

# Práce se soubory

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

Martin Drašar, Martin Husák, Petr Velan

Fakulta informatiky  
Masarykova univerzita  
`husakm@ics.muni.cz`

Brno, 16. listopadu 2015

## Práce se zařízeními

# ioctl

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```

- víceúčelové rozhraní pro řízení technického zařízení
- první argument je deskriptor souboru zařízení
- druhým argumentem je kód požadavku/operace, který dále určuje zbylé argumenty – viz `man 2 ioctl_list`
- operace jsou většinou vysoce specifické podle typu zařízení a vyžadují detailní znalost ovladačů

## úkol

- naimplementujte vlastní verzi programu `eject(1)`
- využijte postupy reverzního inženýrství

## Práce se soubory

Základy, které znáte a pokročilé techniky, které (možná) neznáte

# Základní operace

aneb co byste měli znát

- Otevření/zavření souboru – `open()`, `close()`
- Přejmenování souboru – `rename()`
- Smazání souboru – `unlink()`
- Čtení/zápis do souboru – `read()`, `write()`, ...
- Práce s adresáři – `mkdir()`, `rmdir()`, `opendir()`, `readdir()`, ...
- Práce se symbolickými odkazy – `link()`, `symlink()`, `readlink()`

# Přístupová práva souboru

- Kontrola přístupových práv – F\_OK, R\_OK, W\_OK, X\_OK
- Kontrolují se podle reálných práv procesu

```
#include <unistd.h>
```

```
int access(const char *pathname, int mode);
```

# Přístupová práva souboru

- Kontrola přístupových práv – F\_OK, R\_OK, W\_OK, X\_OK
- Kontrolují se podle reálných práv procesu

```
#include <unistd.h>  
int access(const char *pathname, int mode);
```

- Maska práv při vytváření souborů

```
#include <sys/types.h>  
#include <sys/stat.h>  
mode_t umask(mode_t mask);
```

- `permissions & ~mask`



# Vlastnosti souboru

- Čas přístupu a modifikace souboru

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

```
#include <utime.h>
```

```
int utime(const char *filename, const struct utimbuf *times);
```

```
#include <sys/time.h>
```

```
int utimes(const char *filename, const struct timeval times[2]);
```

# Vlastnosti souboru

- Čas přístupu a modifikace souboru

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *filename, const struct utimbuf *times);

#include <sys/time.h>
int utimes(const char *filename, const struct timeval times[2]);
```

- Komplexní informace o souboru

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

## fcntl()

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */ );
```

- Rozhraní pro řídicí operace na souborových deskriptorech.
  - změna režimu práce
  - zamykání souborů
  - ...
- Obdoba `ioctl()`
- Operace v zásadě generické, ale jejich význam se může lišit podle typu souboru

## fcntl()

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */ );
```

- Rozhraní pro řídicí operace na souborových deskriptorech.
  - změna režimu práce
  - zamykání souborů
  - ...
- Obdoba `ioctl()`
- Operace v zásadě generické, ale jejich význam se může lišit podle typu souboru

## Změna režimu práce se souborem

- `F_GETFL`, `F_SETFL`

# Zámky souborů I

/proc/locks

## Nevynucené (*Advisory*) zamykání

- dobrovolné, spoléhá na spolupráci všech procesů (podobně jako při synchronizaci a řízení přístupu ke sdíleným zdrojům)
- nemá přímý vliv na I/O operace
- zámek je uvolněn i při zavření kteréhokoliv deskriptoru, který ukazuje na stejný soubor!
- F\_GETLK, F\_SETLK, F\_SETLKW, struktura flock pak určuje jak zamykáme (F\_RDLCK, F\_WRLCK, F\_UNLCK) a co všechno

# Zámky souborů II

## Vynucené (*Mandatory*) zamykání

- problematická implementace v Linuxu (race condition)
- má přímý vliv na I/O operace
- může způsobit výrazné zpomalení podsystemu I/O
- striktní, ale vyžaduje speciální nastavení FS a práv

```
# mount -o mand
```

```
# chmod g+s,g-x file
```

Zamykání částí souborů nefunguje se streamovými funkcemi (`stdio.h`), proto je třeba v takovém případě používat nízkoúrovňové funkce `read()`, `write()`

# Zámky souborů III

## lockf()

- Zapouzdřené volání `fcntl()`  
`#include <unistd.h>`  
`int lockf(int fd, int cmd, off_t len);`
- POSIX lock
- Operace `F_LOCK`, `F_ULOCK`, `F_TEST`

## flock()

- Zamknutí **celého** (již otevřeného) souboru
- Jiný typ zámku než předešlé funkce  
`#include <sys/file.h>`  
`int flock(int fd, int operation);`
- Operace `LOCK_SH`, `LOCK_EX`, `LOCK_UN`

### úkol

- demonstrace zamykání (celého) souboru – program zamkne soubor a uvolní ho až na pokyn uživatele (stisk enteru)
- spusťte 2 instance programu a zkontrolujte jeho funkčnost



# Kopírování dat mezi soubory

# Kopírování dat mezi soubory

- Přímé přenesení dat z jednoho souboru do druhého – zero copy
- Operace bez mezipřenosu do uživatelského prostoru
- Možné problémy s přenositelností – běžný soubor může být výstupním souborem až od verze jádra 2.6.22

