

Meziprocesová komunikace (IPC)

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

Jiří Novosad

Fakulta informatiky
Masarykova univerzita
novosad@fi.muni.cz

Brno, 15. 10. 2014

Signály

principy a práce se signály

Signály

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

- princip přerušení – událostmi řízený program
- příchod signálu → přerušeni činnosti → obslužení 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 – SIGRTMAX
- seznam signálů: `man 7 signal`, např.:

SIGTERM	"Termination" - signál ukončení
SIGKILL	"Kill" - signál pro nepodmíněné ukončení
SIGSEGV	Odkaz na nepřipustnou 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 handlers používejte výhradně funkci `sigaction()`
- handler musí být co nejjednodušší
- pokud už měníte globální proměnnou, deklarujte ji jako `volatile`, ideálně typu `sig_atomic_t`
- i handler může být přerušeno signálem – pokud je to nutné, blokujte signály (viz. dále)

Příklad – sigaction()

```
volatile sig_atomic_t done = 0;

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

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

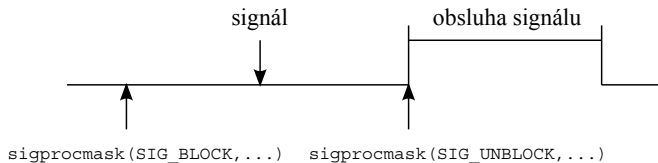
    /* establish the signal handler */
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    action.sa_handler = my_handler;
    sigaction(SIGTERM, &action, NULL);
    sigaction(SIGINT, &action, NULL);

    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 nejdříve vymaskovány – zabrání se doručení signálu před dokončením volání funkce.

Poznámky k signálům

- Signály jsou drahé (asynchronní přerušení) → 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.
- Async-signal-safe functions (`man 7 signal`)
- C99: jen nastavení globální proměnné typu `sig_atomic_t`
- 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 (+ bezpečnost!)

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;
}
```

Příklad – pipe() II

```
int main(int argc, char *argv[]) {
    ...
    if (pipe(pfd) == -1) { perror("pipe() failed"); return 1; }
    child = fork();
    if (child == -1) { perror("fork() failed"); return 1; }
    } else if (child == 0) { /* child process - sender */
        close(pfd[0]);      /* Close unused read end */
        if (dup2(pfd[1], STDOUT_FILENO) == -1) {
            perror("creating pipe failed"); return 1;
        }
        execlp("ls", "ls", "-l", NULL);
        perror("exec failed"); return 1;
    } else { /* Parent - reader */
        close(pfd[1]);      /* Close unused write end */
        while (read(pfd[0], &buf, 1) > 0)
            write(my_file_descriptor, &buf, 1);
        write(my_file_descriptor, "\n", 1);
        close(pfd[0]);
        wait(NULL);        /* Wait for child */
    }
    return 0;
}
```


úkol

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

```
$ echo -e "Ahoj\nPomoc"  
| ./myshell tr '[a-z]' '[A-Z]' : sort -r  
POMOC  
AHOJ
```

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()`

Další možnost

další možnosti IPC, kterým se budeme věnovat v příštích cvičeníh

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

sockets – 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

- vytvořte systémového daemona¹, který přijímá signály
- prvním parametrem daemona je název souboru, do kterého bude zapisovat události ve tvaru `<time>: <signal_name>`
- daemon loguje alespoň signály
`SIG{INT,QUIT,TERM,USR1,USR2,CONT}`
- dalším parametrem daemona je interval (v sekundách), ve kterém daemon zapisuje vlastní statistiku (počet obdržených signálů) do syslogu
- po obdržení signálu `SIGTERM` se daemon korektně ukončí (uzavření souboru, `closelog()`, ...)
- nápověda: `alarm(2)`, `strsignal(3)`

¹využijte funkci `daemon()`

Zdroje

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
- "Real time" signals on Linux

roury

- www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_14.html
- www.tldp.org/LDP/lpg/node7.html