Meziprocesová komunikace (IPC)

Tématicky zaměřený vývoj aplikací v jazyce C
skupina Systémové programování – Linux

Martin Dražar, Martin Husák, Petr Velan

Fakulta informatiky
Masarykova univerzita
drasar|husak|velan@ics.muni.cz

Brno, 26. října 2015

Signály

principy a práce se signály

Signály

```
#include <signal.h>
```

- princip přerušení – událostmi řízený program
- příchod signálu → přerušení činnosti → obsloužení signálu → návrat k předchozí práci (nebo také ne)
- některé signály lze ignorovat, některé blokovat, pro většinu lze měnit reakce na ně – výjimkami jsou `SIGKILL` a `SIGSTOP`
- obyčejné signály vs. real-time (spolehlivé) signály
- nikdy nepoužívejte přímo hodnoty, u real-time signálů `SIGRTMIN+n`
- seznam signálů: `man 7 signal`, např.:

```
SIGTERM         "Termination" - signál ukončení
SIGKILL         "Kill" - signál pro nepodmíněné ukončení
SIGSEGV         Odkaz na nepřípustnou adresu v paměti
SIGUSR1         Signál definovaný uživatelem
...
```

Posílání signálů

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- signály lze posílat nejen konkrétnímu procesu, ale i skupině procesů, nebo všem běžícím procesům
- zaslání signálu je omezeno oprávněním uživatele

```
int raise(int sig);
```

- signály může proces posílat i sám sobě – mechanismus výjimek

```
int sigqueue(pid_t pid, int sig, const union sigval value);
```

- pro real-time signály indikuje, zda se podařilo vložit signál do fronty
- navíc u všech signálů umožňuje zaslat procesu i data (int nebo ukazatel)

Reakce na signály

```
#include <signal.h>
sighandler_t signal(int signum, sighandler_t handler);
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
```

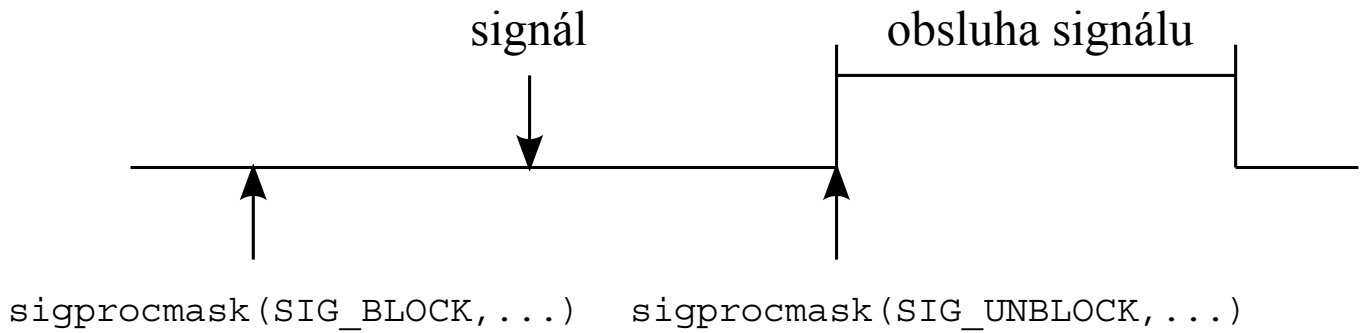
- funkci `signal()` používejte jen pro nastavení handleru na `SIG_IGN` nebo `SIG_DFL`
- pro vlastní handlery používejte výhradně funkci `sigaction()`
- handler musí být co nejjednodušší
- pokud už měníte globální proměnnou, deklarujte ji jako `volatile`
- i handler může být přerušen signálem – pokud je to nutné, blokujte signály (viz. dále)

Příklad – sigaction()

```
volatile int done = 0;

void my_handler(int sig) {
    done = 1;
}

int main () {
    struct sigaction action; /* action structure for specific signal */

    /* establish the signal handler */
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    action.sa_handler = my_handler;
    sigaction(SIGTERM, &action, NULL);
    sigaction(SIGINT, &action, NULL);

    while (!done) { /* do some work */
        sleep(1);
    }

    printf("cleaning up\n"); /* close files, free memory, write output, ... */
    return 0;
}
```

Blokování signálů

Umožňuje zajistit přijetí signálu až v určitý okamžik (odložení příjmu signálu).



SIGKILL a SIGSTOP nelze blokovat.