```
#include <sys/sendfile.h>  
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

# Kopírování dat mezi soubory

- Přímé přenesení dat z jednoho souboru do druhého – zero copy
- Operace bez mezipřenosu do uživatelského prostoru
- Možné problémy s přenositelností – běžný soubor může být výstupním souborem až od verze jádra 2.6.22

```
#include <sys/sendfile.h>
```

```
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

- Od jádra 2.6.17 jsou dostupné funkce pro kontrolu kernel bufferu
- splice(2), tee(2), vmsplice(2)

## Událostmi řízené programování

Zde v kontextu souborů

# Čekání na změnu stavu souboru I

- Sledujeme skupinu file deskriptorů a pokud nastane očekávaná událost (příchozí data), obsloužíme ji
- Vyhýbáme se busy-waitingu
- File deskriptory by měly být v neblokujícím režimu

# Čekání na změnu stavu souboru II

## select()

```
#include <sys/select.h>
int select(int nfd, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
int pselect(int nfd, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, const struct timespec *timeout,
            const sigset_t *sigmask);
```

- Pro práci s množinami file deskriptorů slouží makra `FD_CLR`, `FD_ISSET`, `FD_SET`, `FD_ZERO`
- Po návratu obsahují množiny pouze deskriptory souborů, kde nastala očekávaná událost
- Před každým voláním je třeba nastavit množiny deskriptorů znovu a po návratu celou množinu postupně kontrolovat

# Čekání na změnu stavu souboru III

## poll()

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
#define _GNU_SOURCE
int ppoll(struct pollfd *fds, nfds_t nfds,
          const struct timespec *timeout_ts, const sigset_t *sigmask)

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

- Méně přenositelné než `select()`
- Místo masky signálů je použita speciální struktura → pro každý deskriptor je možné specifikovat události, které nás zajímají

# Čekání na změnu stavu souboru IV

## epoll\_\*

```
#include <sys/epoll.h>
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

- Nepřenositelná skupina funkcí
- Nezávislá manipulace s každým deskriptorem
- Informace o nastalých událostech jsou dostupné lineárně (není třeba prohledávat)



# Monitoring událostí na souborech – *inotify*

- monitoring souborů a adresářů (ne rekurzivně!)
- od jádra 2.6.13, specifický mechanismus pro Linux
- nahrazuje *dnotify*
- existují i knihovny s podobnou funkcionalitou (FAM, Gamin)

# Monitoring událostí na souborech – *inotify*

- monitoring souborů a adresářů (ne rekurzivně!)
- od jádra 2.6.13, specifický mechanismus pro Linux
- nahrazuje *dnotify*
- existují i knihovny s podobnou funkcionalitou (FAM, Gamin)

```
#include <sys/inotify.h>
int inotify_init(void);
int inotify_init1(int flags);
int inotify_add_watch(int fd, const char *pathname, uint32_t mask);
int inotify_rm_watch(int fd, int wd);
```

- pracujeme s *inotify* **file deskriptorem**
- *flags* mohou být *IN\_NONBLOCK* a *IN\_CLOEXEC*
- hodnoty pro masku sledovaných událostí viz man 7 *inotify*
- *wd* je watch deskriptor vrácený funkcí *inotify\_add\_watch()*

# Zpracování *inotify* událostí

- čtení (`read(2)`) z *inotify* fd vrací buffer s jednou nebo více strukturami `inotify_event`
- čtení je blokující, lze použít i `select()/poll()/epoll()`
- velikost jednoho záznamu `inotify_event` je `sizeof(struct inotify_event) + len`

```
struct inotify_event {
    int      wd;          /* Watch descriptor */
    uint32_t mask;       /* Mask of events */
    uint32_t cookie;     /* Unique cookie associating related
                          events (for rename(2)) */
    uint32_t len;        /* Size of name field */
    char     name[];     /* Optional null-terminated name */
};
```

## Závěr

shrnutí, domácí úkoly a zdroje

# Domácí úkol

- world.pid soubor zamykejte (lockf/fcntl), běžící proces drží zámek
- Při spuštění dalšího worldu a existenci world.pid čeká nový world pomocí inotify na smazání existujícího world.pid
- Pokud se pomocí inotify probudí více worldů, běží jen ten, který dostane zámek na world.pid

# Zdroje

- [www.gnu.org/s/libc/manual/html\\_node/IOCTLs.html](http://www.gnu.org/s/libc/manual/html_node/IOCTLs.html)
- [man-wiki.net/index.php/2:ioctl\\_list](http://man-wiki.net/index.php/2:ioctl_list)
- [www.hackinglinuxexposed.com/articles/20030623.html](http://www.hackinglinuxexposed.com/articles/20030623.html)
- [www.makelinux.net/ldd3/chp-6-sect-3.shtml](http://www.makelinux.net/ldd3/chp-6-sect-3.shtml)
- [www.unixguide.net/network/socketfaq/2.14.shtml](http://www.unixguide.net/network/socketfaq/2.14.shtml)
- [daniel.haxx.se/docs/poll-vs-select.html](http://daniel.haxx.se/docs/poll-vs-select.html)
- [www.kegel.com/c10k.html](http://www.kegel.com/c10k.html)