PB173 Linux

10 Udalosti

Roman Lacko xlacko1@fi.muni.cz

2022-11-25

Obsah

Udalosťami riadené programovanie

Udalosťami riadené programovanie

Procedural Programming

Procedurálne programovanie (tradičný prístup)

- Program vykonáva sekvenciu krokov.
- V každom čase sa "rozhodne" podľa dosiahnutého stavu, ako bude pokračovať.

Event-Driven Programming

Udalosťami riadené programovanie

- Program čaká na udalosť v cykle (*Event Loop*).
- Po zaregistrovaní udalosti program vykoná nejakú úlohu, a vráti sa k čakaniu na ďalšie udalosti.
- Použitie: GUI, ovládače, viac-vláknové aplikácie, ...

EDP: Schéma

```c
int handle_signal(event_t event, int sig);
int handle_input(event_t event, int fd);

int main(int argc, char *argv[])
{
    event_handler(&handle_signal, EV_SIGNAL, SIGINT);
    event_handler(&handle_input, EV_FD, STDIN_FILENO);

    event_t event;
    while (event_wait(&event))
        event_process(&event);

    return EXIT_SUCCESS;
}
```

Knižnice typicky *Event Loop* skrývajú vo funkcii.

EDP: Základné rozhrania

`select()`, `poll()`
Čakajú na udalosť na *deskriptoroch*.

Ak vieme zistiť udalosť na deskriptore, môžeme monitorovať *skoro každý* zdroj, ktorý má deskriptor.

Inotify (i-node notify)	Zmeny v súborovom systéme
`eventfd()`	Generické udalosti (IPC)
`pidfd_open()`	Monitorovanie potomka
`signalfd()`	Signály
`socket()`, `listen()`, …	Sieťové kanály
`timerfd_create()`	Časovače

EDP: Zmeny na deskriptoroch: `select()`

```c
#include <sys/select.h>

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

int pselect(… nfds, … readfds, …writefds, …exceptfds, … timeout,
        const sigset_t *sigmask);
```

⚠ Preferujte `poll()`

Volanie `select()` pri návrate **zmení** vstupné množiny.
Preto je trochu nepohodlné.

Používajte, len ak cielite na starší systém.

EDP: Zmeny na deskriptoroch: `poll()`

```c
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
int ppoll(… fds, … nfds, … timeout, const struct timespec *timeout
          const sigset_t *sigmask);

struct pollfd {
    int fd;
    short events;
    short revents;
};

#define POLLIN /* … */
#define POLLOUT /* … */
#define POLLHUP /* … */
#define POLLERR /* … */
```

EDP: Zmeny na deskriptoroch: `epoll()`

```c
#include <sys/epoll.h>

int epoll_create(int size);
int epoll_create1(int flags);

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

#define EPOLL_CTL_ADD /* … */
#define EPOLL_CTL_MOD /* … */
#define EPOLL_CTL_DEL /* … */

union epoll_data { … };

struct epoll_event {
    uint32_t events;
    union epoll_data data;
};
```

EDP: Zmeny na deskriptoroch: `epoll()`

```
int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
int epoll_pwait(… epfd, … events, … maxevents, … timeout,
               const sigset_t *sigmask);
int epoll_pwait2(… epfd, … events, …maxevents,
               const struct timespec *timeout,
               const sigset_t *sigmask);
```

Nie je POSIX

Toto rozhranie je špecifické pre Linux.
Je však výkonnejšie pre veľký počet deskriptorov než `poll()`.

EDP: Inotify

```c
#include <sys/inotify.h>

int inotify_init(void);
int inotify_init1(int flags);

int inotify_add_watch(int fd, const char *path, uint32_t mask);
int inotify_rm_watch(int fd, int wd);

#define IN_CLOSE_WRITE /* … */
#define IN_CLOSE_NOWRITE /* … */
#define IN_CREATE /* … */
#define IN_DELETE /* … */
…
```

EDP: Inotify

Udalosť vráti `read()` z Inotify deskriptora.

```
struct inotify_event {
    int wd;
    uint32_t mask;
    uint32_t cookie;
    uint32_t len;
    char name[];    /* Flexible Array Member */
};

/* Minimum buffer size to read at least one event. */
char buffer[sizeof(struct inotify_event) + NAME_MAX + 1];
read(inotify_fd, buffer, sizeof(buffer));
```

`read()` môže do poľa prečítať **viac než jednu** udalosť,
ak sa tam zmestia. Kernel žiadnu udalosť nerozdelí.

