POSIX, Make, CMake

Miroslav Jaroš

PB071 Úvod do nízkoúrovňového programovnání

24. 4. 2023

Obsah

POSIX

Motivace

Proč se zabývat operačním systémem?

- Standardní knihovna přináší programátorovi základní funkce 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, roury;
  - 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 zdroje viz tutorial.
    - `https://www.fi.muni.cz/pb071/man/os/windows.html#wsl`

Souborový systém I

- V UNIX-like systémech je souborový systém implementován jako n-ární strom s jedním kořenem.
- Všechny svazky (jiné disky, síťová úložište, ale i zařízení) jsou v něm adresovány - `mount`.
- Kořen souborového systému se jmenuje `/`.
- Soubory jsou implementovány pomocí `inode` (i-uzel).
  - `inode` je datová struktura popisující objekt existující v souborovém systému.
  - Váží se na něj všechny atributy souboru, jako velikost, oprávnění, datum vytvoření, modifikace atd.
  - `inode` nicméně neobsahuje informaci o tom, jak se daný soubor jmenuje, nebo kde se ve FS nachází.
- Vazba mezi jménem souboru a jeho `inode` je implementována na úrovni adresáře.

Souborový systém II

- Pro získání informací o souboru se používají funkce `stat`, `fstat`, `lstat`.
- `int stat(const char *path, struct stat *buf);`
- Při úspěchu je `struct stat` naplněna informacemi o souboru.
- Více viz `man 3 stat`.

