Studijní osnova pro předmět:

Implementace v operačních systémech

Ing. Petr Olivka, Ph.D., Mgr. Ing. Michal Krumnikl, Ph.D.

Katedra informatiky VŠB-TU Ostrava

email: petr.olivka@vsb.cz, michal.krumnikl@vsb.cz
http://ivos.mrl.cz
http://poli.cs.vsb.cz

© 2025

Tento text není studijním materiálem! Obsahem textu je výběr důležitých témat a funkcí z doporučené studijní literatury.

1 Procesy, vlákna, signály

1.1 Procesy

Jediným možným způsobem, jak v unixových systémech vytvořit nový proces, je prostřednictvím funkce `fork`. Nové procesy označujeme jako potomky, proces který vytváření nových procesů vyvolal označujeme jako rodiče.

```
pid_t fork(void);
```

V systému je každý proces identifikován jedinečným číslem PID. Vlastní PID a PID rodiče je možno zjistit pomocí fukcí `getpid` a `getppid`.

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

Ukončení procesu je možné pomocí funkce `exit` nebo `_exit`.

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

Funkce `exit` je implementována v knihovně C a zajistí volání funkcí registrovaných pomocí funkcí `atexit` a `on_exit`. Funkce `_exit` je přímo systémové volání.

Rodičovský proces může na své potomky čekat s použitím funkcí `wait` a `waitpid`.

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Povinností rodičovských procesů je přebírat návratový status všech potomků, jinak tyto procesy zůstávají v systému jako tzv. zombie procesy. Pro přebírání statusů je výhodnější funkce `waitpid`, kterou lze použít i v režimu bez zablokování - volba `options` WNOHANG. POZOR! Nevhodná implementace čekání bez zablokování může vést k nežádoucímu aktivnímu čekání (busy waiting)!

Pro výměnu programu v procesu se používají funkce `exec...`.

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Při předávání parametrů je potřeba nezapomenout na povinnost předání názvu spouštěného programu v argumentu 0!

1.2 Vlákna

Pro vlákna je k dispozici podobná sada funkcí, jako pro procesy. Vlákna můžeme vytvářet pomocí funkce `pthread_create`.

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

Pro samotné vytvoření vlákna musí být připravena funkce s odpovídající hlavičkou. Jediný parametr `void*` se funkci předává až při spuštění prostřednictvím funkce `pthread_create` jako poslední parametr `*arg`. Identifikace vlákna je vrácena v parametru `*thread`.

Každé vlákno může zjistit svou identifikaci pomocí funkce `pthread_self`

```
pthread_t  pthread_self(void);
```

Ukončení vlákna je možné dvěma způsoby. Prvním způsobem je regulérní ukončení funkce, která představuje vlákno. Druhám způsobem je pomocí volání funkce `pthread_exit`.

```
void pthread_exit(void *retval);
```

Pokud je potřeba v programu počkat na ukončení vlákna, lze to provést pomocí funkce `pthread_join`. Funkce také převezme návratovou hodnotu vlákna.

```
int pthread_join(pthread_t thread, void **retval);
```

Při ukončení vlákna je potřeba správně převzít návratový kód pomocí ukazatele `void**`.

1.3 Signály

Mezi procesy lze v systému zasílat signály. Z příkazového řádku pro tyto účely slouží program `kill` a v programech je k dispozici funkce stejného jména `kill`.

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

Seznam signálů lze získat v dokumentaci, nebo i pomocí programu `kill -l`. Většinu signálů mohou procesy maskovat, několik signálů je ale nemaskovatelných.

Procesy mohou na signály reagovat, pokud si pro ně programátor napíše obslužnou funkci. Tu je potřeba pro daný proces zaregistrovat pomocí funkce `sigaction`.

```
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

Adresa funkce se vkládá do struktury `sigaction`.

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

Do této struktury se vkládá adresa do `sa_handler`. Dále je možno definovat masku signálů `sa_mask` pomocí funkcí `sigemptyset`, `sigaddset` a `sigdelset`. Upravit chování obsluhy signálu pomocí `sa_flags`. Například pomocí SA_RESTART je možné nepřerušit systémová volání při obsluze signálu.

2 Vstup/výstup

2.1 fprintf, fscanf & spol

Funkce `printf, fprintf, scanf, fscanf` a spol sice nepatří mezi služby operačního systému a jsou implementovány v knihovně jazyka C, přesto by nebylo dobré je opomíjet. Funkcionalita poskytovaná touto sadou funkcí je natolik užitečná, že by bylo škoda ji nevyužívat. Je proto důležité, aby programátor znal, jak jsou tyto funkce napojeny na OS.

Funkce využívají strukturu `FILE` a otevřené soubory jsou reprezentovány jako ukazatele na tuto strukturu. Tato struktura tedy tvoří mezičlánek mezi OS a požadovanou funkcionalitou. Struktura v prvé řadě slouží jako vyrovnávací buffer. Tento buffer lze nastavovat pomocí funkcí `setbuf, setbuffer a setlinebuf`.

```
void setbuf(FILE *stream, char *buf);
void setbuffer(FILE *stream, char *buf, size_t size);
void setlinebuf(FILE *stream);
```

Chování vyrovnávacího bufferu lze podle potřeby nastavit nebo zcela vyřadit pomocí funkce `setvbuf`.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Parametr `mode` může být jedna ze tří hodnot: `_IONBF, _IOLBF a _IOFBF`.

Pokud je potřeba provést zápis všech dosud neuložených dat z vyrovnávacího bufferu do souboru, je pro tyto účely určena funkce `fflush`.

```
int fflush(FILE *stream);
```

Otevření souboru není nutné provádět jen pomocí funkce `fopen`. Pokud je v programu již deskriptor otevřeného souboru, zařízení, soketu a pod., je možné tento deskriptor využít a „zabalit“ jej do struktury `FILE` pomocí funkce `fdopen`. Následně lze dle potřeby nastavit vnitřní vyrovnávací buffery.

```
FILE *fdopen(int fd, const char *mode);
```

Někdy je potřeba provést opačný postup, pro již otevřený soubor lze ze struktury `FILE` získat deskriptor souboru pomocí funkce `fileno`.

```
int fileno(FILE *stream);
```

Pro standardní vstup, výstup a chybový výstup jsou v systému vyhrazeny pro každý proces deskriptory 0, 1 a 2. Těmto deskriptorům odpovídají globální proměnné typu `FILE*` - `stdin, stdout a stderr`. Pokud se v procesu provádí přesměrování standardního vstupu či výstupu, je potřeba následně reinicializovat i odpovídající proměnné `stdio a stdout`. K tomuto účelu slouží funkce `freopen`.

```
FILE *freopen(const char *path, const char *mode, FILE *stream);
```

2.2 Soubory a zařízení

Pro programování produkčních aplikací je nezbytné řádně kontrolovat návratové hodnoty všech funkcí pro I/O. Naprostá většina funkcí v případě chyby vrací jako návratovou hodnotu `-1`. Chybový kód je pak uložen v globální proměnné `errno`. Odpovídající popisný text chyby pak lze vypsat pomocí funkce `perror` nebo text získat pro další zpracování pomocí funkce `strerror`.