EPD: Generické udalosti

```c
#include <sys/eventfd.h>

int eventfd(unsigned int initval, int flags);
#define EFD_CLOEXEC /* … */
#define EFD_NONBLOCK /* … */
#define EFD_SEMAPHORE /* … */

/* GNU libc */
typedef uint64_t eventfd_t;
int eventfd_read(int fd, eventfd_t *value);
int eventfd_write(int fd, eventfd_t value);
```

EPD: Generické udalosti

Čo sú udalosti v `eventfd()`?

- Kernel udržuje celočíselné počítadlo
- `write()` pridáva hodnotu k počítadlu
- `read()` prečíta hodnotu počítadla a reštartuje ho;
  pre `EFD_SEMAPHORE` dekrementuje počítadlo a prečíta 1
- Môže sa kombinovať s `poll()`
- Môžu používať *asynchrónne rozhrania* (napr. `io_uring`)

Jednoduchý signálny mechanizmus

Typicky sa používa ako „light-weight signal pipe“.

EDP: Ďalšie mechanizmy

Signály

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

int signalfd(int fd, const sigset_t *mask, int flags);
```

`read()` číta `struct signalfd_siginfo` štruktúry.

Procesy

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

int syscall(SYS_pidfd_open, pid_t pid, unsigned int flags);
```

Nepodporuje `read()`, treba použiť `waitpid()`.

EDP: Ďalšie mechanizmy

Časovače

```
#include <sys/timerfd.h>
int timerfd_create(int clockid, int flags);
int timerfd_settime(int fd, int flags,
        const struct itimerspec *new_value,
        struct itimerspec *old_value);
int timerfd_gettime(int fd, struct itimerspec *value);

struct itimerspec {
    struct timespec it_interval;
    struct timespec it_value;
};
```

- Analogické `timer_create()`
- `read()` vráti počet expirácií

EDP: Ďalšie mechanizmy

Sieť

Spoiler alert

Nabudúce...

EPD: `libev`

Event-Loop Library (`libev`)

- Podpora viacerých platforiem
- Jednotné rozhranie
  (aj pre iné jazyky, C++, Perl, ...)
- Transparentne vyberie najlepší mechanizmus
  `select()`, `poll()`, `epoll()`, AIO, KQueue, ...

⚠ **`libevent`**

Staršia knižnica, robí trochu viac vecí.

Asynchrónne programovanie

Asynchrónne programovanie

Asynchronous Programming
Asynchrónne programovanie

- Program spracúva úlohu.
- Spracovanie môže niečo prerušiť,
  a môže sa začať spracovať iná úloha.
- Rozpracovaných úloh môže byť naraz niekoľko.

Na rozdiel od EDP program na udalosť
nemusí explicitne čakať.

Asynchrónne programovanie

Program môže výpočet rozdeliť na *asynchrónne bloky*
Výsledok potom reprezentujú dva koncepty:

Promise ≈ výpočet

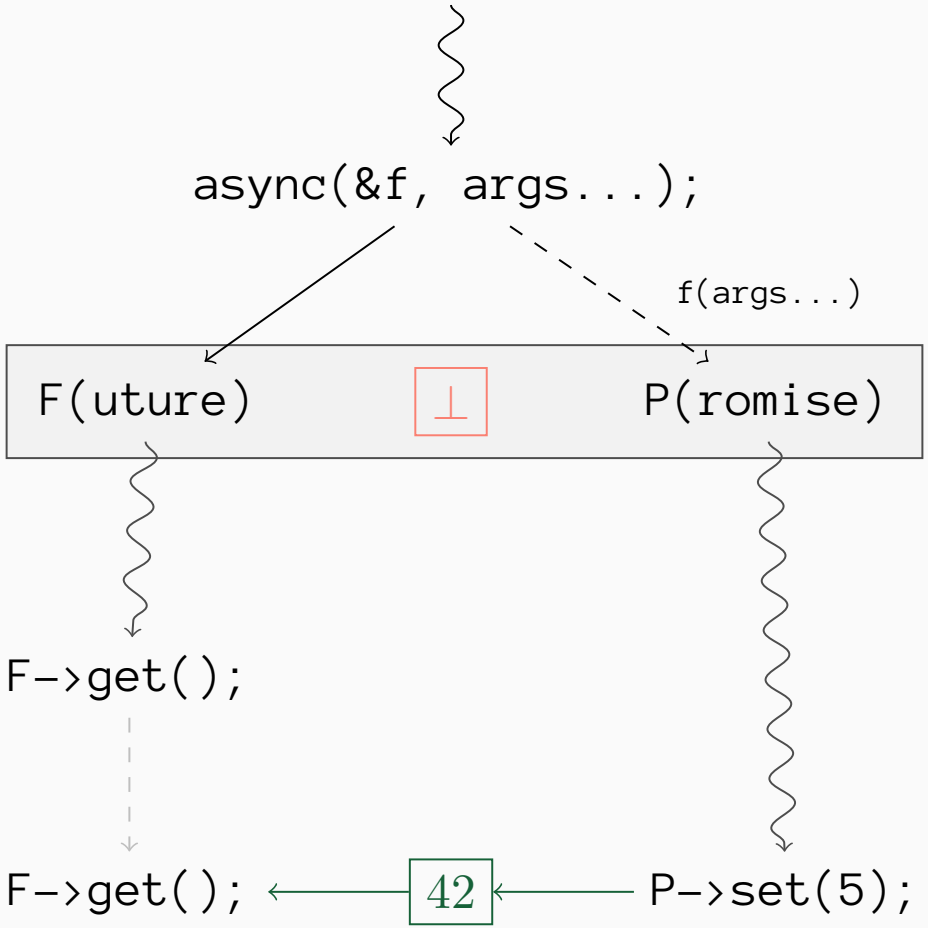
- Zapisujúci koniec výsledku
- Výsledok môže byť aj „zlyhanie“

