

# Procesy

Tématicky zaměřený vývoj aplikací v jazyce C  
skupina Systémové programování – Linux

Jiří Novosad

Fakulta informatiky  
Masarykova univerzita  
novosad@fi.muni.cz

Brno, 5. října 2014

## Úvod

### systemová volání

# Systemová volání

Jsou způsobem komunikace s jádrem systému.

```
man syscalls
```

## Chybové kódy systémových volání

- návratová hodnota: obvykle 0 v případě úspěchu, -1 při chybě

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

- příklady hodnot:

EACCES	Permission denied
EAGAIN	Resource temporarily unavailable
EBADF	Bad file descriptor
...	

- `man errno`

# Ošetření chyb

- tisk chybové zprávy na stderr

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

- řetězec popisující chybu

```
#include <string.h>
char *strerror(int errnum);
```

- skupina funkcí z err(3)

## Příklad

```
if ((fd = open("file.txt", O_RDONLY)) == -1) {
    fprintf(stderr, "open system call failed: %s\n",
            strerror(errno));
    /* perror("open system call failed"); */

    /* here you should cope with the error, e.g. exit program */
}
```

## Procesy

vytvoření, ukončení a čekání na ukončení procesů

# Co je to proces I

## Proces

Instance běžícího programu – objekt pracující podle kódu programu, má vlastní adresní paměťový prostor, využívá prostředky jádra a komunikuje s ostatními procesy.

# Co je to proces I

## Proces

Instance běžícího programu – objekt pracující podle kódu programu, má vlastní adresní paměťový prostor, využívá prostředky jádra a komunikuje s ostatními procesy.

- jednoznačná identifikace procesu pomocí *process id* (PID)

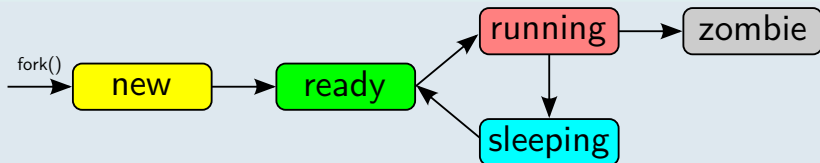
```
$ ps
```

```
  PID TTY          TIME CMD
 3370 pts/0    00:00:00 bash
28553 pts/0    00:00:59 kile
```

- každý proces má svého rodiče
  - výjimkou je pouze proces `init`
  - struktura procesů má podobu stromu v jehož kořeni je proces `init`
- další informace o procesu najdete v adresáři `/proc/<PID>/`

# Co je to proces II

## Životní cyklus procesu



### Stavy procesu (podle ps)

R – připravený nebo běžící (running)

S – blokový (sleep)

Z – zombie, defunct

T – pozastavený nebo krokovaný (stopped, traced)

D – nepřerušitelný spánek

X – mrtvý



# Vytvoření nového procesu I

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

- snadné použití
- blokující
- neefektivní – funkce `system()` vytváří proces shellu (interpret příkazů) a v rámci něj volá požadovaný příkaz  
`/bin/sh -c command`
- nespolehejte na dostupnost konkrétního shellu!  
Problém přenositelnosti není jen v jazyce C – bash, dash, ...
- potenciální bezpečnostní riziko

# Vytvoření nového procesu I

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

- snadné použití
- blokující
- neefektivní – funkce `system()` vytváří proces shellu (interpret příkazů) a v rámci něj volá požadovaný příkaz  
`/bin/sh -c command`
- nespolehejte na dostupnost konkrétního shellu!  
Problém přenositelnosti není jen v jazyce C – bash, dash, ...
- potenciální bezpečnostní riziko

## úkol

Napište program používající volání `system(3)`, který vytiskne seznam procesů.

# Vytvoření nového procesu II

```
#include <unistd.h>
pid_t fork(void);
```

- vytvoří nový proces jako kopii nadřazeného procesu
- neblokující – rozdělení běhu programu do dvou samostatných instancí
- nový proces se liší svým PID a návratovou hodnotou volání `fork()`

## Funkce pro zjištění PID

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

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

### úkol

Pomocí `fork()` vytvořte nový proces – oba procesy vytisknou hlášku, která je identifikuje.

(Např. "I'm the parent, pid = 12" a "I'm the child, pid = 43").