```
int errno;
void perror(const char *s);
char *strerror(int errnum);
```

Ne ve všech případech však musí návratová hodnota `-1` znamenat skutečnou chybu. Může jít jen o informaci o určitém stavu I/O operace. O to důležitější je pak správně zpracovávat hodnoty `errno`.

Pro otevření souboru nebo zařízení slouží funkce `open`. Je-li návratová hodnota nezáporné číslo, jde o souborový deskriptor pro další práci se souborem.

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Jako parametr `pathname` se zadává absolutní nebo relativní cesta k souboru. Dále je potřeba správně nastavit `flags`. Zde je nutno potřebnou hodnotu vytvořit jako kombinaci konstant O_xx. Hlavní je určení způsobu komunikace pomocí `O_RDONLY, O_WRONLY, O_RDWR`. Kromě těchto třech základních konstant je dále k dispozici mnoho dalších voleb: `O_CREAT, O_APPEND, O_TRUNC, O_NONBLOCK, O_ASYNC`. Správná kombinace je vždy závislá na způsobu práce se souborem nebo zařízením.

Při vytváření souboru se nesmí zapomínat zadávat ve funkci `open` i poslední třetí parametr, který určuje počáteční nastavení přístupových práv k souboru.

Pro čtení a zápis z a do souboru slouží funkce `read` a `write`. U funkce `read` je potřeba dbát na správné zpracování návratové hodnoty, která může být kladná, záporná, ale i 0. Hodnota 0 znamená konec souboru.

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

Pro uzavření otevřeného souboru nebo zařízení slouží funkce `close`.

```
int close(int fd);
```

Pro pohyb v souboru slouží funkce `lseek`. Touto funkcí lze také snadno zjistit velikost souboru nastavením `offset` na 0 od konce souboru.

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

Pro získání informací o souboru slouží funkce `stat`.

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

Ovládání zařízení se provádí funkcí `ioctl`. Seznam jednotlivých příkazů pro různé typy zařízení je pak možno nalézt pomocí `man ioclt_list`. Podrobnější význam jednotlivých příkazů je pak potřeba dohledat v dokumentaci pro jednotlivá zařízení.

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

Další funkcí pro manipulaci se souborovými deskriptory je `fcntl`.

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

Tato funkce slouží zejména pro nastavení příznaků deskriptorů `F_GETFD/F_SETFD`, nastavení statusu deskriptorů `F_GETFL/F_SETFL`, kdy je možno měnit O_ASYNC a O_NONBLOCK otevřených souborů a pak je možno nastavovat zámky souboru.

2.3 Neblokující komunikace

Tento způsob komunikace se aktivuje nastavením statusu `O_NONBLOCK` při otevření souboru, nebo později je možno tento status měnit pomocí funkce `fcntl`. Funkce `read` se při čtení nezablokuje, pokud nejsou k dispozici data. Podobně funkce `write` nebude zablokována, pokud není možno data zapsat. Stav, kdy se funkce `read` a `write` neprovedla, ale nezablokovala, se indikuje návratovou hodnotou `-1` a hodnotou `errno=EAGAIN`.

Neblokující komunikaci je potřeba vždy důkladně promyslet. Chyba v návrhu komunikace může snadno způsobit „busy waiting“!

2.4 Funkce select

Funkce `select` je vhodnější variantou pro vstup a výstup bez zablokování. Dále je vhodná v případech, kdy je potřeba komunikovat s několika souborovými deskriptory současně.

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

Funkce umožňuje sledovat několik souborových deskriptorů zvlášť pro čtení a zvlášť pro zápis. Deskriptory se vkládají do množiny `fdset`. Parametr `nfds` je pak nejvyšší hodnota deskriptoru v množinách `+1`.

Pokud bude parametr `timeout=NULL`, čeká funkce až do okamžiku, kdy bude možno z některého souborového deskriptoru číst, nebo do něj zapisovat. V případě, kdy je `timeout` uveden, pak funkce čeká maximálně zadaný čas. Je-li doba čekání kratší, pak `timeout` obsahuje zbylý čas. Pokud funkce vyčerpala celý `timeout`, je návratová hodnota 0.

Funkce ponechá v množinách jen ty deskriptory, které je možno použít pro čtení nebo zápis.

Pro práci s množinami slouží několik maker:

```
void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

2.5 Funkce poll

Další spolehlivá možnost, jak se vyhnout zablokování při práci se soubory nebo zařízeními je použití funkce `poll`. Její princip je obdobný, jako u funkce `select`.

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Funkce však pro zadání seznamu sledovaných souborových deskriptorů nevyužívá množiny, ale pole struktur `pollfd`. Hodnota `timeout` se zadává v milisekundách.

```
struct pollfd {
   int   fd;         /* file descriptor */
   short events;     /* requested events */
   short revents;    /* returned events */
};
```

Pro každý sledovaný deskriptor tak musí být v poli právě jedna položka. Zda je sledována možnost čtení nebo zápisu se uvádí v položce `events` pomocí konstant `POLLIN` a `POLLOUT`.

2.6 Paměťově mapované soubory

Další technikou pro práci se soubory je možnost soubor namapovat do adresního prostoru procesu.

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Jako parametr `*addr` se používá hodnota `NULL`, což znamená, že OS namapuje soubor na adresu dle své potřeby. Parametr `length` udává počet bajtů, které se ze souboru do paměti namapují. Pro parametr `prot` se dle způsobu otevření souboru volí kombinace `PROT_READ` a `PROT_WRITE`. Parametrem `flags` je možno určit, zda bude mapování jen pro daný proces `MAP_PRIVATE`, nebo bude sdílené mezi dalšími procesy - `MAP_SHARED`. Parametr `fd` je souborový deskriptor otevřeného souboru. Poslední parametr `offset` určuje začátek mapování souboru do paměti, obvykle hodnota `0`.

Ukazatel vrácený funkcí `mmap` je pak přímo ukazatelem na data souboru.

Při zápisu do paměti se synchronizace dat s obsahem souboru neprovádí okamžitě. Pokud je potřeba data synchronizovat, je možno použít funkci `msync`. Synchronizace dat se provede i po volání funkce `munmap`.

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

2.7 Asynchronní vstup/výstup signálem SIGIO

Kromě možností `O_NONBLOCK`, `select` a `poll` je možno informaci o dostupnosti dat z daného souborového deskriptoru získat i pomocí signálu. Pro nastavení tohoto chování je potřeba provést 3 základní kroky:

1. Definovat funkci pro zachycení signálu `SIGIO` pomocí funkce `sigaction`.
2. Pomocí funkce `fcntl` a volby `F_SETFL` nastavit určenému souborovému deskriptoru příznak `O_ASYNC`.
3. Pomocí funkce `fcntl` a volby `F_SETSIG` přiřadit určenému souborovému deskriptoru vybraný signál.
4. Pomocí funkce `fcntl` a `F_SETOWN` je potřeba nastavit PID daného procesu jako příjemce signálu `SIGIO`.

Tato metoda je vhodná v případech, kdy z daného zařízení vstupuje nepříliš velké množství dat a navíc nepravidelně. Je však potřeba mít na paměti, že signál je generován v situaci, kdy jsou dostupná data z daného deskriptoru a také když je možno data odeslat!

Použití tohoto způsobu asynchronní komunikace je zejména vhodné u souborových deskriptorů, které jsou určeny pouze pro čtení. Možnost zápisu dat do zařízení daného souborového deskriptorem generuje obvykle nadměrné a nežádoucí množství signálů a způsobuje zbytečnou zátěž systému.

