POSIX, Make, CMake

Miroslav Jaroš

PB071 Úvod do nízkoúrovňového programovnání

4. května 2018

Obsah

POSIX

Motivace

Proč se zabývat operačním systémem?

- Standardní knihovna přináší programátorovi základní funkci pro práci s prostředky počítače
- Nicméně kdykoliv program potřebuje interagovat s hw musí požádat OS o zpřístupnění
  - Práce se soubory
  - Požadavek na naalokování stránky do virtuální paměti
  - Spuštění podprocesu v shellu
  - Řízení vláken (od C11)
- Jak standardní knihovna zvládá toto všechno implementovat?

Motivace

Historie

Aneb od UNIXu ke standardu

- `POSIX - Portable Operating System Interface`
- Norma pro rozhraní operačního systému založená na operačním systému `UNIX`
  - UNIX vznikl roku 1971 a již roku 1973 byl přepsán do jazyka C
  - Jeho autoři jsou Dennis M. Ritchie a Ken Thompson
- Součástí normy POSIX je knihovna pro jazyk C – `POSIX C Library`, která definuje základní rozhraní operačního systému
- Pro používání ochrané známky UNIX musí operační systém plně implementovat normu POSIX a být certifikovaný podle Single Unix Specification
  - Např. macOS je certifikovaný UNIX
  - Linux není
- Nicméně většina `UNIX-like` (nebo také UN*X) systémů jej dodržuje (s odchylkami)

POSIX C Library

API operačního systému

- Zpřistupňuje funkce pro interakci s operačním systémem
- Snaží se o vytvoření API kompatibilního se standardní knihovnou C, která je její součástí
- Pokrývá širokou škálu služeb jádra poskytovaných programům
  - Správa procesů (start, komunikace, ukončení)
  - Přístup k souborovému systému, síťovému rozhraní, pipes
  - Správa a synchronizace vláken (spouštění, vyloučení přístupu, semafory . . .)
  - Správa virtuální paměti procesu (mapování stránek, dealokace . . .)
  - A mnoho dalších . . .
- Tím umožňuje psaní "přenositelných" programů s daleko širším záběrem, než má standardní knihovna

Kompatibilita

- `UN*X` systémy (Linux, macOS, Solaris . . .)
  - Pokud je systém certifikován jako UNIX, potom splňuje POSIX
  - Linux jej s minimálními odchylkami splňuje taktéž
- Windows
  - Má vlastní API operačního systému `Win32` a `WinRT`
  - Nicméně části POSIX implementuje, ale ne plně
  - MinGW a Cygwin implementuji POSIX prostředí pomocí `Win32` API
  - Od Windows 10 obsahuje `Windows Subsystem for Linux`
    - Emuluje rozhraní Linuxu, pro běh linuxových aplikací
    - Pro instalaci a další zrdoje viz tutorial
    - `https://www.fi.muni.cz/pb071/tutorials/ubuntu-on-windows/index.html`

Práce se soubory

- Velice podobná jako ve standardní knihovně
- Místo struktury `FILE *` se používá `file deskriptor`, který je typu `int`
- `int open(const char *path, int oflag, ...);`
- `int close(int fd);`
- `ssize_t read(int fildes, void *buf, size_t nbyte);`
- `ssize_t write(int fildes, const void *buf, size_t nbyte);`
- `ssize_t` je rozšíření typu `size_t` o záporná čísla
- Manuálové stránky těchto funkcí dle POSIX `man 3 $JMENO_FUNKCE`

Adresáře

- Adresář je specifická entita v rámci file systému a její formát na něm záleží
- Stejně jako u souborů (jimiž ve skutečnosti většinou i bývají) je práce s nimi programátorovi zpřístupněna operačním systémem
- Pro práci s adresáři se používá kombinace funkcí `opendir`, `readdir` a `closedir`
- `DIR *opendir(const char *name);`
  - Vrací ukazatel na `DIR`, což je reprezentace otevřeného adresáře pro OS
- `struct dirent *readdir(DIR *dirp);`
  - Přečte další položku v adresáři
  - Pokud v adresáři není žádný další prvek, vrací `NULL`
- `int closedir(DIR *dirp);`
  - Ukončuje práci s adresářem a uvolňuje zdroje
  - I zde platí stejně jako u `fopen` nebo `malloc`, že pokud funkce `opendir` neselhala, musí být nad její návratovou hodnotou zavolána funkce `closedir`

Adresáře II.