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigpending(sigset_t *set);
```

Operace s maskou signálů viz. `man 3 sigsetops`

Synchronní doručení signálů

- Pokud je třeba signály používat intenzivněji, stojí za zvážení jejich synchronní použití.
- Funkce `sigwait()`, `sigwaitinfo()`, `sigsuspend()`
- Pro skutečně synchronní chování, by signály, na které se čeká, měly být nejříve vymaskovány.

Poznámky k signálům

- Signály jsou drahé (asynchronní přerušení) $\rightarrow$ přemýšlejte zda je signál nejvhodnějším řešením.
- Kvůli optimalizaci kompilátoru deklarujte globální proměnné jako `volatile`.
- V obslužné rutině provádějte jen to nejnutnější (žádné zapisování do souborů, alokace paměti apod.).
- Pozor na `errno` – signály jsou asynchronní!.
- I obslužná rutina může být přerušena signálem.

úkol

- Napište program, který po doručení (konkrétního, výběr je na vás) signálu zapíše do syslogu informaci o zpracování signálu

Roury

anonymní a pojmenované roury

Roury

```
$ ls | less
```

- někdy se setkáte i s pojmem *datová kolona*
- **jednosměrný** proud bajtů mezi dvěma konci roury
- roura je dvojice file deskriptorů (`int pipefd[2]`) – `pipefd[0]` pro čtení a `pipefd[1]` pro zápis
- velikost roury je omezená (65536 bajtů)
- nepojmenované roury jsou určeny pouze pro příbuzné procesy

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

Roury

```
$ ls | less
```

- někdy se setkáte i s pojmem *datová kolona*
- **jednosměrný** proud bajtů mezi dvěma konci roury
- roura je dvojice file deskriptorů (`int pipefd[2]`) – `pipefd[0]` pro čtení a `pipefd[1]` pro zápis
- velikost roury je omezená (65536 bajtů)
- nepojmenované roury jsou určeny pouze pro příbuzné procesy

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

- často používáno v kombinaci s funkcí `dup2()`

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

Roury

```
$ ls | less
```

- někdy se setkáte i s pojmem *datová kolona*
- **jednosměrný** proud bajtů mezi dvěma konci roury
- roura je dvojice file deskriptorů (`int pipefd[2]`) – `pipefd[0]` pro čtení a `pipefd[1]` pro zápis
- velikost roury je omezená (65536 bajtů)
- nepojmenované roury jsou určeny pouze pro příbuzné procesy

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

- často používáno v kombinaci s funkcí `dup2()`

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

- zjednodušené použití nepojmenovaných rour – funkce `popen()` a `pclose()`, cílový proces je spouštěn pomocí shellu → značná režie navíc

Příklad – pipe() I

```
int main(int argc, char *argv[]) {
    int pfd[2];                 /* pipe */
    pid_t child;          /* child's PID */
    char buf;             /* char buffer */