# Příklad – fork()

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

int main () {
    pid_t child; /* child's PID */

    child = fork();
    if (child == -1) {          /* some error occured */
        perror("fork() failed");
        return 1;
    } else if (child == 0) {   /* child process */
        printf("Child process\n");
        printf("\tPID          : %d\n", (int) getpid());
        printf("\tparent's PID: %d\n", (int) getpid());
    } else {                  /* parent process */
        printf("Parent process\n");
        printf("\tPID          : %d\n", (int) getpid());
        printf("\tchild's PID : %d\n", (int) child);
    }
    return 0;
}
```

# Skupina funkcí exec

```
#include <unistd.h>
int exeve(const char *filename, char *const argv[],
          char *const envp[]);

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);1
int execl_e(const char *path, const char *arg,
            ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);1
```

- nahrazení běžícího procesu jiným programem
- v případě úspěchu se funkce nikdy nevrací

---

<sup>1</sup>používá PATH!

### úkol

Rozšiřte předchozí příklad: potomek zavolá funkci z rodiny `exec` a vypíše běžící procesy. Rodičovský proces stále vypisuje svůj identifikační řetězec.

# Příklad – `exec1()`

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

int main () {
    pid_t child; /* child's PID */

    child = fork();
    if (child == -1) { /* some error ocured */
        perror("fork() failed");
        return 1;
    } else if (child == 0) { /* child process */
        execl("/bin/ls", "ls", "-l", (char *)NULL);
        /* never should be here */
        perror("exec failed");
        exit(1);
    }

    /* do some useful work */

    return 0;
}
```



# Čekání na ukončení procesu I

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

- čekání na ukončení libovolného z podřízených procesů
- pouze blokující režim

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

- čekání na ukončení (nebo změnu stavu – podle nastavení options) konkrétního podřízeného procesu
- blokující i neblokující (WNOHANG) režim

### úkol

Upravte předchozí program tak, aby rodičovský proces vypsal svůj text **VŽDY** jako poslední.

# Čekání na ukončení procesu II

```
#include <sys/types.h>
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- od verze Linuxového jádra 2.6.9
- umožňuje čekat na konkrétní změnu stavu
- identifikace procesu, na který se čeká, podle různých kritérií:  
P\_PID, P\_PGID, P\_ALL
- struktura `siginf_t` obsahuje po návratu stavové informace o procesu

# Příklad – waitid()

```
int main () {
    pid_t child; /* child's PID */
    int ret;
    siginfo_t info;

    child = fork();
    if (child == -1) {          /* some error ocured */
        perror("fork() failed"); exit(1);
    } else if (child == 0) {   /* child process */
        execl("/bin/ls", "ls", "-l", "myfile", (char *)NULL);
        /* never should be here */
        perror("exec failed"); _exit(1);
    }

    ret = waitid(P_PID, (int)child, &info, WEXITED);
    if (ret == -1) {
        perror("waitid failed"); return 1;
    }
    printf("Child's exit status: %d\n", info.si_status);

    return 0;
}
```

# Činnosti při ukončení procesu

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

- registrace funkcí prováděných při ukončení procesu (exit(3) a return ve funkci main)
- funkce se volají v opačném pořadí než jsou registrovány

# Činnosti při ukončení procesu

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

- registrace funkcí prováděných při ukončení procesu (exit(3) a return ve funkci main)
- funkce se volají v opačném pořadí než jsou registrovány

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

# Další operace s procesy

- asynchronní uklízení podřízených procesů
- ukončování jiných procesů
- a nejen to

# Další operace s procesy

- asynchronní uklízení podřízených procesů
- ukončování jiných procesů
- a nejen to

## Signály



## Úvod

nastavení a změna práv procesu

# Práva procesu I

Každý proces má přiřazena práva uživatele (UID) a skupiny (GID)  
– vlastně jich je několik.

## Reálné ID (RID)

- ID uživatele/skupiny, který spustil daný proces
- kontrolované při volání `access(2)`
- kontrolované při posílání signálů
- kontrolované při změně EID

## Efektivní ID (EID)

- ID uživatele/skupiny vlastníci (**setuid/setgid**) spustitelný soubor
- kontrolovaný při ostatních operacích (systémová volání)
- pokud  $EID \neq 0$ , lze ho změnit pouze na hodnotu RID nebo uloženého Set-User/Group-ID
- pokud  $EID = 0$ , lze ho změnit na jakoukoliv hodnotu

# Práva procesu II

## Uložené Set-User/Group-ID

- kopie EID při startu procesu

## set-UID/GID programy

- z příkazové řádky pomocí `chmod(1)`:

