Základní meziprocesová komunikace v Linuxu II: Vstup a výstup programu 5

1 Vstup a výstup programu

1.1 Standardní vstup a výstup

V unixových operačních systémech aplikace komunikují s uživatelem pomocí standardního vstupu a výstupu.[1] Někdy se můžeme setkat se zjednodušeným pohledem, že programy vypisují svůj výstup na terminál (např. funkcí `printf`) nebo načítají vstup z terminálu (např. funkcí `scanf`). Takové zjednodušení je ale velmi zavádějící.

Je pravda, že implicitně je standardní vstup a výstup spojen s terminálem, ze kterého byl program spuštěn, ale operační systém (a jeho uživatelské rozhraní) nám umožňují předefinovat, co bude standardní vstupem a výstupem programu. Například následující příkazy ukazují přesměrování standardního výstupu programu `ls` do souboru `foo.dat` a přesměrování standardního vstupu programu `nl` ze souboru `foo.dat`.

```
ls > foo.dat
nl < foo.dat
```

Unixové operační systémy dále umožňují propojit vstupy a výstupy jednotlivých procesů a vytvořit tak „řetěz" procesů, které si mezi sebou předávájí data. Například následují příkaz ukazuje propojení tří příkazů. První z logovacího souboru vybere první sloupec (např. IP adresy), druhý příkaz tyto hodnoty seřadí a třetí spočítá počet unikátních hodnot.

```
cut -d" " -f1 access.log | sort | uniq -c
```

V tomto případě programy `sort` a `uniq` načítají vstupní data ze standardního vstupu a zapisují je (stejně i program `cut`) na standardní výstup. Shell, ve kterém tento příkaz spustíme, zajistí, že programy oddělené znakem `|` (svislá čára) budou po svém spuštění propojeny tak, že jejich standardní vstupy a výstupy budou provázany, aby si předávaly data.

Poznámka: Součástí unixové filozofie je, že máme spoustu malých jednoduchých programů, které dělají jednu věc (řazení hodnot, výběr unikátních prvků) a dělají ji dobře. Komplexnější funkcionalita pak vzniká kompozicí těchto menších programů právě propojením jejich vstupů a výstupů. Všimněme si podobnosti s běžným programováním (zejména funkcionálním), kdy máme jednoduché funkce a jejich skládáním

[1] Ve skutečnosti se jedná o dva výstupy – standardní výstup, kam se vypisují zpracovaná data, a standardní chybový výstup, kam se vypisují informace o chybách případně ladící informace.

získáváme složitější funkcionalitu. K tradiční unixové filozofii patří i to, že data jsou reprezentována v textové podobě, a proto máme v základu celou řadu nástrojů pro práci s textem, které umožňují filtrovat řádky textu, transformovat je apod.

1.2 Rozhraní OS pro práci se soubory

Se soubory v jazyce C obvykle pracujeme pomocí sady funkcí (`fopen`, `fread`, `fprintf`, ...). Tyto funkce jsou součástí standardní knihovny jazyka a lze je používat bez ohledu na to, na jakém operačním systému program poběží. Vedle toho operační systémy poskytují ještě rozhraní, které je specifické pro daný operační systém.[2] Unixové operační systémy nejsou výjimkou a poskytují specifické rozhraní pro práci se soubory,[3] které je podobné tomu ze standardní knihovny jazyka C, ale jsou tam určité rozdíly.

1.2.1 Otevření souboru

K otevření souboru slouží funkce deklarované v hlavičkovém souboru `fcntl.h`:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Funkce `open` otevře (případně vytvoří) soubor specifikovaný cestou `pathname` a při otevírání souboru se chová podle příznaků v argumentu `flags`. Minimálně bychom měli uvést v jakém režimu soubor otevíráme. To pro běžné datové soubory znamená použít buď příznak `O_RDONLY` (soubor je otevřen pouze pro čtení), `O_WRONLY` (soubor je otevřen pouze pro zápis) nebo `O_RDWR` (čtení i zápis).[4] Dále je možné specifikovat, zda ukazatel pozice v souboru ukazuje na konec souboru (`O_APPEND`), jestli obsah souboru má být odstraněn (`O_TRUNC`) nebo zda má být soubor vytvořen, neexistuje-li (`O_CREAT`).