Future ≈ klient

- Čítajúci koniec výsledku
- Môže požiadať o zrušenie asynchrónneho bloku

Význam *Future*, *Promise* a ich vlastnosti závisia na zdroji, niekde sa môžu používať ako synonymum.

Abstraktná schéma



Asynchrónne programovanie

Často priama podpora vo vyšších programovacích jazykoch

- C++ `std::async()`, `std::future()`, `std::promise()`
- Python, C#, ... `async`, `await`

C priamu podporu nemá, používa nízkoúrovňové mechanizmy.

Coroutines

V princípe podobný koncept: *koopeatívny multitasking*.

Asynchrónne programovanie: AIO

POSIX Asynchronous I/O

```
#include <aiocb.h>
struct aiocb {
    int aio_lio_opcode; /* LIO_READ, LIO_WRITE */
    int aio_fildes;
    off_t aio_offset;
    size_t aio_nbytes;
    volatile void *aio_buf;
    struct sigevent aio_sigevent;
};

#include <aio.h>
int aio_read(struct aiocb *aiocbp);
int aio_write(struct aiocb *aiocbp);
int aio_error(const struct aiocb *aiocbp);
int aio_return(struct aiocb *aiocbp);
```

Asynchrónne programovanie: AIO

POSIX Asynchronous I/O

```
int lio_listio(int mode, struct aiocb *const list[],
        int nitems, struct sigevent *sevp);

#define LIO_WAIT /* … */
#define LIO_NOWAIT /* … */
```

Asynchrónne programovanie: Linux IO

Linux Asynchronous I/O

- `man 3 io(libaio)`
- Pôvodne operácie s diskom, dnes všeobecnejšie

Linux I/O User Ring

- `man 7 io_uring`
- Dátové štruktúry zdieľané s kernelom
  Submission Queue, Completion Queue
- Sľubuje oveľa väčší výkon než AIO
- Zložité na nastavenie

Asynchrónne programovanie: `libuv`

Unicorn Velociraptor Library (`libuv`)

- Asynchronous Event-Driven Library
- **Zjednodušene** `libev` s *Thread Pool*

Trivia

`libuv` si autor vymyslel bez významu,
Unicorn Velociraptor vymyslel dodatočne.

Pokročilé súborové operácie

File Control

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

#define F_GETFD /* … */ /* File Descriptor Flags */
#define F_SETFD /* … */
#define F_GETFL /* … */ /* File Description Flags */
#define F_SETFL /* … */
```

File Control

Prehľad možností

- Duplikácia deskriptora
- Zistenie a zmena príznakov
- Zamykanie častí súboru
- Generovanie udalostí nad súborom
- Zmena kapacity rúry
- Pečate
- Optimalizácie čítania súboru

File Control: Zámky

Mandatory Locks

- Vynútené zamykanie
- Model v Microsoft Windows
- Môže výrazne spomaliť I/O
- Problematická implementácia (súbeh)
- Linux: **Zámerne** zložité povolenie

```
mount -o mand …
chmod g+s,g-x file
```

Porovnajte s `open()` → `ETXTBSY`.

File Control: Zámky

Advisory Locks

- Nevynútené, kooperatívne zamykanie
- Analogické ku synchronizácii vláken
- Nemá priamy vplyv na I/O

File Control: POSIX Record Lock

POSIX Record Lock

