Systémová volání UNIXu

1.1. Nebufferované I/O funkce

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

open – otevření souboru

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

Flagy (zadávat s bitovým OR):

- `O_RDONLY` – otevření pouze pro čtení
- `O_WRONLY` – otevření pouze pro zápis
- `O_RDWR` – otevření pro čtení i zápis
- `O_APPEND` – připojení na konec souboru
- `O_CREAT` – vytvoření nového souboru, vyžaduje nastavit práva pomocí `mode` (viz `creat`)
- `O_EXCL` – otestuje existenci souboru, dohromady s `O_CREAT` vrací chybu
- `O_SYNC` – synchronní I/O operace, každý `write` čeká na fyzické ukončení operace

creat – vytvoření nového souboru, ekvivaletní s `open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);`

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

Nastavení práv:

- Uživatel: `S_IRWXU, S_IRUSR (S_IREAD), S_IWUSR (S_IWRITE), S_IXUSR (S_IEXEC);`
- Skupina: `S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP;`
- Ostatní: `S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH.`

close – zavře otevřený soubor (deskriptor)

```
int close(int filedes);
```

lseek – nastavení pozice pro čtení/zápis uvnitř souboru, pozice (`offset`) je nezáporný pocet bytů od počátku souboru

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

Interpretace `offset` závisi na parametru `whence`:

- `SEEK_SET` – offset je vzdálenost od počátku souboru
- `SEEK_CUR` – offset je relativní vzdálenost od aktualní pozice
- `SEEK_END` – offset je vzdálenost od konce souboru

read – čtení dat z daného deskriptoru, vrací počet načtených bytů

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

write – zapiš do daného souboru, až `count` bytů z bufferu `buf` do file deskriptoru `fd`

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

dup, dup2 – duplikace existujícího deskriptoru, je sdílena pozice, flagy. Volání `dup` vytvoří nejnižší volný deskriptor ve stejné tabulce jako starý, s `dup2` specifikujeme nový hodnotou `newfd` (musí být předtím otevřen)

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

fnctl – změna charakteru otevřeného souboru

```
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

Lze použít pro tyto účely (parametr `cmd`):

- duplikování existujícího deskriptoru (`F_DUPD`)
- zjištění/nastavení příznaku deskriptoru (`F_GETFD, F_SETFD`)
- zjištění/nastavení příznaku souboru (`F_GETFL,F_SETFL`)
- zjištění/nastavení vlastnictví I/O (`F_GETOWN, F_SETOWN`)
- zjištění/nastavení zámku (`F_GETLK, F_SETLK, F_STELKW`)

ioctl – provádění řídících operací nad I/O zařízením

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

1.2. Procesy

exit, _exit – `exit` normálně ukončí proces, `_exit` okamžitě ukončí právě probíhající proces

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

atexit – registruje funkci, která se má vyvolat při ukončení procesu

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

getpid, getppid – vrací identifikátor procesu, tj. unikátní nezáporné číslo, `getpid` vrací ID volajícího, `getppid` ID rodiče volajícího

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

Pro zjištění ID uživatelů slouží `getuid`, `geteuid` (efektivní), pro ID skupiny `getgid`, `getegid`.

fork – vytvoření nového procesu, po zavolání provádí rodič i potomek stejný kód, programově se dají rozlišit např. pomocí návratové hodnoty (u potomka vrací 0, u rodiče PID potomka)

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

exec – nahradí probíhající proces jiným, vrací se do nadřazeného programu jen při chybě, alternativa nejdřív `fork` a v podprocesu `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[]);
```

system – alternativa `exec`, spustí daný *příkaz* pomocí `/bin/sh -c "`*`příkaz`*`"`

```
int system(const char *string);
```

wait, waitpid – čekání na skončení procesu, čeká na signál `SIGCHLD`, `wait` blokuje volajícího, `waitpid` má možnost řízení na koho čekat

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

Parametr `pid`:

- < -1 – čeká na dokončení libovolného potomka,
- -1 – čeká na potomka, jehož ID skupiny je shodné s absolutní hodnotou PID,
- 0 – čeká na potomka, jehož ID skupiny je shodné s ID skupiny volajícího,
- > 0 – čeká na dokončení potomka s daným PID.

