# Proradná obsluha chyb 🔧

Kar(t)el Kočí

3.10.2020

# Kdo za ty chyby může?

# Kdo za ty chyby může?

**Uživatel!**

# Kdo za ty chyby může?

Programátor..

# Co si pod chybami programátor představí?

- **Nečekaný stav dat**
- Nečekaný běh aplikace (špatný algoritmus)
- Nečekané ukončení aplikace (segfault, ..)
- Nečekaný obsah souboru či čtení ze soketu
- Nečekaný obsah sdílené paměti
- **Nečekaná návratová hodnota**
- Nečekaný stav souborů (chybějící složka atd.)
- Nečekaná odezva od periferie či soketu
- A další ...

# Co si pod chybami představí uživatel?

Nefunguje to!

# Teď vážně, požadavky

Uživatel:

- Nechce, aby program spadl
- Když už spadne, tak chce vědět jak to opravit
- Když už neřekneme jak opravit, tak chce vědět proč
- Nechce více informací než dokáže vstřebat

Programátor:

- To někdy není možné a je to jediné východisko
- Programátor nemůže pokrýt každou situaci
- To nemusí být ani v moci programu
- Potřebuje trace, například aby dohledal jak se to stalo

Programátor není Bůh, ale měl by být a měl by se snažit uživatelům práci s jeho stvořením zpříjemnit.

# Co s chybou když nastane?

Záleží na tom o jakou chybu se jedná.

- Fatální chyby
  - Pády (*segfault, invalid instruction, ...*)
  - Nečekané stavy (*tady ještě před chvílí byl soubor ...*)
  - Bezvýchodné stavy (*tady měl být soubor se vstupem ...*)
- Řešitelné chyby
  - Krajní (*soubor vytvořil někdo jiný ...*)
  - Očekávané (*soubor se nezdařilo vytvořit, protože chybí složka, vytvoříme složku ...*)
  - Ignorované (*unlink souboru selhal, protože neexistuje ...*)

A taky záleží, jestli jsme knihovna nebo program!

# Co když nastane pád?

**Program**

Pokud se nesnaží, tak se ani o tom nedozví.

**Knihovna**

Ani se o tom nedozví a většinou by se neměla ani snažit.

```
struct foo *foo = NULL;
foo->fee = 0;
```

# Co když nastane nečekaný stav?

**Program**

Ukončení programu a výpis stavových informací

```
file = open(path, "w")
file.close()
file = open(path)
# raises: FileNotFoundError
```

```
fclose(file);
file = fopen(path, "r");
assert(file);
```

**Knihovna**

Necháme vyřešit aplikaci. Nikdy ne `exit` a vyvarujte se `abort`.

```
fclose(file);
file = fopen(path, "r");
if (file == NULL) {
  liberrno = LIB_ERR_LOST_FILE;
  return false;
}
```

# Odbočka k C a knihovnám: Jak hlásit error?

- Návratová hodnota funkce

```
enum {
  FOO_ERR_FOO,
} foo() { return FOO_ERR_FOO; }
```

- Proměnná s chybou

```
enum liberror {
  LIB_ERR_FOO,
};
thread_local enum liberror liberrno;
```

- Výjimka (s nějakou knihovnou, která je implementuje)

# Co když nastane bezvýchodný stav?

## Program

Ukončení programu.

```
try:
  file = open(sys.argv[1])
except FileNotFoundError:
  logger.critical("Input missing: %s",
    sys.argv[1])
except PermissionError:
  logger.critical(
    "Input unaccessible: %s",
    sys.argv[1])
```

## Knihovna

Nechám vyřešit aplikaci.

```
def handle_args(argv):
  """
  :raises FileNotFoundError:
    First argument wasn't valid path
  :raises PermissionError:
    First is unaccessible path
  """
  file = open(argv[1])
```

# Odbočka k ukončovacím rutinám: Jak ukončit program?

Je nutné uvolnit každou dosažitelnou paměť?