```
int fcntl(int fd, int cmd, struct flock *flockptr);
#define F_SETLK  /* … */
#define F_SETLKW /* … */
#define F_GETLK  /* … */

struct flock {
    short l_type;   // F_RDLCK, F_WRLCK, R_UNLCK
    short l_whence; // SEEK_SET, SEEK_CUR, SEEK_END
    off_t l_offset;
    off_t l_len;
    pid_t l_pid;
};
```

File Control: POSIX Record Lock

POSIX Record Lock

- Zámky priradené páru `[PID, i-node]`
- Atomické konverzie medzi typmi zámku

Porovnajte s `pthread_rwlock_t`.

⚠ Neintuitívne správanie

- Každý deskriptor pre daný súbor je zamknutý
  Nepoužiteľné na vzájomné vylúčenie vláken
- Zatvorenie *hociktorého* z deskriptorov odomkne zámok
  `getpwent()` … `endpwent()`

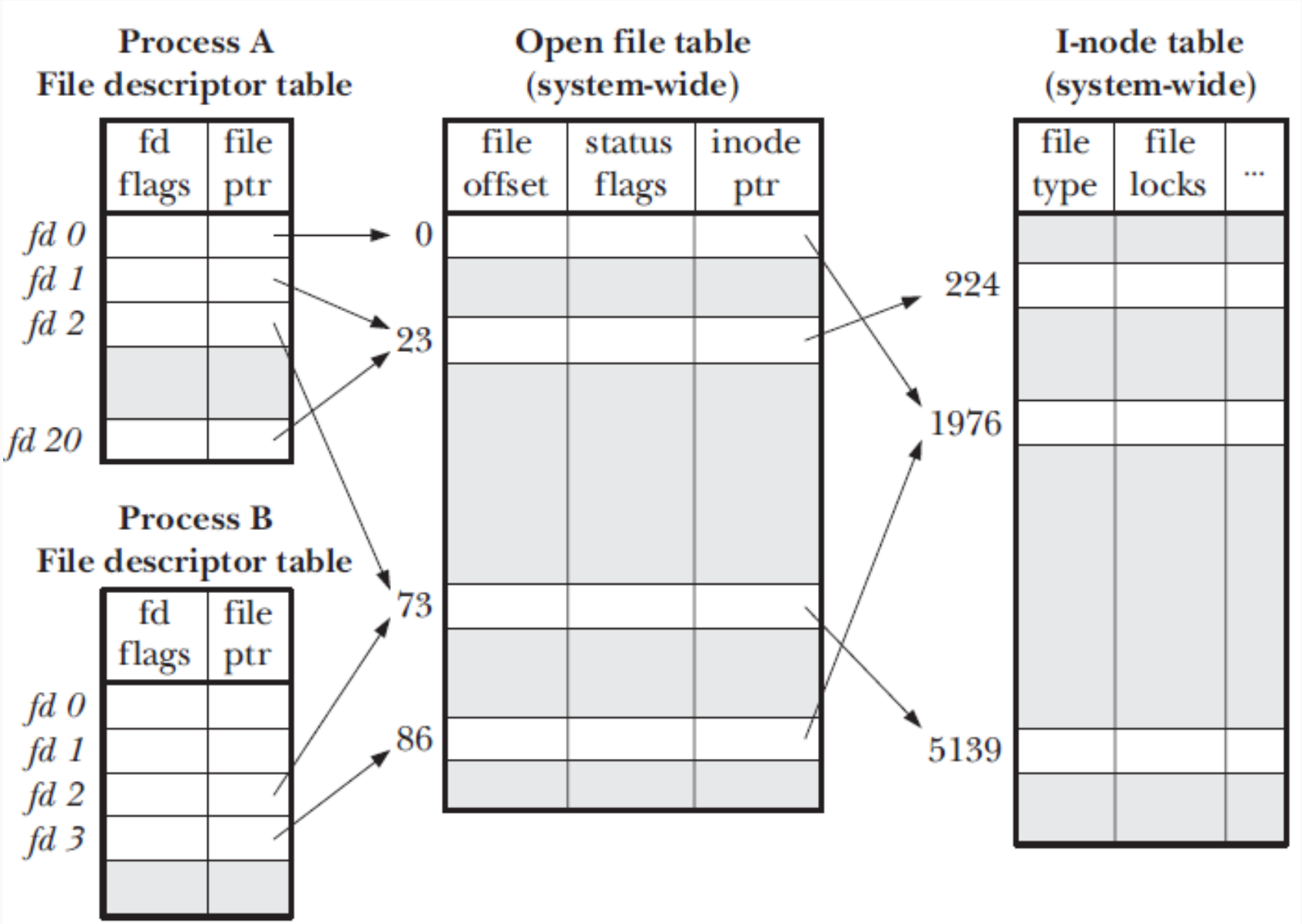
File Control: Open File Descrption Lock

Open File Description Lock (OFD)