Otevření souboru pro čtení ilustruje následující příklad:

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
        int fd = open("foo.txt", O_RDONLY);
        if (fd < 0) {
                printf("Unable to open file foo.txt\n");
                exit(1);
        }
        close(fd);
        return 0;
}
```

[2] To jsme mohli pozorovat například na cvičení v předchozím věnovaném *Windows API a základní práce s procesy ve Windows*.
[3] Viz cvičení z minulého semestru věnované *Systémovým voláním*.
[4] Tento přehled si nečiní ambice být úplný a podrobnější vysvětlení a popis dalších příznaků lze nalézt v dokumentaci.

Všimněme si návratové hodnoty. Funkce `open` vrací celé číslo, to se označuje jako *file descriptor* nebo *popisovač souboru* a identifikuje námi otevřený soubor. Pomocí tohoto čísla budeme se souborem pracovat a odkazovat na něj. V případě, že operace selže, vrací funkce `open` záporné číslo.

Úkol č. 1: Vyzkoušejte program tak, aby otevření souboru jednou selhalo a jednou neselhalo.

K ukončení práce se souborem (a uvolnění popisovače souboru) slouží funkce `close`, která je deklarovaná v hlavičkovém souboru `unistd.h`.

V případě, že by měl být soubor vytvořen, je nutné uvést třetí argument `mode`, který specifikuje oprávnění pro práci s nově vytvořeným souborem.[5] Tato oprávnění jsou definována jako konstanty ve tvaru `S_I[RWX](USR|GRP|OTH)`, kde `R`, `W`, `X` udávají oprávnění pro čtení, zápis, spuštění souboru, a `USR`, `GRP`, `OTH` určuje, jestli se dané opravnění vztahuje k vlastníkovi souboru, skupině nebo všem ostatním. Například příznak `S_IWUSR` značí, že do souboru může zapisovat jeho vlastník, `S_IROTH` značí, že všichni mohou číst soubor apod.

Otevření a vytvoření souboru, pokud neexistuje, pro zápis a čtení s tím, že se předpokládá zápis za konec souboru ukazuje následující volání:

```
int fd = open("foo.txt", O_CREAT | O_RDWR | O_APPEND, S_IRUSR | S_IWUSR);
```

Pokud bude soubor vytvořen, budou nastavena oprávnění tak, že pouze jeho vlastník bude moci do souboru zapisovat a číst z něj, viz oprávnění `S_IRUSR | S_IWUSR`.

1.2.2 Práce se souborem

K práci se souborem máme k dispozici mimo jiné funkce:

```
ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
```

Funkce `write` zapíše do souboru `fd` obsah bufferu `buf`, kde `count` udává počet bytů, které se mají zapsat. Analogicky funkce `read` načte do bufferu `count` bytů ze souboru `fd`. Obě tyto funkce vrací skutečný počet bytů, které se podařilo přečíst nebo zapsat.[6] Funkce `lseek` umožňuje přesunout aktuální ukazatel v souboru na námi zadaný offset.

Úkol č. 2: Do souboru `foo.txt` zapište libovolný řetězec. Ověřte, že jsou data zapsána na konec souboru.

Alternativně můžeme popisovač souboru pomocí funkce `FILE *fdopen(int fd, const char *mode)` převést na hodnotu typu `FILE *` a pracovat se souborem pomocí funkcí standardní knihovny jazyka C. Pozor, `mode` v tomto případě znamená způsob otevření souboru jako u funkce `fopen` a musí být kompatibilní se způsobem otevření souboru pomocí `open`. K uzavření souboru se používá funkce `fclose`, která obstará i ukončení práce s popisovačem souboru.

Úkol č. 3: Převeďte popisovač souboru `foo.txt` na hodnotu typu `FILE *` a zapište do souboru řetězec pomocí funkce `fprintf`.

[5] Pokud tato opravnění neuvedeme, vezmou se libovolná (náhodná) data ze zásobníku.

[6] Návratová hodnota může být menší než `count`, např. pokud v souboru již nejsou žádná data, která by bylo možné přečíst, protože jsme narazili na konec souboru.

2 Přesměrování vstupů a výstupů

2.1 Práce se soubory

Po spuštění má každý proces k dispozici tři popisovače – *standardní vstup* (0), *standardní výstup* (1), *standardní chybový výstup* (2).

Úkol č. 4: Pomocí funkce `write` zapište vhodný řetězec na standardní výstup.

Tyto popisovače jsou obvykle spojeny s terminálem, ze kterého byl program spuštěn a ze kterého čte vstup nebo na který zapisuje výstup. Jak bylo zmíněno v úvodní kapitole, toto není výhradní způsob práce se standardními vstupy a výstupy. Vstup nebo výstup můžeme přesměrovat z/do souboru, např. jako u příkazu `ls > foo.txt`.

Ke změně popisovače souboru slouží funkce `dup2` deklarována v hlavičkovém souboru `unistd.h`:

