednom okamžiku provádět pouze jeden proces) zámky – před vstupem do kritické sekce zámek získat a po opuštění ho vrátit. Pokud má zámek jiný proces, musí se čekat. Zámky mohou být rozlišeny na zámky pro čtení a pro zápis. Zámek pro čtení může mít současně více čtoucích procesů, ale pro zápis jen jeden zapisující proces. Zámky se nemohou kombinovat.

Deadlock nastává v okamžiku, kdy proces čeká na událost, která nemůže nastat. Např. pokud 2 procesy navzájem čekají, až ten druhý uvolní nějaký zámek.

```
int fcntl(int fildes, int cmd, struct flock *lock)
```

- **F_GETLK** – vezme popis zámku z třetího argumentu a nahradí ho popisem zámku, který s ním koliduje
- **F_SETLK** – nastaví nebo zruší zámek popsany třetím argumentem; pokud zámek nelze nastavit, ihned vrátí -1
- **F_SETLKW** – jako F_SETLK, ale uspí proces, pokud není možné nastavit zámek

```
struct flock
```

- **l_type** – typ zámku (F_RDLCK, F_WRLCK, F_UNLCK – čtení, zápis, odemčení)
- **l_whence** – typ počáteční pozice – SEEK_SET, SEEK_CUR, SEEK_END
- **l_start** – začátek zamykaného úseku souboru
- **l_len** – délka úseku; 0 = do konce souboru
- **l_pid** – PID procesu držícího zámek (při použití F_GETLK)

Vysvětlete zamykání pomocí lock souborů.

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

Klíčem k úspěchu je použití příznaků O_EXCL v systémovém volání open() - vrátí chybu, pokud soubor existuje.

Je vhodné do souboru vkládat PID procesu, který ho vytvořil, pro případ, že tento proces zhavaruje a zámek neuvolní. Ostatní procesy mohou kontrolovat, zda tento proces stále existuje. Avšak může nastat problém, když neexistenci procesu zjistí 2 procesy naráz.

IPC

Popište semafore, implementované v UNIX System V IPC.

Semafor je metoda ohraničení kritické sekce, která povolí, aby se v kritické sekci mohl nacházet v jednom okamžiku jen daný počet procesů (pro binární semafor jen jeden proces). Čekající procesy jsou evidovány ve frontě. Proces, který opouští kritickou sekci, probudí první proces ve frontě.

Systém V je specifický tím, že provádí atomické operace nad celým polem semaforů – tj. operace se provede buď nad všemi nebo nad žádným, což zabraňuje deadlocku při práci více procesů nad více systémovými zdroji.

Sítě

Popište činnost serveru a klienta (posloupnost systémových volání) pro spojované síťové služby.

server:	klient:
fd = socket()	fd = socket()
bind(fd)	
listen(fd)	
fd2 = accept(fd)	connect(fd)
read(fd2); write(fd2)	write(fd); read(fd)
close(fd2)	close(fd)

Popište činnost serveru a klienta (posloupnost systémových volání) pro nespojované síťové služby.

server:	klient:
fd = socket()	fd = socket()
bind(fd)	
recvfrom(fd)	sendto(fd)
sendto(fd)	recvfrom(fd)
	close(fd)

Co je to soket? Jaké jsou varianty adresace soketů (v rámci jednoho stroje a po síti)? Jaké funkce (uved'te aspoň některé) slouží pro převod mezi číselnými a symbolickými jmény protokolů, portů a IP adres? Proč se používají funkce pro konverzi mezi lokálním a síťovým pořadím bajtů?

Soket je jeden konec obousměrného komunikačního kanálu vytvořeného mezi dvěma procesy buď lokálně na jednom počítači nebo s využitím síťového spojení.

V rámci jednoho stroje se adresace provádí pomocí domény **AF_UNIX**, adresy jsou jména speciálních souborů ve filesystému, které reprezentují sokety. Případně je možno použít loopback TCP/IP (127.0.0.1). Po síti probíhá adresace v doméně **AF_INET**, používána je především rodina protokolů TCP/IP s adresováním pomocí IP adres a portů.

- `getprotobyname()` - vrátí číslo protokolu podle jeho názvu (/etc/protocols)
- `getservbyname()` - pro zadané jméno služby a jméno protokolu vrátí číslo portu
- `gethostbyname()` - pro zadané jméno vrátí seznam příslušných síťových adres
- `gethostbyaddr()` - pro danou adresu vrátí jméno a případně další aliasy

Funkce pro konverzi pořadí bajtů se používají proto, že architektury používají pořadí různá. Podle toho, na jaké jsme architektuře, funkce buď nedělá žádné změny nebo pořadí obrací.