V uvedené základní variantě však není možno rozlišit v případě signálu `SIGIO` mezi více souborovými deskriptory, kde je zdroj dat. Aby bylo možno rozlišit mezi více zdroji data, je potřeba nahradit základní funkci pro zpracování signálu z verze s jedním parametrem `int` na variantu se třemi parametry. Viz položka struktury `sigaction`:

```
...
void (*sa_sigaction)(int, siginfo_t *, void *);
...
```

Ve struktuře `siginfo_t` je pak mnoho položek upřesňujících zdroj signálu. Zejména položka `si_fd` určuje souborový deskriptor, který je zdrojem signálu `SIGIO`.

Ve funkci `sigaction` je pro předání parametru `siginfo_t` ještě nutno do struktury `sigaction` uvést do položky `sa_flags` volbu `SA_SIGINFO`.

Samotné rozšíření zpracování signálu o strukturu `siginfo_t` však nestačí. Je potřeba ještě pomocí funkce `fcntl` nastavit `F_SETSIG`, nejčastěji na hodnotu `SIGIO`. Bez tohoto nastavení nebude struktura `siginfo_t` řádně nastavena.

2.8 Asynchronní komunikace aio_*

Nejmodernějším způsobem asynchronní komunikace je používání funkcí `aio_*` (aio_read, aio_write, ...). Chování těchto funkcí je řízeno obsahem struktury `aiocb`:

```
struct aiocb {
    int             aio_fildes;     /* File descriptor */
    off_t           aio_offset;     /* File offset */
    volatile void  *aio_buf;        /* Location of buffer */
    size_t          aio_nbytes;     /* Length of transfer */
    int             aio_reqprio;    /* Request priority */
    struct sigevent aio_sigevent;   /* Notification method */
    int             aio_lio_opcode; /* Operation to be performed; */
};
```

Většina položek struktury má zřejmý význam: souborový deskriptor, offset (obvykle 0), buffer pro data nebo s daty pro uložení, délka dat a priorita (obvykle 0). Popis události, co se má provést při načtení nebo zápisu dat je v položce `sigevent`.

```
struct sigevent {
    int          sigev_notify; /* Notification method */
    int          sigev_signo;  /* Notification signal */
    union sigval sigev_value;  /* Data passed with
                                  notification */
    void       (*sigev_notify_function) (union sigval);
                    /* Function used for thread
                       notification (SIGEV_THREAD) */
    void        *sigev_notify_attributes;
                    /* Attributes for notification thread
                       (SIGEV_THREAD) */
    pid_t        sigev_notify_thread_id;
                    /* ID of thread to signal (SIGEV_THREAD_ID) */
};
```

V této struktuře je možno uvést v položce `sigev_notify`, že se po provedení požadované vstupně/výstupní operace nestane nic - `SIGEV_NONE`. Nebo bude vygenerován signál - `SIGEV_SIGNAL`. Případně může být o události informace předána vláknu - `SIGEV_THREAD`.

Požadavky na provedení vstupní a výstupní operace se zadávají pomocí funkcí `aio_read` a `aio_write`. Průběh zadaných asynchronních operací je možno sledovat pomocí funkce `aio_error` a výsledek operace je možno získat pomocí `aio_return`.

3 Meziprocesní komunikace

3.1 Anonymní roury

Základním a nejčastěji používaným způsobem meziprocesní komunikace je roura. Vytváří se pomocí funkce `pipe`.

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

Roura je jednosměrný komunikační prostředek daný dvěma souborovými deskriptory, kdy deskriptor 0 slouží jako výstupní a deskriptor 1 jako vstupní. Číslování odpovídá stdin a stdout.

Protože roury vytváříme v procesu, není možné tuto rouru předat jinému procesu. Rouru může převzít pouze proces vytvořený pomocí `fork`. V procesu může být roura využívána bez jakéhokoliv omezení, včetně komunikace mezi vlákny.

3.2 Pojmenované roury

Anonymní roury mají omezenou možnost využití v procesu a mezi procesy, které si proces sám vytvoří, tedy rodičovský proces a jeho potomci. Toto omezení neplatí pro pojmenované roury.

Pojmenované roury se vytváří přímo v souborovém systému a to pomocí příkazu nebo funkce `mknod` nebo `mkfifo`.

```
mknod [OPTION]... NAME TYPE [MAJOR MINOR]
mkfifo [OPTION]... NAME...
```

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

Příkazem "`mknod naseroura p`" nebo "`mkfifo naseroura`" se vytvoří pojmenovaná roura `naseroura`, kterou lze v programu využívat pomocí otevření souboru funkcí `open`. Při otevírání roury je vždy třeba správně určit, který z obou konců roury má být použit. Pro otevření konce pro zápis je potřeba zadat při otevírání parametr `O_WRONLY` a odpovídajícím způsobem je potřeba otevřít i konec pro čtení jako soubor `O_RDONLY`. Je možno otevřít rouru i jako `O_RDWR`, to však není nejvhodnější řešení.

Práce s pojmenovanou rourou však přináší několik nástrah. První problém může být samotné otevření roury pro čtení. Pokud v dané chvíli již není otevřen konec pro zápis, funkce `open` se zablokuje. Tomu lze zabránit přidáním `O_NONBLOCK` při otevření souboru. Práce se neblokovaným souborovým deskriptorem však vyžaduje jistou opatrnost, aby se program nedostal do stavu busy waiting. Dále pak použitím neblokovaného deskriptoru nelze detekovat uzavření roury. Funkce `read` bude vždy detekovat chybu `EAGAIN` a nebude vracet návratovou hodnotu 0, obvykle detekující konec souboru.

Problému s neblokujícím deskriptorem se lze částečně zbavit tím, že po otevření roury pro čtení následně funkcí `fcntl` příznak `O_NONBLOCK` odstraníme. Tím se stane deskriptor blokujícím a při čtení z uzavřené roury bude funkce `read` vracet hodnotu 0. To však při čekání na připojení procesu, který bude data posílat, může opět způsobit busy waiting. V této situaci si však lze snadno poradit s pomocí funkce `select` nebo `poll`.

3.3 Sdílená paměť - mmap a POSIX standard

Jednou z možností využití sdílené paměti je funkce `mmap`, která již byla zmíněna pro mapování souborů do paměti. Funkci `mmap` však lze využít i pro mapování paměti bez souboru. Stačí při volání funkce `mmap` uvést volbu `MAP_ANONYMOUS` a jako souborový deskriptor uvést `-1`. Takto mapovaná paměť může být sdílena mezi procesy. Problém je pouze v tom, že paměť je anonymní a proces, který ji vytvořil ji může předat pouze svým potomkům.

Pokud je potřeba předávat data mezi procesy zcela nezávislými, je jedním z možných způsobů použití sdílené paměti dle standardu POSIX. Tento způsob používání sdílené paměti je pak složen z několika kroků:

1. Vytvoření sdílené paměti pomocí funkce `shm_open`. Tato funkce vytvoří soubor určeného jména v paměťově mapovaném souborovém systému a výsledkem funkce je souborový deskriptor.

2. Pomocí funkce `ftruncate` je potřeba nastavit souboru velikost odpovídající požadované velikosti sdílené paměti.

3. Pomocí funkce `mmap` namapovat soubor do paměti.

```
int shm_open(const char *name, int oflag, mode_t mode);
int ftruncate(int fd, off_t length);
```

Vytvořená paměť je dostupná pomocí známého jména souboru všem procesům. Jméno by mělo začínat znakem '/'. Přístup je samozřejmě chráněn pomocí standardních přístupových práv. Přesná pozice souboru v souborovém systému však není pevně stanovena a může se lišit v každé unixové distribuci. Proto se jméno sdílené paměti uvádí bez adresářů.