1.3. Signály

Signály – softwarová přerušení, po obdržení se spustí obsluha, kompletní seznam viz `man 7 signal`, nebo `kill -l`

Signál	Hodnota	Poznámka
SIGHUP	1	zavěšení na řídícím TTY nebo ukončení řídícího procesu
SIGINT	2	přerušení z klávesnice
SIGQUIT	3	ukončení z klávesnice.
SIGABRT	6	ukončení funkcí `abort`
SIGKILL	9	signál pro nepodmíněné ukončení procesu
SIGSEGV	11	odkaz na nepřípustnou adresu v paměti
SIGPIPE	13	pokus o zápis do roury, kterou nikdo nečte.
SIGALRM	14	signál od časovače, nastaveného funkcí alarm
SIGTERM	15	signál ukončení
SIGUSR1	30,10,16	signál 1 definovaný uživatelem
SIGUSR2	31,12,17	signál 2 definovaný uživatelem
SIGCHLD	20,17,18	zastavení nebo ukončení potomka

sigaction – nastavení obsluhy pro daný signál

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

struct sigaction {
/* Ukazatel na funkci obsluhy */
/* SIG_DFL - implicitní */
/* SIG_IGN - ignorování */
   void (*sa_handler)(int);
/* Blokované signály při obsluze */
   sigset_t sa_mask;
/* Přiznaky ovlivňující obsluhu */
   int sa_flags; }
```

Proměná pro záznam signálu v globální obsluze by měla být typu `sig_atomic_t`.

kill – zaslání signálu procesu

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

Parametr `pid` podobný jako u `wait`.

1.4. Meziprocesová komunikace

pipe – vytvoření datové roury

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

Pro čtení z roury se použije deskriptor `filedes[0]`, pro zápis do ní `filedes[1]`.

popen, pclose – vytvoření jednosměrné roury pro I/O operace, `popen` otevře nový proces pro vytvoření roury, parametr `type` je `r` pro čtění, `w` pro zápis.

```
FILE *popen(const char *command,
            const char *type);
int pclose(FILE *stream);
```

mkfifo – vytvoření pojmenované roury (položky v souborovém systému); normální roury mohou používat pouze procesy se shodným předchůdcem, FIFO mohou používat i procesy, které spolu jinak nesouvisí.

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

Sdílená paměť – sdílení adresního prostoru mezi více procesy, `<sys/ipc.h>`,`<sys/shm.h>`; použití pomocí

- Alokace

  ```
  int   shmget(key_t, size_t, int);
  ```

- Operace

  ```
  void *shmat(int shmid, const void *shmaddr,
              int shmflg);
  ```

- Ovládání

  ```
  int shmctl(int shmid, int cmd,
             struct shmid_ds *buf);
  ```

  - `IPC_STAT` – načtení řídící struktury sdílené paměti
  - `IPC_SET` – nastaveni některych polí struktury `shm_perms`
  - `IPC_RMID` – odstranění sdílené paměti ze systému
  - `SHM_LOCK` – uzamčení sdílené paměti (jen superuživatel)
  - `SHM_UNLOCK` – odemčení sdílené paměti (jen superuživatel)

- Odpojení

  ```
  int   shmdt(const void *shmaddr);
  ```

Semafory – slouží k jednoduché synchronizaci procesů, jsou realizovány polem čítačů, `<sys/sem.h>`

- Vytvoření identifikátoru, `nsems` udává počet semaforů

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

- Ovládání

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

  Některé operace, parametr `cmd`:

  - `IPC_STAT` – načtení řídící struktury semaforu
  - `IPC_SET` – nastavení některých polí struktury
  - `semid_ds`
  - `IPC_RMID` – odstranění semaforu ze systému
  - `GETVAL` – zjištění hodnoty semaforu
  - `SETVAL` – nastavení hodnoty semaforu
  - `GETPID` – zjištění sempid

- Operace nad semaforem

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

  struct sembuf {
    unsigned short sem_num; /* číslo semaforu */
    short sem_op;    /* operace nad semaforem */
    short sem_flg;      /* IPC_WAIT, SEM_UNDO */
  }
  ```

  Operace:

  - `sem_op` > 0 – hodnota `sem_op` se přičte k čítači,
  - `sem_op` < 0 – `sem_op` se odečte, v případě, že čítač je záporný volání se pozastaví do doby než bude 0,
  - `sem_op` $= 0$ – čekání dokud se semafor nevynuluje.

1.5. Vlákna

Zahrnout `<pthread.h>`, překládát s `-lpthread`.

pthread_create – vytvoření vlákna

```
int pthread_create(pthread_t *restrict threa21d,
          const pthread_attr_t *restrict attr,
          void *(*start_routine)(void*),
          void *restrict arg);
```

pthread_self – získání ID volajícího vlákna

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

pthread_equal – porovnání ID vláken

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

pthread_join – čekání na skončení vlákna

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