```
int dup2(int oldfd, int newfd);
```

Pokud tato funkce proběhne úspěšně, popisovač souboru `newfd` bude ukazovat na soubor, který je dán popisovačem `oldlfd`.

Praktické použití ukazuje následující příklad:

```
1  int fd = open("foo.txt", O_CREAT | O_RDWR | O_APPEND, S_IRUSR | S_IWUSR);
2  if (fd < 0) {
3          printf("Unable to open file foo.txt\n");
4          exit(1);
5  }
6  if (dup2(fd, 1) < 0) {
7          printf("Unable to redirect std. output\n");
8          exit(1);
9  }
10 printf("Hello world\n");
11 close(fd);
```

Na řádku č. 6 funkce `dup2` zajistí, že popisovač souboru s číslem 1 (tj. standardní výstup) bude ukazovat na soubor daný popisovačem `fd`, tj. na soubor `foo.txt`. To znamená, že jakýkoliv další zápis na standardní výstup (viz volání `printf`) bude proveden do souboru `foo.txt`.

Úkol č. 5: Ověřte, že opravdu jakýkoliv další výstup je přesměrován do souboru `foo.txt`. Volání funkce `printf` nahraďte spuštěním příkazu `ls` pomocí volání `exec`.

Úkol č. 6: Upravte řešení předchozího úkolu tak, aby příkaz `ls` spustil potomek vytvořený pomocí systémového volání `fork`. Zajistěte, že přesměrování standardního výstupu je provedeno až v potomkovi a rodič může zapisovat na svůj standardní výstup, tj. výpis se provede na terminál.

2.2 Roury

2.2.1 Vytvoření roury

K propojení dvou programů v unixových operačních systémech se používají roury. Rouru vytvoříme pomocí funkce (resp. systémového volání) `pipe`:

```
int pipe(int fildes[2]);
```

Funkce `pipe` jako svůj argument akceptuje pole popisovačů `fildes` o dvou prvcích. Pokud funkce proběhne v pořádku, uloží se do pole `fildes` dva popisovače souborů. Popisovač `fildes[1]` slouží k zápisu do roury (zapisovací konec roury), popisovač `fildes[0]` slouží k čtení dat z roury (čtecí konec roury). Zapíšeme-li do „souboru", který je určen popisovačem `fildes[1]`, jsou tato data uložena do bufferu (v jádře operačního systému). Opačně, při čtení „souboru", který je dán popisovačem `fildes[0]`, jsou data načítána z tohoto bufferu v jádře OS.

Důležité je, že v situaci, kdy vznikne potomek pomocí volání `fork`, mají rodič i potomek přístup ke stejné rouře, kterou spolu mohou komunikavat, jak ukazuje následující příklad.