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* Inode number */
    mode_t    st_mode;    /* File type and mode */
    nlink_t   st_nlink;   /* Number of hard links */
    uid_t     st_uid;     /* User ID of owner */
    gid_t     st_gid;     /* Group ID of owner */
    dev_t     st_rdev;    /* Device ID (if special file) */
    off_t     st_size;    /* Total size, in bytes */
    blksize_t st_blksize; /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;  /* Number of 512B blocks allocated */
}
```

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 souborových systému a jeho formát na souborovém systému 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 souborovém systému 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

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

Listing 1: Procházení FS, převzato a upraveno ze starší přednášky Šimona Totha

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 běžícího procesu kódem programu předaného funkci.
  - Volání pouze nahrazuje výkonný kód procesu, ale nemění zdroje alokované u OS (např. tabulka file descriptorů).
  - `int execve(const char *path, char *const argv[], char *const envp[]);`
- Funkce `popen(3)`
  - `FILE *popen(const char *command, const char *type);`
  - Narozdíl od funkce system spouští proces asynchronně, 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 návratovou hodnotou funkce zavoláno `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 v předmětech PB152 nebo PB153

Více o těchto funkcích například ve slidech PB065 [1].

Make, CMake

Překládání projektů

- Prozatím jste se v tomto předmětu setkali s jednoduchými prográmky s jednotkami překladových jednotek, které šlo přeložit jedním příkazem.
- `gcc -std=c99 -Wall -Wextra -pedantic -Werror *.c`
- Tento přístup lze aplikovat na jednoduché programy, které se přeloží okamžitě.
- Co ale s projekty, které obsahují tisíce řádků zdrojového kódu ve stovkách souborů.
  - I malá změna v jednom zdrojovém souboru znamená překompilování všech zdrojových souborů.
  - Což pro velké projekty trvá opravdu dlouho (linux cca 20 minut, gcc cca 1.5 hodiny).
- Jak reagovat na případy, kdy potřebujeme více spustitelných souborů?
  - Systémový démon a jeho CLI.
  - Klient server aplikace.
- A co závislosti? Kde najdeme knihovny, jak se jmenuji?

Překládání projektů II - Přístupy

- Přímý překlad není moudrý, je nepřenositelný a těžko reprodukovatelný.
- Trošku lepším řešením je napsat si nějakou vlastní automatizaci.
  - Typicky ve forme skriptu.
  - Tento přístup možná vyřeší problém překladu více binárek a opakovatelnosti sestavení, ale!
  - Překládat pouze změněné překladové jednotky je poměrně těžké implementovat.
  - Pro vyřešení vztahů mezi zdrojovými soubory, bychom museli implementovat alespoň základní parser jazyka pro jejich pochopení.
  - A nezapomínejme na chyby.
- Vůbec nejlepším řešením je použít nástroje k tomu určené.

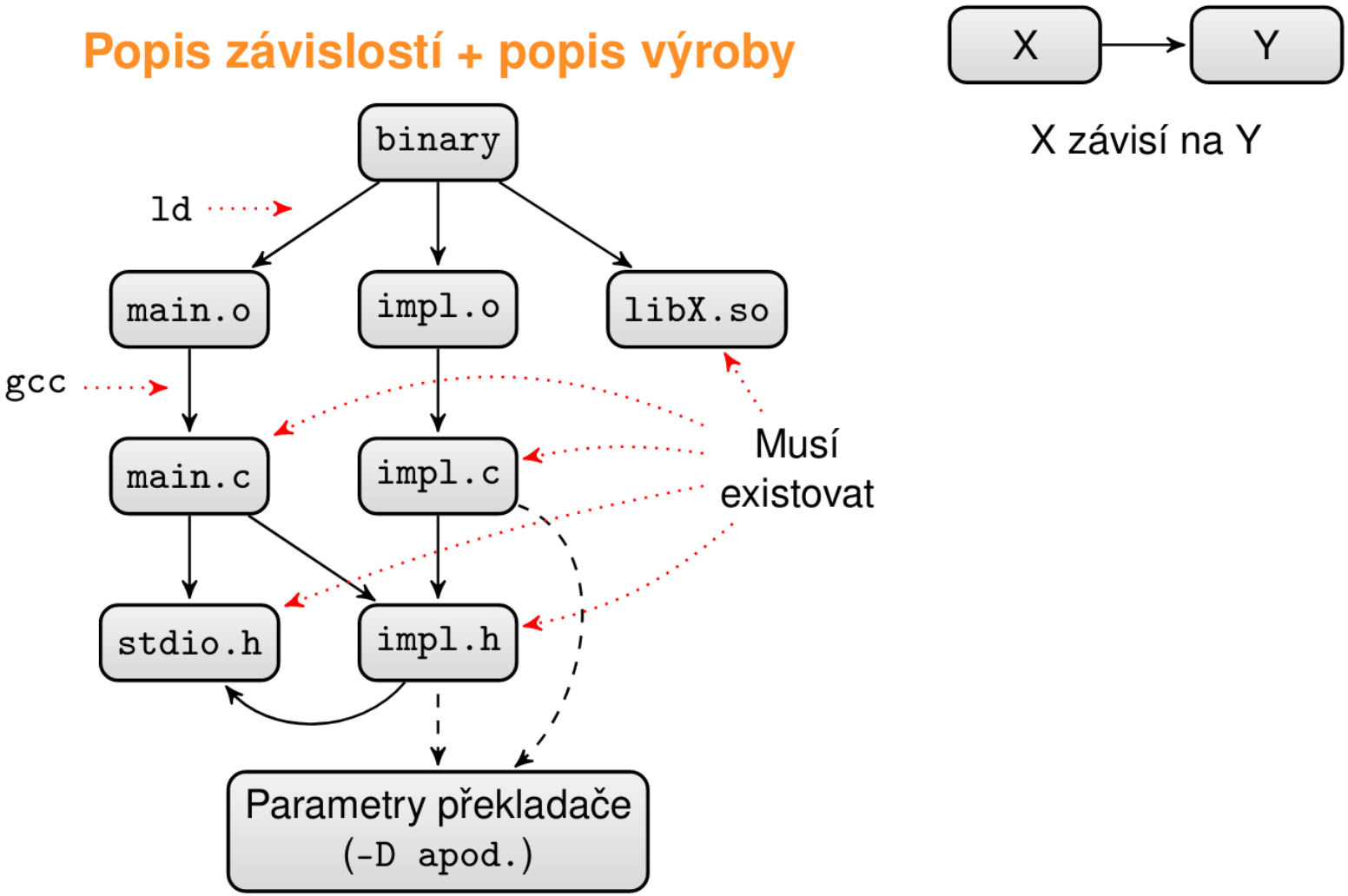
make I

- Nástroj na automatizaci sestavení.
- Popis sestavení projektu se ukládá do `Makefile`.
- Využívá rekurzivních pravidel pro vystavění stromu závislostí, který následně projde a nad každým cílem vykoná požadovanou akci
- Základním kamenem syntaxe je pravidlo `target: source1 source2`
  - `target` cíl, typicky se jedná o produkovaný soubor (binárka, nebo mezilehlá překladová jednotka), případně jde o `phony` cíl (viz dále).
  - `source1 source2` zdrojové soubory, může jít o soubory vytvořené předchozím pravidlem (cíle jiného pravidla), nebo zdrojové soubory překládaného programu.
- Akce se nad cílem vykoná, pokud:
  - cíl neexistuje a je potřeba jej vytvořit,
  - libovolný ze zdrojových souborů byl upraven po vytvoření cíle (datum modifikace zdrojového souboru je vyšší než datum modifikace cíle),

make II

  - cíl je `phony`, tedy neprodukuje žádný soubor.
- Spuštěním příkazu `make` se vyhledá soubor `Makefile` v lokálním adresáři.
- Pravidlo, které se začne vykonávat je první nalezené.
- Toto chování lze změnit explicitním zadáním očekávaného pravidla (*např.* `make clean`).
- Proměnné makefile lze nastavit na jiné hodnoty při spuštění `make all CC=clang`.
- Pro zrychlení překladu lze použít přepínač `-jN`, který překládá paralelně až `N` překladových jednotek.
- `make -j5 all`

make III



Obrázek: Popis závislostí v projektu, zdroj: Jiří Slabý, přednáška PB071 z roku 2017

Makefile

- Základ syntaxe jsou příkazy `target: source1 source2`.
- Pod touto deklarací jsou tabulátorem odsazené příkazy, které se mají provést.
- Lze v něm deklarovat proměnné
  - `VAR_NAME=value`
  - A následně je odkazovat jako `$(VAR_NAME)`
- Zobecnění pravidel (v GNU Make):
  - Speciální znak % je zástupným symbolem.
  - Lze jej použít pro konstrukci pravidel typu:
  - `%.o:` %c, které říká, že libovolné pravidlo, končící na .o, závisí na stejně se jmenujícím pravidle končícím na .c
  - Ke každému pravidlu potom náleží další speciální proměnné
    - `$@` je jméno cíle.
    - `$<` je první závislost.
    - `$^` jsou všechny závislosti

Makefile II