3.4 Sdílená paměť - System V

Další způsob pro práci se sdílenou pamětí nabízí funkce dle staršího standardu System V. Pro práci se sdílenou pamětí slouží 4 funkce:

```
int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Funkce `shmget` provede otevření, případně i vytvoření sdílené paměti požadované velikosti. Paměť se identifikuje svým jedinečným číselným klíčem. Velikost jednou vytvořené paměti již nelze měnit, požaduje-li proces změnu velikosti paměti, musí paměť uvolnit a vytvořit znovu. To však může být problém v případě, že paměť používají další procesy.

Výsledkem funkce `shmget` je `id` požadované paměti. Získané `id` se následně využije pro připojení získané paměti do adresního prostoru procesu pomocí funkce `shmat`. Přístup do paměti se pak realizuje přes adresu získanou touto funkcí.

Pro odpojení paměti od procesu se používá funkce `shmdt`. Zrušení alokované paměti lze provést přímo v procesu pomocí funkce `shmctl`.

3.4.1 Správa meziprocesních prostředků System V

Pro sledování prostředků meziprocesní komunikace lze v systému využít program `ipcs`, který vypíše všechny alokované prostředky - sdílené paměti, fronty zpráv a semafory. Prostředky které procesy vytvořily a neuvolnily lze zrušit programem `ipcrm`.

3.5 Semafory - POSIX

Semafory lze považovat za nejjednodušší a nejdůležitější synchronizační nástroj mezi procesy a vlákny.

Semafory dle POSIX lze rozdělit na semafory pojmenované a anonymní. Pojmenované semafory se vytváří pomocí funkce `sem_open`.

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

Parametr `name` specifikuje jednoznačné jméno právě jednoho semaforu v systému. Dle doporučení má jméno začínat znakem '/'. Semafor je vytvořen systémem mimo proces a do procesu mapován na adresu, která je návratovou hodnotou funkce `sem_open`. Druhým parametrem `oflag` se určí přístup k semaforu pro čtení, zápis a případně vytvoření semaforu `O_CREAT`. Třetí a čtvrtý parametr je potřeba uvádět pouze při vytváření semaforu. Parametry určují přístupová práva a počáteční hodnotu semaforu.

Semafory získané pomocí funkce `sem_open` je možno využívat mezi procesy i mezi vlákny v procesu.

Pro ovládání semaforů slouží funkce `sem_post` a `sem_wait`.

```
int sem_post(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

Funkce `sem_post` zvyšuje vnitřní hodnotu semaforu o jedna. Funkce `sem_wait` snižuje hodnotu semaforu o jedna. Je-li např. semafor používán pro kritickou sekci, je uvedená dvojice funkcí používána tak, že před vstupem do kritické sekce se hodnota semaforu snižuje o jedna na hodnotu nula. Za kritickou sekcí se pak hodnota semaforu zvyšuje z nuly na jedna, čímž se umožní opětovný vstup do kritické sekce dalšímu procesu, nebo vláknu.

Funkci `sem_wait` je možno použít ve třech různých variantách. Varianta `sem_trywait` se nezablokuje, pokud je právě semafor na hodnotě nula. Varianta `sem_timedwait` se v případě, kdy semafor není možno snížit o jedna, zablokuje jen do doby určené časovým omezením. Časový limit je dán absolutním časem, do kdy se může funkce zablokovat. Časový údaj tedy neudává dobu čekání!

Anonymní semafory nejsou v systému jednoznačně pojmenovány a vytváří se v programu jako běžné proměnné. Než si ale programátor vytvoří anonymní semafor, měl by si analyzovat, jak bude semafor používat. Jsou zde dvě zásadně odlišné možnosti, dle rozsahu platnosti (činnosti) semaforu.

Pokud bude semafor využíván jen v rámci jednoho procesu, např. pro synchronizaci vláken, je dostačující deklarovat semafor jako lokální nebo globální proměnnou v programu.

Bude-li potřeba semafor využívat i mezi procesy, např. pro synchronizaci procesu rodičovského s procesy jeho potomků, pak je nezbytně nutné vytvořit semafor v paměti, která bude všem procesům dostupná.

Bez ohledu na místo v paměti, kde byl semafor vytvořen, je potřeba jej na počátku inicializovat funkcí `sem_init`.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Při inicializaci je potřeba zejména správně nastavit druhý parametr, který určuje, zda bude semafor využíván pouze v jediném procesu, nebo zda bude využíván ve více procesech. Umístění semaforu ve sdílené paměti samo o sobě neurčuje, zda bude možno semafor používat mezi více procesy.

Poslední parametr funkce `sem_init` určuje počáteční hodnotu semaforu. Pro inicializovaný anonymní semafor se pak dále používají stejné funkce, jako pro semafor pojmenovaný.

Anonymní semafor umístěný ve sdílené paměti nemusí být až tak docela anonymní. Stačí pro jeho umístění použít pojmenovanou sdílenou paměť. Tím se stane i anonymní semafor zcela rovnocenný s pojmenovaným semaforem.

3.6 Semafory - System V

Semafory dle staršího standardu System V se vytváří skupinově a v systému jsou globálně identifikovány číselným klíče, podobně jako sdílená paměť dle System V. Vytvoření semaforů se provádí pomocí funkce `semget`.

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

Klíč musí být v daném systému jedinečný. Druhý parametr určuje, kolik semaforů se vytvoří. Jednotlivé semafory se následně číslují jako prvky pole, tedy od `0` do `nsems-1`. Poslední parametr funkce určuje přístupová práva k semaforům, opět jako 3 oktalové číslice. Pokud je potřeba semafory poprvé vytvořit, je potřeba přidat volbu `IPC_CREAT`.

Pro ovládání semaforu je určena funkce `semop`.

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

Prvním parametrem je identifikační číslo skupiny semaforů získané pomocí `semget`. Jako druhý argument funkce `semop` se předává pole struktur popisujících operace, které se voláním funkce provedou. Poslední parametr udává počet struktur v poli.

Samotná operace pro zvýšení a snížení hodnoty semaforu se popisuje pomocí struktury `sembuf`.

```
struct sembuf
{
  unsigned short sem_num;  /* semaphore number */
  short          sem_op;   /* semaphore operation */
  short          sem_flg;  /* operation flags */
};
```

První položka struktury `sem_num` je pořadové číslo semaforu, se kterým se bude operace provádět. Druhá položka `sem_op` je hodnota, o kterou se mám hodnota semaforu změnit. Nejčastěji hodnota `1` nebo `-1`. Poslední položka struktury `sem_flg` se nastavuje na `0`. Je ale možno nastavit např. `IPC_NOWAIT`, aby se provedení operace nezablokovalo.

Pro nastavení počáteční hodnoty semaforu, zjištění aktuálního stavu, zrušení semaforu a pod., slouží funkce `semctl`.

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

Seznam příkazů je popsán v dokumentaci. Pro nastavení počáteční hodnoty semaforu slouží příkaz `SETVAL`.

3.7 Fronta zpráv - POSIX

Frontu zpráv lze považovat za nejuniverzálnější způsob komunikace. Frontu zpráv lze využít nejen pro předávání informací mezi procesy, kdy ve frontě zůstávají jednotlivé zprávy od sebe odděleny a nedojte k jejich spojení od jednoho celku, jako např. v rouře. Operace odeslání a příjmu zprávy je z pohledu procesů atomická a frontu zpráv lze bezpečně použít i jako prostředek synchronizační.