- big endian – SPARC, MIPS, ethernet
- little endian – Intel

Popište varianty sekvenční a paralelní obsluhy klientů TCP serveru.

- **sekvenční** – během obsluhování požadavku musí ostatní čekat ve frontě, dostane se na ně až po dokončení
- **paralelní obsluha** – přijaté spojení je předáno k obslužení jinému procesu či vláknu, hlavní proces hned čeká na spojení další
- **paralelní accept()** - více procesů sdílí soket pro přích. spojení, jádro mezi nimi nedeterministicky vybírá

Jak lze čekat na příchod dat z několika deskriptorů souborů zároveň? Jak byste tuto funkci použili při implementaci síťového serveru, který obsluhuje několik klientů zároveň jedním procesem bez použití vláken?

Systémovému volání **select()** nebo **poll()** lze předat množinu deskriptorů. Zjistí se, které z nich jsou připraveny pro čtení, zápis nebo na kterých došlo k nějakému výjimečnému stavu. Pokud žádný takový deskriptor není, čeká až do vypršení nastaveného timeoutu. Lze testovat také deskriptor použitý pro `accept()`, a tak testovat příchozí spojení na více soketech.

Administrace

Jak probíhá start operačního systému UNIX? Co jsou to úrovně běhu systému? Jak se spouští startovací skripty na různých systémech?

Start je závislý na architektuře systému, operačním systému a boot manageru. Nejprve se načte 1. blok z boot disku do paměti a skočí se na jeho první instrukci. Pak se načte první blok ze zvoleného oddílu na disku a skočí se na jeho první instrukci. Načte se jádro do paměti. Jádro provede svoji inicializaci, připojí kořenový svazek souborů, vytvoří nový proces a spustí program **init**, který provede další inicializace a spuštění startovních skriptů.

Úrovně běhu systému určují, v jakém režimu systém poběží – např. administrátorský režim, víceuživatelský režim, zastavení systému, restart systému, ... Podle toho se spouští příslušní startovací skripty, které spouští/zastavují služby a provádí další potřebné příkazy.

Co se děje v různých úrovních, určuje především skript `/etc/inittab`. Dále se spouští startovací skripty v `/etc/rc.X.d`, nejprve ty začínající na **K** (kill) a poté na písmeno **S** (start). Ve skutečnosti to jsou pouze symlinky na soubory, uložené v `/etc/init.d`.

- **rcNG** (rc Next Generation) na NetBSD – zavádí ve skriptech metainformace o závislostech
- **SMF** (Service Management Facility) na Solaris – služby startovány paralelně podle závislostí, automatické restartování služeb podle potřeby, každá služba má vlastní log

Jak probíhá přihlášení uživatele do systému?

- **init** spouští na každém terminálu proces **getty**, který vypíše výzvu „login:“, přečte jméno a spustí program **login**
- **login** přečte heslo, nastaví nové UID a GIDs a spustí **shell**
- po odhlášení uživatele **init** spustí znovu **getty**

Jak se používá démon cron?

Cron slouží k provádění periodických operací. Jejich nastavení se nachází v souboru `crontab`, který má každý uživatel svůj. Obvykle slouží k mazání nepotřebných souborů, rotaci logů a zálohování. Každý řádek konfiguračního souboru má tvar:

```
min hod den_v_měsíci m den_v_týdnu uživatel příkaz
```

Jak funguje síťový server inetd?

Démon **inetd** slouží ke spouštění síťových serverů (démonů), které se nepoužívají příliš často a je zbytečné, aby čekaly na příchozí spojení. **inetd** sleduje příchozí spojení na zadaných portech a protokolech, příchozí klienty předává příslušným procesům, které spustí. Těmto procesům nastaví standardní vstupy a výstupy do soketu, a tak program ani nemusí vědět, že komunikuje po síti.

Alternativou je, že **inetd** předá procesu soket pro příchozí spojení, proces musí provést jednou `accept()`. Po ukončení procesu pokračuje **inetd** v přijímání klientů.

Jaké úkoly má správce Unixu?

- instalace a konfigurace operačního systému a aplikací
- správa uživatelských účtů a přidělování diskového prostoru uživatelům (quota)
- údržba systému souborů, kontrola konzistence svazků, zálohování
- sledování zatížení systému
- sledování provozních hlášení systému (logy)
- konfigurace periodicky prováděných akcí
- správa síťových služeb
- zajištění bezpečnosti systému – detekce narušení, sledování novinek, patche, správné nastavení práv a SUID programů, vypnutí nepotřebných démonů, filtrace síťového provozu