```
# chmod u+s myprog
# chmod 4711 myprog
# ls -l myprog
-rwsr-xr-x  1 root  root  302485 Jun 15 10:45 myprog
```

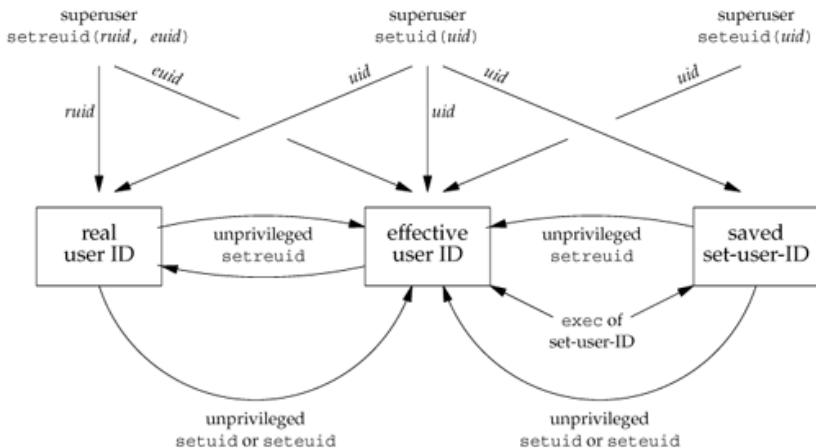
## POSIX funkce pro práci s ID

- `getuid()`, `geteuid()`, `getgid()`, `getegid()`
- `setuid()` (*a.k.a. one-way trip*), `setgid()`
- `seteuid()`, `setreuid()`, `setegid()`, `setregid()`

## GNU rozšíření

- `getresuid()`, `setresuid()`, `getresgid()`, `setresgid()`

# Změna práv procesu



# Dropping Privileges

```
/* Init UID: real=10 effective=0 saved=0 */  
orig_euid = geteuid();  
  
seteuid(getuid()); // drop privileges temporarily  
  
/* do some work with UID: real=10 effective=10 saved=0 */  
  
setuid(getuid()); // drop privileges permanently
```

# Dropping Privileges

```
/* Init UID: real=10 effective=0 saved=0 */
orig_euid = geteuid();

seteuid(getuid()); // drop privileges temporarily

/* do some work with UID: real=10 effective=10 saved=0 */

setuid(getuid()); // drop privileges permanently

/* WRONG! Since it's not a privileged process it affects only EID */
/* UID: real=10 effective=10 saved=0 */

/* do it right now */

seteuid(orig_euid); // restore privileges
/* UID: real=10 effective=0 saved=0 */

setuid(getuid()); // drop all privileges
/* UID: real=10 effective=10 saved=10 */

if (setuid(0) != -1) { // check if privileges really can't be restored
    /* they can, handle error */
}
```

# Nebezpečí privilegovaných procesů

- pracujte vždy s nejnižšími právy
- vyhněte se volání `access()` (time-of-check/time-of-use)
- pamatujte na hodnotu `umask()` (a mód volání `creat`)
- pozor na `exec()` (nebo `system()`, `popen()`, etc.)
  - dropněte práva
  - zavřete všechny nepotřebné file deskriptory
  - vyčistěte sadu proměnných prostředí
- vyhněte se `exec` funkcím používajícím PATH
- kontrolujte návratové hodnoty funkcí

# Dodatečné skupiny – supplementary groups

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

int getgroups(int size, gid_t list[]);

#include <grp.h>

struct group *getgrent(void);
void setgrent(void);
void endgrent(void);
int getgrouplist(const char *user, gid_t group,
                 gid_t *groups, int *ngroups);
```

- uživatel může být členem více skupin
- kromě primární skupiny je s procesem spojen seznam dodatečných skupin, lze jej získat voláním `getgroups()`
- lze získat seznam všech skupin; nestandardní volání pro získání všech skupin uživatele – `getgrouplist()`



## Závěr

domácí úkoly a zdroje

# Domácí úkol

Napište vlastní implementaci programu `id(1)`, s těmito přepínači a argumenty:

```
id [-gGnru] [user]
```

Zkuste `man` stránky k funkcím

- `getpwuid()`, `getpwnam()`
- `getgrgid()`, `getgrnam()`

# Zdroje

## ošetření chyb

- [www.ibm.com/developerworks/aix/library/au-errnovariable/](http://www.ibm.com/developerworks/aix/library/au-errnovariable/)

## procesy

- [www.linuxjournal.com/article/3814](http://www.linuxjournal.com/article/3814)
- [oreilly.com/catalog/linuxkernel/chapter/ch10.html](http://oreilly.com/catalog/linuxkernel/chapter/ch10.html)

## práva

- [www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf](http://www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf)
- [www.eecs.berkeley.edu/~daw/papers/setuid-login08b.pdf](http://www.eecs.berkeley.edu/~daw/papers/setuid-login08b.pdf)