Fronta zpráv může být jen pojmenovaná a lze ji vytvořit a následně získat pomocí funkce `mq_open`.

```
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);
```

První parametr funkce určuje jednoznačné jméno fronty v systému. Jméno musí začínat znakem '/'. Druhý parametr udává, zda fronta bude jen pro čtení nebo i pro zápis. V případě, kdy je potřeba frontu vytvořit, je nutno přidat i hodnotu `O_CREAT`. Třetí parametr má význam pouze při vytváření fronty a říká, s jakými přístupovými právy se fronta vytvoří. Poslední parametr může být `NULL` a vytvořená fronta bude mít přednastavené parametry, jako počet zpráv, maximální délku zprávy a pod.

Výsledkem funkce `mq_open` je v Linuxu regulérní souborový deskriptor. Ten je pak možno využívat pomocí ve funkcích `select` a `poll`.

Pro odeslání zprávy slouží funkce `mq_send`. Příjem zprávy zajistí funkce `mq_receive`.

```
int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned msg_prio);
int mq_timedsend(mqd_t mqdes, const char *msg_ptr,
                 size_t msg_len, unsigned msg_prio,
                 const struct timespec *abs_timeout);

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned *msg_prio);
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr,
                        size_t msg_len, unsigned *msg_prio,
                        const struct timespec *abs_timeout);
```

Význam parametrů funkcí je zřejmý. Je nutno dodržet maximální délku zprávy při odesílání a při příjmu zprávy je potřeba připravit buffer odpovídající maximální délce zprávy pro danou frontu. Pokud nebyla nastavena maximální délka zprávy při vytváření fronty zpráv, je nutno tuto hodnotu v programu získat pomocí funkce `mq_getattr`.

```
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);

struct mq_attr {
  long mq_flags;       /* Flags: 0 or O_NONBLOCK */
  long mq_maxmsg;      /* Max. # of messages on queue */
  long mq_msgsize;     /* Max. message size (bytes) */
  long mq_curmsgs;     /* # of messages currently in queue */
  };
```

Při odesílání zpráv funkcí `mq_send` je možno také zadat prioritu zprávy. Zprávy s vyšší prioritou ve frontě "předběhnou" zprávy s prioritou nižší.

3.8 Fronta zpráv - System V

Frontu dle standardu System V je možno vytvořit a následně získat pomocí funkce `msgget`.

```
int msgget(key_t key, int msgflg);
```

Fronta se v systému jednoznačně identifikuje svým jedinečným klíčem, který se zadává jako první parametru funkce. Druhým parametrem jsou přístupová práva a dle potřeby i požadavek na vytvoření fronty `IPC_CREAT`.

Pro odesílání zpráv slouží funkce `msgsnd`, pro příjem zpráv `msgrcv`.

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
               int msgflg);
```

Pro předávání zpráv dle System V je v prvé řadě nutno dodržet formát zprávy. Ten je definován pomyslnou strukturou `msgbuf`.

```
struct msgbuf {
    long mtype;       /* message type, must be > 0 */
    char mtext[LEN];  /* message data, LEN must be at least 1 */
};
```

Každá zpráva musí začínat položkou `mtype`, která udává typ zprávy. Ten si určuje každý programátor sám dle vlastní potřeby. Typ zprávy musím být číslo kladné, větší než 0 (typ zprávy neovlivňuje pořadí zpráv ve frontě). Dále musí mít každá zpráva nenulovou délku, která je ve struktuře `msgbuf` uvedena jako `LEN`.

Pozor! Typ zprávy v její hlavičce se do délky zprávy nepočítá!

Při příjmu zprávy se ve funkci `msgrcv` zadává kromě bufferu pro data a jeho délky i typ očekávané zprávy. Tento typ může být při příjmu zpráv

nastaven na `0`, což znamená, že bude přijata první zpráva, která je ve frontě, bez ohledu na typ ve své hlavičce. Pokud je požadovaný typ zprávy nenulový, bude přijata první zpráva požadovaného typu, nebo se funkce zablokuje a bude čekat na požadovanou zprávu, dokud nebude požadovaná zpráva do fronty vložena.

Pomocí parametrů `msgflg` je možno se vyhnout zablokování volbou `IPC_NOWAIT`.

Pokud nebyl pro přijímanou zprávu připraven dostatečně velký buffer, nepřevezme se z fronty zpráva celá. Zbytek nepřijaté zprávy bude zahozen.

Pro zjištění stavu fronty, nastavení některých parametrů, zrušení fronty a pod., slouží funkce `msgctl`.

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Popis potřebných příkazů a struktur je v dokumentaci funkce.

4 Sokety

Problematika komunikačního síťového rozhraní `socket` je velmi rozsáhlá. V rámci výuky bude rozsah celé problematiky zúžen pouze na potvrzovaný protokol TCP. Nepotvrzovaná komunikace protokolem UDP nebude probírána.

Kromě studijního materiálu je možno (i nutno) řadu informací získat přímo z manuálových stránek:

[m7sock]	man 7 socket
[m7unix]	man 7 unix
[m7tcp]	man 7 tcp
[m7ip]	man 7 ip
[m7ipv6]	man 7 ipv6
[msock]	man socket
[mbind]	man bind
[mlst]	man listen
[macc]	man accept
[mcon]	man connect
[mopt]	man setsockopt, getsockopt
[mhn]	man htons, ntohs, htonl, ntohl
[minet]	man inet
[mnp]	man inet_ntop, inet_pton
[madr]	man getaddrinfo
[mnam]	man getpeername, getsockname

4.1 Postup navazování spojení

Při navazování spojení a následné komunikaci je nutno použít na straně serveru a klienta funkce v následujícím pořadí:

Server	Klient
socket	socket
bind	? bind
listen	
accept	connect
read/write	read/write
close	close

Pro vytvoření soketového spojení musí být nejprve soket vytvořen na straně serveru a nabídnut k připojení klienta.

Vytvoření soketu se provádí pomocí funkce `socket`:

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

Parametrem `domain` určujeme rodinu protokolů. Téměř vždy to bude hodnota z trojice: `AF_UNIX` - lokální soket, `AF_INET` - IPv4 a `AF_INET6` - IPv6.

Druhým parametrem `type` vybíráme z dané rodiny protokolů potřebný typ protokolu. Pro uvedené rodiny protokolů můžeme volit ze tří možností: `SOCK_STREAM` - TCP, `SOCK_DGRAM` - UDP a `SOCK_RAW` - IP.

Pokud by v dané rodině protokolů `domain` byl vybraný typ `type` protokolu implementován více protokoly, bude se konkrétní protokol vybírat parametrem `protocol`. Seznam protokolů lze nalézt v `/etc/protocols`. Pro výše uvedené rodiny a typy protokolů bude `protocol` hodnota 0.

Dalším krokem je na straně serveru nastavení portu (souboru), kterým bude soket nabízen. K tomuto účelu slouží funkce `bind`.

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Adresa se přiřazuje již vytvořenému soketu `sockfd`. Typ parametru `addr` je obecná struktura `sockaddr`. Konkrétní typ struktury je potřeba dle rodiny protokolů najít v [m7unix, m7ip, m7ipv6]. Poslední parametr funkce `addrlen` je délka předávané struktury.