```
int fcntl(int fd, int cmd, struct flock *flockptr);
#define F_OFD_SETLK  /* … */
#define F_OFD_SETLKW /* … */
#define F_OFD_GETLK  /* … */
```

- Linuxové rozhranie
- Naviazané na záznam v *Open File Table*
- Možné použiť s vláknami
- Nepodporujú atomickú konverziu zámkov

Odbočka: Open File Description



Odbočka: Iné rozhrania

Lock File

```
#include <unistd.h>
int lockf(int fd, int cmd, off_t len);
/* F_LOCK, F_TLOCK, F_ULOCK, F_TEST */
```

- Zjednodušené rozhranie nad POSIX Record Lock
- `F_LOCK` ≈ `F_WRLCK`

POSIX interakciu medzi `lockf()` a `fcntl()` nešpecifikuje

Odbočka: Iné rozhrania

BSD Lock

```c
#include <sys/file.h>
int flock(int fd, int operation);
/* LOCK_SH, LOCK_EX, LOCK_UN */
```

- OFD zámky
- Neinteraguje so zámkami `fcntl()`
- Nefunguje nad NFS (kernel emuluje `fcntl()`)

Dočasné súbory

Staré rozhranie

Nepoužívajte tieto funkcie, sú náchylné na súbeh!

```
#include <stdlib.h>
char *mktemp(char *template);
char *tempnam(const char *dir, const char *pfx);
char *tmpnam(char *s);
char *tmpnam_r(char *s);
```

- Vrátia unikátny názov súboru
- Medzi volaním `mktemp()` a `open()` je súbeh

Dočasné súbory

```c
#include <stdlib.h>
int mkstemp(char *template);
int mkostemp(char *template, int flags);
int mkstemps(char *template, int suffixlen);
int mkostemps(char *template, int suffixlen, int flags);

char *mkdtemp(char *template);
FILE *tmpfile(void);
```

- **Atomicky** vytvoria súbor a vrátia deskriptor
- Šablóna tvaru …XXXXXX

Rozšírené atribúty

Extended Attributes

- Arbitrárne hodnoty priradené k súboru
- Špeciálne bloky na disku
  (súborový systém nemusí podporovať)

Použitie:

- Vlastné atribúty
- *Access-Control Lists*
- Značky bezpečnostných modulov (napr. SELinux)

Rozšírené atribúty

Menné priestory

Každý atribút má názov tvaru `NAMESPACE.NAME`,
kde `NAMESPACE` je menný priestor:

`user`	používateľké atribúty
`trusted`	atribúty administrátorských procesov
`system`	atribúty pre služby jadra
`security`	atribúty pre bezpečnostné moduly

Programy: `getfattr`, `setfattr`

Neprivilegované procesy vidia len user atribúty.

Rozšírené atribúty

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

ssize_t listxattr(const char *path, char *list, size_t size);
int setxattr(const char *path, const char *name,
        const void *value, size_t size, int flags);
ssize_t getxattr(const char *path, const char *name,
        void *value, size_t size);
int removexattr(const char *path, const char *name);

/* Plus la() and fa() */
```

`listxattr(P, NULL, 0)` vráti potrebnú veľkosť reťazca

Rozšírené atribúty

Access Control Lists

- Rozšírenie štandardných POSIX práv
- `system.posix_acl_access`
- `man 5 acl`

File Capabilities

- Schopnosti naviazané na spustiteľný súbor
- `setcap`
- `security.capability`

Zariadenia: Interakcia s ovládačom

Väčšina zariadení existuje ako súbor v `devfs` (`/dev`)
Deskriptor je možné získať pomocou `open()`.

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

int ioctl(int fd, unsigned int command, /* … */);
```

- `man -k ioctl_`

Záver

Záver

EDP, Async

- `poll()` vs `select()`
- Register an `eventfd`
- `io_submit`: The epoll alternative ...
- Introduction to `libuv`

Zámky

- File Locking in Linux
- Mandatory Locking in Linux