```
#define BUF_SIZE        1024
int fds[2];
if (pipe(fds) < 0) {
        printf("Unable to create a pipe\n");
        exit(1);
}
pid_t pid = fork();
if (pid < 0) {
        printf("Unable to create child process\n");
        exit(1);
} else if (pid == 0) {
        /* potomek: cte data z roury*/
        close(fds[1]); // uzavren zapisovaci konec roury
        char buf[BUF_SIZE];
        // precteni dat
        ssize_t cnt = read(fds[0], buf, BUF_SIZE);
        close(fds[0]); // uzavren cteci konec roury
        buf[cnt] = '\0';
        if (cnt >= 0) printf("Received:\n%s\n", buf);
} else {
        /* rodic zapisuje do roury */
        close(fds[0]); // uzavren cteci konec roury
        write(fds[1], "hello\nworld\n", 12); // zapise data
        close(fds[1]); // uzavre zapisovaci konec
}
```

Na řádcích 2 až 6 je vytvořený objekt roury. Na řádku č. 6 je vytvořen nový proces, kdy řádky 11 až 19 představují kód, který provádí potomek, a řádky 22 až 24 představují kód provedený rodičovským procesem. V tomto ukázkovém příkladu rodičovský proces zapisuje do roury a potomek data z roury načítá, jinými slovy, rodič posílá data potomkovi.

Rodičovský proces nejdříve uzavře nepotřebný (čtecí) konec roury (řádek 22) a do zapisovacího konce roury zapíše data (řádek 23) a zavře i tento konec roury (řádek 24).

Potomek postupuje analogicky. Zavře nepotřebný (zapisovací) konec roury (řádek 13), přečte data z roury (řádek 16), a zavře i čtecí konec roury (řádek 17). Zbylé řádky v kódu potomka slouží k vypsání předaných dat.

2.2.2 Propojení vstupů a výstup rourou

Protože oba konce roury jsou popisovače souboru, můžeme je přesměrovat na standardní vstup programu nebo naopak můžeme přesměrovat standardní výstup do roury.

Přesměrování roury na standardní vstup programu dosáhneme pomocí:

```
dup2(fds[0], 0);
```

Po provedení této funkce bude popisovač standardního vstupu (tj. 0), ukazovat na čtecí konec roury. Jinými slovy, pokud bychom použili funkci (`scanf`), bude číst data přímo z roury.

Nemusíme zůstávat u čtení dat ze souboru. Můžeme pomocí systémového volání `exec` spustit program, jehož standardní vstup bude nastaven na čtecí konec roury. Například následovně.

```
dup2(fds[0], 0);
execlp("nl", "nl", NULL);
```

Úkol č. 7: Upravte ukázkový kód tak, aby potomek přesměroval čtecí konec roury na svůj standardní vstup a následně spustil program `nl`.

Analogicky můžeme postupovat u zapisovacího konce roury a rodičovského procesu. Standardní výstup programu přesměrujeme do roury.

```
dup2(fds[1], 1);
```

V tento okamžik, cokoliv je zapsáno na standardní vstup (např. funkcí `printf`), je zapsáno do roury. Opět můžeme spustit program, jehož standardní výstup bude směrován do roury.

```
dup2(fds[1], 1);
execlp("ls", "ls", NULL);
```

Úkol č. 8: Upravte ukázkový kód tak, aby rodič přesměroval svůj standardní výstup do roury a následně spustil program `ls`.

Úkol č. 9: Vytvořte program, kde rodič jako potomka spustí program `ls`. A výstup tohoto programu bude rourou směrován k rodiči, který výstup programu `ls` bude číst a bude na svůj standardní výstup vypisovat název souboru spolu s pořadovým číslem. Např.

```
bar.txt (1)
baz.txt (2)
foo.txt (3)
```

2.3 Pojmenované roury

Roury, které jsme si ukázali v předchozí kapitole, jsou omezeny na procesy, které jsou příbuzné, je mezi nimi vztah rodič-potomek nebo se jedná o potomky stejného rodiče. Pokud chceme propojit nepříbuzné procesy, můžeme použít pojmenované roury, což je speciální typ souboru, který se chová jako roura, tj. je možné do něj zapisovat, a následně z něj data číst v pořadí tak, jak do něj byla zapsána. Proto se tento typ souborů označuje jako FIFO.

Pojmenovanou rouru můžeme vytvořit příkazem `mkfifo`, případně funkcí stejného jména.

Následující příkaz vytvoří speciální soubor `foo.fifo`, který se chová jako roura.

```
mkfifo foo.foo
```

Můžeme si to vyzkoušet následovně.

```
echo -e "abc\ncde\nefg" > foo.fifo
```

Tento příkaz zapíše do souboru tři řádky, které můžeme následně přečíst vhodným příkazem, například s pomocí `cat`, `nl`, ...

```
cat foo.fifo
```

Poznámka: Roury jsou určeny k jednosměrné komunikaci právě dvou procesů. Neexistují žádné prostředky, které by znemnožnily jiné použití, např. pokud by někdo chtěl, aby do roury zapisovalo více procesů, není tomu bráněno, avšak chování je v takovém případě nedefinováno. Pravděpodobně dojde k chybě souběhu, a proto by se mělo s rourami pracovat jen očekávaným způsobem.