```
#define UNIX_PATH_MAX    108

 struct sockaddr_un {
     sa_family_t  sun_family;               /* AF_UNIX */
     char         sun_path[UNIX_PATH_MAX];  /* pathname */
 };

struct sockaddr_in {
     sa_family_t    sin_family; /* address family: AF_INET */
     in_port_t      sin_port;   /* port in network byte order */
     struct in_addr sin_addr;   /* internet address */
 };

 /* Internet address. */
 struct in_addr {
     uint32_t       s_addr;     /* address in network byte order */
 };

struct sockaddr_in6 {
     sa_family_t     sin6_family;   /* AF_INET6 */
     in_port_t       sin6_port;     /* port number */
     uint32_t        sin6_flowinfo; /* IPv6 flow information */
     struct in6_addr sin6_addr;     /* IPv6 address */
     uint32_t        sin6_scope_id; /* Scope ID (new in 2.4) */
 };

 struct in6_addr {
     unsigned char   s6_addr[16];   /* IPv6 address */
 };
```

V síti se služba nejčastěji nabízí prostřednictvím všech síťových rozhraní počítače. Pro protokol IPv4 je k tomuto účelu definována adresa `INADDR_ANY` a pro IPv6 adresa `in6addr_any`.

Číslo portu je potřeba potřeba do struktury dosadit ve formátu odpovídajícímu síťovému rozhraní. Pro převod formátu čísel mezi počítačem a sítí slouží funkce `ntoh*` a `hton*`, viz [mhn].

Čísla portů do 1024 jsou rezervována a smí je použít pouze správce systému. Uživatelé si mohou volit libovolná čísla vyšší než 1024.

Funkce `bind` není na straně klienta povinná. Používá se pouze v případě, kdy klient chce navazovat spojení z konkrétního portu.

Dalším krokem na straně serveru je zaregistrování soketu v systému pomocí funkce `listen`.

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

Tato funkce zajistí procesu, že jeho soket bude nabízet připojení dalším procesům i počítačům. Parametr `backlog` určuje délku vnitřní fronty čekajících připojení. Dnes je tato hodnota prakticky ignorovaná.

Jednou zaregistrovaná služba s vybraným číslem portu zůstává v systému po ukončení procesu ještě několik minut blokována. Aby bylo možno při novém spuštění procesu serveru nabízet službu okamžitě znovu na stejném čísle portu, je potřeba nastavit soketu vlastnost „znovupoužití“ adresy. To se realizuje funkcí `setsockopt`, viz [mopt].

```
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

Odpovídající hodnota `optname` je `SO_REUSEADDR` a je uvedena v [m7sock].

Po zaregistrování soketu v systému už si server dále jen vybírá navázaná spojení pomocí funkce `accept`.

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Nové soketové spojení s klientem je vráceno jako návratová hodnota funkce. Parametr `sockfd` je dříve zaregistrovaný soket. Pomocí parametru `addr` je možno získat informaci o adrese počítače, který se k soketu připojil. Parametr `addrlen` je délka struktury na vstupu a parametr bude obsahovat skutečnou délku předaných dat na výstupu.

Funkce `accept` se standardně zablokuje, dokud není k dispozici připojený klient. Pokud se zablokování chceme vyhnout, můžeme sledovat stav soketu `sockfd` monitorovat pomocí funkcí `select` a `poll`.

Návratová hodnota funkce `accept` je regulérní souborový deskriptor a slouží dále pro komunikaci s klientem. Pro komunikaci je možno použít všechny způsoby I/O komunikace, které byly dosud uvedeny.

Na straně klienta je postup navazování spojení o trochu jednodušší. Soket je nutno vytvořit pomocí funkce `socket`, stejně jako na serveru. Následné použití funkce `bind` je pro protokoly IPv4 a IPv6 volitelné, viz bind výše. Pro sokety `AF_UNIX` je však nezbytné.

Spojení se serverem se navazuje pomocí funkce `connect`.

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Pro navázání spojení je nutno použít dříve vytvořený soket `sockfd`. Jako parametr `addr` se použije struktura odpovídající zvolenému protokolu, viz funkce `socket` a [m7unix, m7ip, m7ipv6].

Po úspěšném navázání spojení je soket dále používán jako regulérní souborový deskriptor a lze proto využít všechny známé principy I/O komunikace.

Výsledkem funkce `connect` na straně klienta a funkce `accept` na straně serveru je soketové spojení, které je pro protokoly IPv* specifikováno čtveřicí údajů [IP:PORT]-[IP:PORT]. Tato čtveřice údajů musí být v internetu (nebo dané uzavřené síti) VŽDY jedinečná!

4.1.1 Zablokování funkce `connect`

Funkce `connect` se může při navazování spojení zablokovat na dobu, která je dána vnitřním nastavením soketů v systému. Toto chování vnáší do aplikace možné nežádoucí nepredikovatelné zablokování (podobně jako je tomu u pojmenované roury ve funkci `open`).

Tomuto chování lze předejít řízeným způsobem. Pomocí funkce `fcntl` je potřeba nastavit soket jako `O_NONBLOCK` před voláním funkce `connect`. Pokud se nepodaří navázat spojení během velmi krátkého volání funkce `connect`, vrátí funkce výsledek -1 a chybový kód EINPROGRESS (jako je pravidlem u neblokujících operací). Na dokončení navazování spojení je pak možno vyčkat pomocí funkcí `poll` nebo `select`, viz [m7sock]. Dle dokumentace [mcon], chyba EINPROGESS, je u soketu potřeba počkat, až bude připraven pro zápis (writable).

Tímto je ukončen proces navazování spojení a je potřena ověřit, zda k navázání spojení skutečně došlo, nebo se spojení navázat nepodařilo. Dle [mcon], chyba EINPROGREES, je potřeba pomocí funkce `getsockopt` ověřit stav soketu.

```
int getsockopt(int sockfd, int level, int optname,
               void *optval, socklen_t *optlen);
```

Pro zjištění stavu soketu slouží parametr `optname` nastavený na `SO_ERROR`, viz [m7sock]. Korektně navázané spojení je indikováno hodnotou 0. Nenulový výsledek je některý ze standardních chybových kódů.

4.1.2 Překlad textové IP adresy a DNS jména do síťového tvaru

Pro překlad IPv4 adresy v textové podobě do síťového tvaru a zpět slouží funkce `inet_aton` a `inet_ntoa`, viz [minet].

Univerzálnější je použití funkcí `inet_pton` a `inet_ntop`, viz [mnp].

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

Tyto funkce lze použít pro sokety rodiny `AF_INET` i `AF_INET6`.

Mnohem používanější, než překlad textově zadané IP adresy, je překlad DNS jména na síťovou adresu. Pro tento účel se používá POSIX funkce `getaddrinfo`, viz [madr].

```
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);

struct addrinfo {
    int              ai_flags;
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
    socklen_t        ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

První parametr funkce `node` je řetězec obsahující DNS jméno počítače. Druhý parametr `service` je požadovaná služba - viz `/etc/services`. Tento parametr může být `NULL` a výsledná adresa nebude obsahovat číslo portu.

Třetí parametr `hints` lze nastavit také na `NULL`. Chování funkce je pak popsáno v [madr]. Výhodnější je však pro parametr `hints` připravit přednastavenou strukturu `addrinfo`. Význam položek `ai_family, ai_socktype, ai_protocol` je shodný s parametry funkce `socket` a to i ve stejném pořadí. Je proto vhodné omezit překlad DNS jména na vybranou rodinu protokolů nastavením `ai_family` na `AF_INET`, nebo `AF_INET6`. Pokud je potřeba přeložit DNS jméno pro IPv4 i IPv6, lze použít předdefinovanou hodnotu `AF_UNSPEC`. Položku `ai_socktype` je možno nastavit na `SOCK_STREAM`,

SOCK_DGRAM, nebo SOCK_RAW. Všechny ostatní položky struktury je potřeba nastavit na 0 a NULL.

Výsledkem funkce getaddrinfo je zřetězený seznam struktur addrinfo. V každé položce seznamu je jedna nalezené adresa, včetně uvedení příslušné rodiny protokolů a typu soketu. Z toho seznamu je následně potřeba vybrat jednu strukturu, použít ji a celý seznam struktur uvolnit pomocí funkce freeaddrinfo.

4.2 Pojmenované sokety (AF_UNIX)

Pojmenované sokety jsou obdobou pojmenované roury. Kontaktní bod pro procesy je speciální typ souboru v souborovém systému. Tyto sokety slouží výhradně pro komunikaci mezi procesy v systému. Pomocí těchto soketů nelze komunikovat s dalšími počítači v síti.

Následující ukázka kódu serveru očekává připojení klienta. Kód je pouze ilustrační a nejsou v něm ošetřeny návratové kódu funkcí!.