```
CC=gcc
CFLAGS=-std=c99 -Wall -Wextra -pedantic -Werror

all: myprog

myprog: main.o test.o
    $(CC) $(CFLAGS) $^ -o $@

%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $^

clean:
    rm -f *.o myprog

.PHONY: all clean
```

Generátory

- `make` je silný nástroj pro popis sestavení projektu, nicméně neřeší všechny problémy.
- Je nezbytné stále popisovat závislosti ručně. (Nebo si vypomáhat dodatečnými soubory třeba s pomocí `gcc -MM`)
- Má komplikovanou syntaxi a pro velké projekty již nedostačuje.
- Je méně přenositelný a nemá přímou podporu pro ověření vlastností kompilátoru nebo existence knihoven.
- Proto se pro větší projekty používají pokročilejší nástroje, které umí `Makefile` vygenerovat.

Generátory II – Autotools

- Jeden z nejrozšířenějších nástrojů v UNIXovém světe.
- Skládá se ze tří nástrojů:
  - Autoconf – vyhledávání externích závislostí,
  - Automake – popis překladu (podadresáře, layout projektu),
  - Libtool – podpora pro tvorbu knihoven.
- Výsledkem je několik servisních souborů a jeden spustitelný skript `configure`.
- Po jeho spuštění začne testování OS, zda a kde má potřebné závislosti.
- Při úspěchu je vygenerován `Makefile`.

Generátory III – CMake

- Se `CMake` se všichni minimálně od vidění známe :-)
- Základem konfigurace je jeden soubor `CMakeLists.txt`, který obsahuje popis projektu.
- Podobně jako u `Autotools` jeho spuštění generuje (krom obrovské spousty servisních souborů) `Makefile`.
- Ale neomezuje se pouze na ně, umí vygenerovat i projektové soubory pro `VisualStudio`, nebo `Ninja` build system.

```
cmake_minimum_required(VERSION 3.6)
project(hw04)

set(SOURCE_FILES test.cpp)
add_executable(hw04 ${SOURCE_FILES})
```

Listing 2: Ukázka CMakeLists.txt

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!

Kam dál?

- PV004 UNIX
- PV065 UNIX – programování a správa systému I
- PB173 Tematický vývoj aplikací v C/C++ (skupina zaměřená na POSIX)

Další zdroje

Jan Kasprzak.

PV065 Unix programování a správa systému I.

https://www.fi.muni.cz/~kas/pv065/pv065.pdf.

Makefile reference

https://www.gnu.org/software/make/manual/html_node/index.html

CMake tutorial

https://cmake.org/cmake/help/latest/guide/tutorial/index.html

Děkuji za pozornost