Po zavolání funkce `readdir` je vrácena `struct dirent *`

```
struct dirent {
       ino_t          d_ino;       /* Inode number */
       off_t          d_off;       /* Not an offset; see below */
       unsigned short d_reclen;    /* Length of this record */
       unsigned char  d_type;      /* Type of file; not supported
                                      by all filesystem types */
       char           d_name[256]; /* Null-terminated filename */
};
```

- Položka `d_name` obsahuje jméno prvku ve složce, ale ne cestu, ta musí být zrekonstruována jiným způsobem
- Položka `d_type` může obsahovat typ prvku, ale v závislosti na použitém file systemu také nemusí
- Pro zjištění typu lze `d_type` testovat proti konstantám
  - `DT_REG` běžný soubor
  - `DT_DIR` adresář
  - `DT_UNKNOWN` File system nepodporuje `d_type` a typ je potřeba zjistit jinak, například voláním `lstat`
  - A dalším (viz `man 3 readdir`)

Procházení adresáře

```
void PosixPrintFiles(const char* path) {
    DIR *dir = NULL;
    if ((dir = opendir(path))) { // connect to directory
        struct dirent *dirEntry = NULL;
        while ((dirEntry = readdir(dir)) != NULL) {// obtain next item
          printf("File %s\n", dirEntry->d_name); // get name
        }
    closedir(dir); // finish work with directory
  }
}
```

Procesy

- Proces je v OS jednotka pro běh samostatného programu s vlastní oddělenou pamětí
- V rámci POSIX má proces všechny zdroje (pokud nejsou sdílené) alokované pro vlastní použití (například file deskriptory)
- V rámcí standardní knihovny existuje funkce `system`, která spouští shell v novém procesu a blokuje rodičovský proces, dokud potomek neskončí
  - V Linuxu je tato funkce implementována, jako sekvence volání `fork(2)`, `execl(3)` a `waitpid(3)`
- Spuštění nového procesu `pid_t fork(void);`
  - Vytváří nový proces zkopírováním celé virtuální paměti rodičovského procesu.
  - Oba procesy, jak rodič tak potomek, pokračují ve vykonávání instrukcí na následujícím řádku po volání fork
  - Zda je proces rodič nebo potomek, lze zjistit pomocí návratové hodnoty fork

Procesy II.

- Rodina funkcí `exec(3)`
  - Spouští předaný program nahrazením kódu běžicího procesu kódem programu předaného funkci
  - Volání pouze nahrazuje výkonný kód procesu, ale nemění zdroje alokované u OS (např. tabulka file descriptorů)
- Funkce `popen(3)`
  - `FILE *popen(const char *command, const char *type);`
  - Narozdíl od funkce system spouští proces asynchroně, tedy bez blokování rodiče
  - Návratová hodnota funkce je rourou pro komunikaci s procesem
  - Může být pouze pro čtení nebo zápis, nikoliv obojí
  - Po skončení práce s procesem musí být nad nárvatovou hodnotou funkce `pclose`, nikoliv `fclose`

Vlákna

- Vlákna umožňují procesu vykonávat několik paralelních činností zároveň
- Zároveň ale všechna vlákna mezi sebou sdílí prostředky (například paměť)
- Což může přinášet problémy -> souběh, uváznutí a další
- Podpora pro vlákna je implementována v knihovně `pthread`
- Umožňuje vlákna spouštět, nebo plánovat jejich odstranění
- Zároveň obsahuje techniky pro synchronizaci vláken
  - Mutexy
  - Conditional variable
- Více o vláknech se dozvíte na předmětu v předmětech PB152 nebo PB153

Více o těchto funkcích v doplňkových materiálech od Šimona Totha

Make, CMake

Závěr

Závěr

- Nebojte se Operačních systémů ani jejich API
- Dávají vám do rukou silné zbraně při programování
- Používejte manuálové stránky
- A hlavně: Nepiště si vlastní nástroje pro sestavování projektů!

Děkuji za pozornost