```
int sock = socket( AF_UNIX, SOCK_STREAM, 0 );

sockaddr_un sun;
sun.sun_family = AF_UNIX;
strcpy( sun.sun_path, "/tmp/filename" );
bind( sock, ( sockaddr * ) &sun, sizeof( sun );

listen( sock, 1 );

int newsock = accept( sock, NULL, 0 );

// communication by newsock

close( newsock );
close( sock );
unlink( sun.sun_path );
```

V následující ukázce je kód klienta, který se k serveru připojí.

```
int sock = socket( AF_UNIX, SOCK_STREAM, 0 );

sockaddr_un sun;
sun.sun_family = AF_UNIX;
strcpy( sun.sun_path, "/tmp/filename" );

connect( sock, ( sockaddr * ) &sun, sizeof( sun ) );

// communication by sock

close( sock );
```

Vytvoření spojení pomocí pojmenovaného soketu je trochu složitější, než pomocí pojmenované roury. Samotná komunikace je pak ale obousměrná a odpovídá standardnímu používání souborových deskriptorů.

Na rozdíl od pojmenované roury se však pojmenovaný soket v souborovém systému nevytváří pomocí `mknode`, ale vytvoří se voláním funkce `listen`. Další rozdíl spočívá v nutnosti soubor ze souborovém systému odstranit pomocí `unlink`, před opakovaným spuštěním procesu serveru.

4.3 Vytvoření spojení pomocí AF_INET soketu

Na straně serveru se pro různé rodiny protokolů kód programu prakticky neliší. Rozdíl je jen v použité struktuře ve funkci `bind`. Následující kód je opět je ukázka a není vhodný pro produkční účely!

```
int sock = socket( AF_INET, SOCK_STREAM, 0 );

sockaddr_in sin;
sin.sin_family = AF_INET;
sin.sin_port = htons( port_number );
sin.sin_addr.s_addr = INADDR_ANY;

bind( sock, ( sockaddr * ) &sin, sizeof( sin ) );

int opt = 1;
setsockopt( sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof( opt ) );

listen( sock, 0 );
int newsock = accept( sock, NULL, 0 );

sockaddr_in tmpaddr;
char buf[ 128 ];
socklen_t len = sizeof( tmpaddr );

getpeername( newsock, ( sockaddr * ) &tmpaddr, &len );
printf( "peer: %s:%d\n",
  inet_ntop( AF_INET, &tmpaddr.sin_addr, buf, sizeof( buf ) ),
  ntohs( tmpaddr.sin_port ) );

getsockname( newsock, ( sockaddr * ) &tmpaddr, &len );
printf( "sock: %s:%d\n",
  inet_ntop( AF_INET, &tmpaddr.sin_addr, buf, sizeof( buf ) ),
  ntohs( tmpaddr.sin_port ) );

// communication by newsock

close( newsock );
close( sock );
```

Na straně klienta se kód programu pro `AF_INET` rozšíří proti kódu pro `AF_UNIX` o překlad jména počítače.

```
int sock = socket( AF_INET, SOCK_STREAM, 0 );

addrinfo hint, *retaddr;

bzero( &hint, sizeof( hint ) );
hint.ai_family = AF_INET;
hint.ai_socktype = SOCK_STREAM;

getaddrinfo( host_name, NULL, &hint, &retaddr );

sockaddr_in sin = * ( sockaddr_in * ) retaddr->ai_addr;
sin.sin_port = htons( atoi( port_number ) );

freeaddrinfo( retaddr );

connect( sock, ( sockaddr * ) &sin, sizeof( sin ) );

// communication by sock

close( sock );
```

4.4 Vytvoření nepojmenovaných propojených soketů

Funkce `socketpair()` je užitečným nástrojem pro vytváření obousměrné komunikační linky mezi procesy. Poskytuje dvojici propojených soketů, které mohou být použity pro efektivní meziprocesovou komunikaci. Funkce se často používá pro výměnu dat mezi rodičovským procesem a potomkem.

Funkce `socketpair()` vytváří dva propojené sokety, které umožňují přímou komunikaci mezi procesy. Tyto sokety jsou ekvivalentní anonymním rourám (pipes), ale na rozdíl od rour jsou obousměrné, což umožňuje komunikaci v obou směrech.

```
int sockets[2];
if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) == -1) {
    perror("socketpair");
    exit(EXIT_FAILURE);
}

pid_t pid = fork();
if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
}
```

```
if (pid == 0) { // Child
    close(sockets[0]);
    char msg[] = "Hello, parent!";
    write(sockets[1], msg, sizeof(msg));
    close(sockets[1]);
} else { // Parent
    close(sockets[1]);
    char buffer[100];
    read(sockets[0], buffer, sizeof(buffer));
    printf("Received: %s\n", buffer);
    close(sockets[0]);
}
```

Funkci `socketpair()` můžeme využít, kdy je potřeba efektivní komunikace mezi procesy, například:

- Předávání souborových deskriptorů mezi procesy pomocí `sendmsg()` a struktury `cmsghdr` (viz následující kapitola).
- Implementace IPC mechanismů pro synchronizaci a výměnu dat.
- Vytváření vícevláknových serverů, kde hlavní proces deleguje úlohy podprocesům.

5 Předání souborového deskriptoru mezi procesy

Procesy jsou v OS mezi sebou velmi silně izolovány a to včetně svých souborových deskriptorů. Nelze tedy mezi procesy předat souborový deskriptor, stejně jako nelze předat např. ukazatel. Tyto hodnoty popisují určitou vnitřní informaci v procesu a nemají řádnou informační hodnotu pro další procesy.

Pomocí AF_UNIX soketů je však možno předat souborový deskriptor pomocí speciálního typu zprávy funkcí `sendmsg`. Zpráva je operačním systémem interpretována a předávaný souborový deskriptor je „implantován“ do cílového procesu. Popis potřebné funkce a souvisejících datových struktur je popsán v manuálových stránkách:

man sendmsg
man cmsg

Pro odeslání zprávy slouží funkce `sendmsg` a jako parametr používá strukturu `msghdr`.

```
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);

struct msghdr {
    void         *msg_name;       /* optional address */
    socklen_t     msg_namelen;    /* size of address */
    struct iovec *msg_iov;        /* scatter/gather array */
    size_t        msg_iovlen;     /* # elements in msg_iov */
    void         *msg_control;    /* ancillary data, see below */
    size_t        msg_controllen; /* ancillary data buffer len */
    int           msg_flags;      /* flags (unused) */
};
```

Speciální typ zprávy (ancillary message) se předává ve struktuře `msghdr` v položce `msg_control` jako zřetězený seznam struktur `cmsghdr`.