Rozdělme si zdroje vůči procesu:

Interní

- Alokovaná paměť
- Otevřené file descriptory (soubory, sockety, atd.)
- Namapovaná paměť

Externí

- Data buffery s výstupem na disk (ne `fclose` ale `fflush`)
- Dočasné soubory dosažitelné na file-systému

Dělejme jen to, co musíme. Zbytek za nás udělá kernel a jazyk.

Interní za nás udělá někdo jiný, my se musíme starat pouze o externí.

# Ještě větší odbočka k C exit

https://www.gnu.org/software/libc/manual/html_node/Termination-Internals.html

```
typedef void (*onexit_t)(void *arg);
struct onexit { onexit_t func; void *arg; struct onexit *next; };
struct onexit *onexit = NULL;
void onexit_handler() {
  for (; onexit; onexit = onexit->next)
    onexit->func(onexit->arg);
}

atexit(onexit_handler);
```

# Co když nastane řešitelná chyba?

**Program**

Tak ji prostě vyřešíme, nebo ještě lépe ji předejdeme.

```
pathlib.Path("foo").mkdir(
  parents=True, exist_ok=True)
file = open("foo/fee", "w")
```

**Knihovna**

Tak ji vyřešíme. Lepší je ale nechat vybrat aplikaci, jestli se má automaticky řešit nebo vrátit raději error.

```
def open(self, path, mkdir=True):
  if mkdir:
    pathlib.Path(path).parent.mkdir(
      parent=True, exist_ok=True)
  file = open(path, "w")
```

# Závěrem: jde to i přehnat

```
int fd = open(path, O_RDONLY);
char buf[BUFSIZ];
while (true) {
        if (read(fd, buf, BUFSIZ) == -1) {
                switch (errno) {
                        case EWOULDBLOCK:
                        case EAGAIN:
                                continue;
```

Error: unable to exit, user is not informed!

# Nahlášení chyby

Před tím než chybu ošetříme, musíme ji nahlásit!

V pořadí od vývojářsky po uživatelsky přívětivé:

- Memory dump
- Stack trace
- Popis chyby se zdrojem chyby (zdrojový soubor, funkce, řádka, ...)
- Popis chyby
- Popis chyby a návrh řešení

# Co kdy zvolit ale?

**Memory dump** jen pro pády a většinou zajistí systém

**Stack trace** jen pro nečekané stavy a na vyžádání

**Zdroj chyby** jen jako méně invazivní alternativa ke stacktrace

**Popis** vždy

**Návrh řešení** vždy pokud je možné

# Co v popisu říci uživateli, aby jim i nám to k něčemu bylo?

1. Popis by měl být pokud možno unikátní pro každou chybu.
2. Poskytnout výpis relevantních dat (cesta k souboru a pod.).
3. Nevypisujte nerelevantní data!
4. Popis by měl být výstižný a přesný.

Ujistěte se, že data která vypisujete jsou validní!

```
Called uri_path on URI of scheme: https
```

```
-error("URI download failed (" .. u:path() .. "): " .. u:download_error())
+error("Getting URI (" .. u:uri() .. ") failed: " .. u:download_error())
```

# Kdy hlásit chybu?

**Vždy!**

Hlašte chyby ve více úrovních:

| | |
|---|---|
| **CRITICAL** | pro nečekané a bezvýchodné stavy (následuje ukončení programu) |
| **ERROR** | pro krajní řešitelné chyby |
| **WARNING** | pro krajní řešitelné chyby |
| **NOTICE** | žádné chyby |
| **INFO** | pro očekávané chyby |
| **DEBUG** | pro ignorované chyby |

# Checklist závěrem

1. Zamyslete se, co za data od volání získáte, a jak moc datům ke kterým přistupujete můžete věřit
2. Chyby které nám mohou nastat rozřadíme
3. Při detekci chyby oznámíme uživateli
4. Vyřešíme chybu dle toho o jaký typ se jedná

**Fatal error: no more slides** 😢

Děkuji za pozornost.

`git.cynerd.cz`