    if (pipe(pfd) == -1) { perror("creating pipe failed"); return 1; }

    child = fork();
    if (child == -1) { perror("fork() failed"); return 1;
    } else if (child == 0) { /* child process - reader */
        close(pfd[1]);        /* Close unused write end */
        while (read(pfd[0], &buf, 1) > 0) {
            write(STDOUT_FILENO, &buf, 1);
        }
        write(STDOUT_FILENO, "\n", 1);
        close(pfd[0]);
    } else {              /* Parent writes string to pipe */
        close(pfd[0]);         /* Close unused read end */
        write(pfd[1], "Ahoj!", strlen("Ahoj!"));
        close(pfd[1]);           /* Reader will see EOF */
        wait(NULL);                   /* Wait for child */
    }
    return 0;
}
```

úkol

- Napodobte chování shellu při řetězení aplikací pomocí datových kolon (|)
- Aplikace dostane jako parametry 2 řetezce představující příkazy ke spuštění – nezapoměňte, že součástí příkazu mohou být parametry spouštěného programu
- Vaše aplikace pak zařídí spuštění zadaných příkazů a propojení standardního výstupu první aplikace se standardním vstupem druhé.
- Nápověda: dup2()

Pojmenované roury (FIFO)

- nepojmenované roury jsou určeny pouze pro příbuzné procesy
- co když ale potřebujeme propojit nepříbuzné procesy?

Pojmenované roury (FIFO)

- nepojmenované roury jsou určeny pouze pro příbuzné procesy
- co když ale potřebujeme propojit nepříbuzné procesy?

FIFO alias pojmenovaná roura

- součást souborového systému – pouze jako referenční bod pro přístup procesů, do souborového systému se nic nezapisuje
- je možné nastavit přístupová práva jako kterémukoliv jinému souboru
- pro komunikaci je nutné, aby oba konce roury byly otevřené
- s rourou se pak pracuje jako se souborem – `open()`, `read()`, `write()`, `close()`, `unlink()`

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

úkol

- stejnou úlohu jako pro nepojmenované roury implementujte pomocí `mkfifo()`

D-Bus

jemný úvod D-Busu

D-Bus

- http://www.freedesktop.org/wiki/Software/dbus
- systém pro komunikaci procesů pomocí rozesílání zpráv
- základem knihovna `libdbus`, pro různé jazyky/frameworky (Python, Qt, Glib, ...) existují obalové knihovny
- dva typy zpráv
  - volání metod (RPC) - zprávy s definovaným adresátem, obvykle vás zajímá i odpověď na zprávu
  - signály - "veřejné"oznámení bez adresáta
- součástí zprávy může být cokoli – čísla, pole, řetězce, struktury, ...
- další vychytávky v podobě startování služeb na vyžádání nebo vlastního systému oprávnění (obdoba UNIXových uživatelských práv)

Další možnost

další možnosti IPC, kterým se budeme věnovat v příštích cvičeních

Další možnosti IPC

sdílená paměť

- dva či více procesů přistupuje ke sdílenému paměťovému místu
- žádné kopírování dat, synchronizaci přístupu zajišťují procesy
- jeden z procesů paměť alokuje, ostatní se k alokovanému segmentu připojují

Další možnosti IPC

sdílená paměť

- dva či více procesů přistupuje ke sdílenému paměťovému místu
- žádné kopírování dat, synchronizaci přístupu zajišťují procesy
- jeden z procesů paměť alokuje, ostatní se k alokovanému segmentu připojují

sockety – síťová rozhraní

- obousměrný komunikační kanál
- komunikace mezi procesy běžícími na stejném počítači, ale i pro komunikaci s procesem na jiném počítači
- spojovaná komunikace (model klient-server) vs. datagramová komunikace

Závěr

domácí úkoly a zdroje

Domácí úkol I.

World I.

- přidejte přepínač *--daemonize* / *-d*, který spustí world jako daemona
- bude reagovat na signály SIGQUIT, SIGINT, SIGTERM, korektně se ukončí
- při ukončení napřed pošle SIGTERM všem tankům, vypíše stav hry, uzavře otevřené zdroje (roury), atd.
- na SIGUSR1 se restartuje (jako ukončení a spuštění se stejnými parametry)
- v daemon režimu bude psát výstup do syslogu
- mapu bude vykreslovat do pojmenované roury (parametr *-p* / *--pipe*), formát dále

Domácí úkol II.

World II.

- ve /var/run/world.pid bude po spuštění PID procesu.
- world bude kontrolovat zda už je spuštěn, smí běžet pouze jednou
- world komunikuje s tanky pomocí nepojmenované roury, více u tanků
- zrušte přepínač a funkci total-respawn, svět se bude ukončovat signálem
- přidejte přepínač *--round-time N* kde N je čas trvání jednoho kola v milisekundách
- přidejte přepínače *--green-tank path* a *--red-tank path* kde path je cesta k binárce tanku. Díky tomu budeme spouštět různé implementace proti sobě.

Domácí úkol III.

Formát informací o hrací ploše pro přenos rourou

- X,Y,(0|r|g),(0|r|g),...
- X: Velikost hrací plochy v ose x
- Y: Velikost hrací plochy v ose y
- (0|r|g)
  - 0: prázdné pole
  - r: tank červeného týmu
  - g: tank zeleného týmu
- celkem x * y hodnot
- hodnoty postupují podle indexu: x1y1, x2y1, ..., xXy1, x1y2, ..., xXyY

Domácí úkol IV.

Worldclient

- nový program
- bude číst informace o hrací ploše a vykreslovat je (místo worldu)
- bude reagovat na stisk kláves
  - x ukončí world (pošle SIGINT)
  - q ukončí worldclient
  - r restartuje world (SIGUSR1)
- PID worldu zjistí ze /var/run/world.pid
- Přepínače
  - *-p | --pipe*
  - *-h | --help*

Domácí úkol V.

Tank

- už neumírá sám, odstranit přepínače a náhodné ukončení
- tank může buď střílet, nebo se pohybovat v jednom ze čtyř směrů
- pro provedení akce zapíše do roury

  mu, md, ml, mr (move up, down, left, right)
  fu, fd, fl, fr (fire up, down, left, right)
- akci provede až po přijetí signálu SIGUSR2
- bude reagovat na signály SIGQUIT, SIGINT, SIGTERM, korektně se ukončí
- vymyslete jak bude tank volit akce
- Tank dostane přepínač *--area-size*, aby věděl kde se smí pohybovat

Domácí úkol VI.

Průběh kola:

- Svět pošle signál všem tankům, pak přečte z rour jejich akci.
- Vyhodnocení kola probíhá následovně:
  - Srážka tanků končí zničením obou
  - Pokud tank narazí do okraje mapy, je zničen
  - Pokud se v cestě střely tanku nachází jiný, je zničen
  - Vše v kole probíhá zároveň (tanky se mohou sestřelit navzájem)
- Jestliže je tank zničen, dostane SIGTERM od světa
- Pokud jsou všechny tanky zničeny, svět čeká na ukončení nebo restart.
- Svět loguje zničení každého tanku (PID a pozici útočníka pokud existuje, PID a pozici zničeného tanku, číslo kola kdy se tak stalo).

Zdroje

www.tldp.org/LDP/lpg/node7.html

davmac.org/davpage/linux/rtsignals.html

signály

- www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_21.html
- www.win.tue.nl/~aeb/linux/lk/lk-5.html

roury

- www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_14.html