```
struct cmsghdr {
    size_t cmsg_len;    /* Data byte count, including header
                           (type is socklen_t in POSIX) */
    int    cmsg_level;  /* Originating protocol */
    int    cmsg_type;   /* Protocol-specific type */
/* followed by
   unsigned char cmsg_data[]; */
};
```

Pro manipulaci s jednotlivými zprávami a jejich obsahem slouží sada maker:

```
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *msgh);
struct cmsghdr *CMSG_NXTHDR(struct msghdr *msgh, struct cmsghdr *cmsg);
```

```
size_t CMSG_ALIGN(size_t length);
size_t CMSG_SPACE(size_t length);
size_t CMSG_LEN(size_t length);
unsigned char *CMSG_DATA(struct cmsghdr *cmsg);
```

Příklad použití výše uvedených struktur pro předání souborového deskriptoru je uveden v manuálové stránce `man cmsg`.

6 OpenSSL

OpenSSL je populární open-source knihovna pro kryptografii a správu certifikátů SSL/TLS. Knihovna umožňuje implementaci zabezpečeného přenosu dat v rámci webového serveru prostřednictvím protokolu HTTPS. Zajišťuje šifrování HTTP komunikace pomocí odpovídajících klíčů a zároveň poskytuje mechanismy pro dešifrování přijatých dat.

Instalace

Na Linuxu lze OpenSSL nainstalovat pomocí:

```
sudo apt-get install libssl-dev // Debian
sudo dnf install openssl-devel // Fedora
```

Na macOS:

```
brew install openssl
```

6.1 Základní inicializace OpenSSL

Před použitím OpenSSL je nutné provést jeho inicializaci:

```
#include <openssl/ssl.h>
#include <openssl/err.h>

void initialize_openssl() {
    SSL_library_init();
    SSL_load_error_strings();     // Loads error messages
    OpenSSL_add_ssl_algorithms(); // Initialize cryptographic algorithms
}
// Cleanup at exit
void cleanup_openssl() {
    EVP_cleanup();
}
```

6.2 Vytvoření SSL kontextu

V OpenSSL kontext (`SSL_CTX`) slouží jako hlavní objekt pro správu parametrů SSL/TLS komunikace. Určuje metodu šifrování, uchovává certifikáty a klíče, nastavuje bezpečnostní politiky a umožňuje vytvoření individuálních SSL objektů pro konkrétní spojení. Používá se jak na straně klienta, tak na straně serveru, kde zajišťuje jednotnou konfiguraci pro více připojení. Pro práci s TLS spojeními je nutné vytvořit SSL kontext:

```
SSL_CTX *create_context() {
    const SSL_METHOD *method;
    SSL_CTX *ctx;

    method = TLS_client_method(); // for client (TLS_server_method() for server)
    ctx = SSL_CTX_new(method);
    if (!ctx) {
        perror("Unable to create SSL context");
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
    }
  return ctx;
}
```

6.3 Nastavení certifikátů

Certifikáty a privátní klíče hrají klíčovou roli při zabezpečení SSL/TLS spojení. **Certifikát** slouží k ověření identity serveru (a případně klienta). Obsahuje veřejný klíč a je digitálně podepsán důvěryhodnou certifikační autoritou (CA). Když se klient připojí k serveru, ověřuje, zda je certifikát platný a důvěryhodný.

```
if ( SSL_CTX_use_certificate_file(ctx, "cert.pem", SSL_FILETYPE_PEM) <= 0) {
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
}
```

Privátní klíč odpovídá veřejnému klíči v certifikátu a slouží k dešifrování dat a digitálním podpisům. Musí být bezpečně uložen na serveru, protože jeho únik by umožnil dešifrování komunikace a napodobení serveru.

```
if ( SSL_CTX_use_PrivateKey_file(ctx, "key.pem", SSL_FILETYPE_PEM) <= 0) {
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
}
```

Během TLS handshake server prokáže svou identitu tím, že poskytne svůj certifikát a umožní klientovi ověřit jeho důvěryhodnost. Klient pak může ověřit podpis certifikátu pomocí CA a vytvořit bezpečné šifrované spojení.

6.4 Generování testovacího certifikátu a privátního klíče

Pro vygenerování testovacího páru certifikátu a privátního klíče použijte následující příkazy:

6.4.1 Vytvoření privátního klíče

Následující příkaz vytvoří 2048bitový RSA privátní klíč a uloží ho do souboru:

```
openssl genpkey -algorithm RSA -out privkey.pem -pkeyopt rsa_keygen_bits:2048
```

6.4.2 Vytvoření self-signed certifikátu

Následující příkaz vytvoří s využitím privátního klíče X.509 certifikát platný po dobu 365 dnů:

```
openssl req -new -x509 -key privkey.pem -out cert.pem -days 365
```

Při generování budete požádáni o vyplnění informací, jako je název organizace nebo společnost.

6.4.3 Zobrazení informací o certifikátu

```
openssl x509 -in cert.pem -text -noout
```

6.5 Vytvoření SSL spojení jako klient

```
int sockfd = socket( ... )
SSL *ssl = SSL_new(ctx);
SSL_set_fd(ssl, sockfd);

if (SSL_connect(ssl) <= 0) {
    ERR_print_errors_fp(stderr);
} else {
    printf("Connected␣with␣%s␣encryption\n", SSL_get_cipher(ssl));
}
```

6.6 Odesílání a přijímání dat

```
SSL_write(ssl, "GET␣/␣HTTP/1.1\r\nHost:␣example.com\r\n\r\n", 44);
char response[1024] = {0};
SSL_read(ssl, response, sizeof(response));
printf("Response:␣%s\n", response);
```

6.7 Ukončení SSL spojení

```
SSL_shutdown(ssl);
SSL_free(ssl);
SSL_CTX_free(ctx);
cleanup_openssl();
```

6.8 Použití OpenSSL pro server

Pro vytvoření TLS serveru je nutné:

1. Inicializovat OpenSSL
2. Vytvořit SSL kontext
3. Nastavit certifikáty
4. Otevřít socket a čekat na připojení
5. Přijmout spojení a provést handshake
6. Odesílat a přijímat data

```
SSL_CTX *ctx = create_context();
SSL *ssl;
...
// load certificate and key from file

int server_fd = socket( ... )
struct sockaddr_in addr;
uint len = sizeof(addr);

int client_fd = accept(server_fd, (struct sockaddr*)&addr, &len);
ssl = SSL_new(ctx);
SSL_set_fd(ssl, client_fd);

if (SSL_accept(ssl) <= 0) {
    ERR_print_errors_fp(stderr);
} else {
    printf("Client␣connected␣with␣%s␣encryption\n", SSL_get_cipher(ssl));
}

char buffer[1024] = {0};
SSL_read(ssl, buffer, sizeof(buffer));
printf("Received:␣%s\n", buffer);
SSL_write(ssl, "Hello,␣secure␣world!", 21);

SSL_shutdown(ssl);
```

```
SSL_free(ssl);
close(client_fd);
close(server_fd);
SSL_CTX_free(ctx);
cleanup_openssl();
```

6.9 Dokumentace

- OpenSSL dokumentace: https://www.openssl.org/docs/
- API reference: https://docs.openssl.org/master/man3/