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, 21. 10. 2016

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` – `SIGRTMAX`
- 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
- UID/EUID $\rightarrow$ UID/Saved-UID

```
unsigned int alarm(unsigned int seconds);
void abort(void);
int raise(int sig);
int pause(void);
```

- 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 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);
```

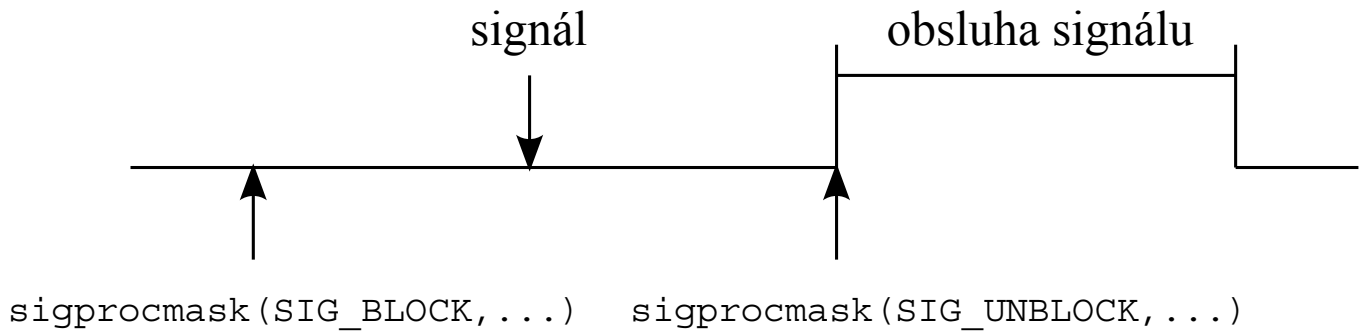
- funkci `signal()` používejte jen pro nastavení handleru na `SIG_IGN` nebo `SIG_DFL`
- pro vlastní handlery používejte výhradně funkci `sigaction()`
- handler musí být co nejjednodušší
- pokud už měníte globální proměnnou, deklarujte ji jako `volatile`, ideálně typu `sig_atomic_t`
- i handler může být přerušen 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 */
        pause();
    }

    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 sigsuspend(const sigset_t *mask);
```

Operace s maskou signálů viz. `man 3 sigsetops`

Synchronní doručení signálů

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
int sigpending(sigset_t *set);
```

- Pokud je třeba signály používat intenzivněji, stojí za zvážení jejich synchronní použití.
- Pro skutečně synchronní chování by signály, na které se čeká, měly být nejříve vymaskovány – zabrání se doručení signálu před dokončením volání funkce.

Poznámky k signálům

- Signály jsou drahé (asynchronní přerušení) → přemýšlejte, zda je signál nejvhodnějším řešením.
- Kvůli zabránění nechtěné optimalizace kompilátorem 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 v cyklu zapisuje do Syslogu po doručení konkrétního signálu informaci o jeho přijetí.
- Výběr signálu je na vás, ale doporučuji `SIGINT`.
- Nechci „busy-waiting“ – cyklení naprázdno.
- Zkuste si představit na různých místech programu volání `sleep(1)` – nemůže se signál ztratit?

Roury

anonymní a pojmenované roury

Roury

```
$ ls | sort
$ ls | sort | head | vim -
```

- **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, atomicky lze zapsat `PIPE_BUF` bytů (`fpathconf(3), _PC_PIPE_BUF`) – Posix 512 / Linux 4096
- nepojmenované roury jsou určeny pouze pro příbuzné procesy

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

Roury

```
$ ls | sort
$ ls | sort | head | vim -
```

- **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, atomicky lze zapsat `PIPE_BUF` bytů (`fpathconf(3), _PC_PIPE_BUF`) – Posix 512 / Linux 4096
- 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 | sort
$ ls | sort | head | vim -
```

- **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, atomicky lze zapsat `PIPE_BUF` bytů (`fpathconf(3), _PC_PIPE_BUF`) – Posix 512 / Linux 4096
- nepojmenované roury jsou určeny pouze pro příbuzné procesy

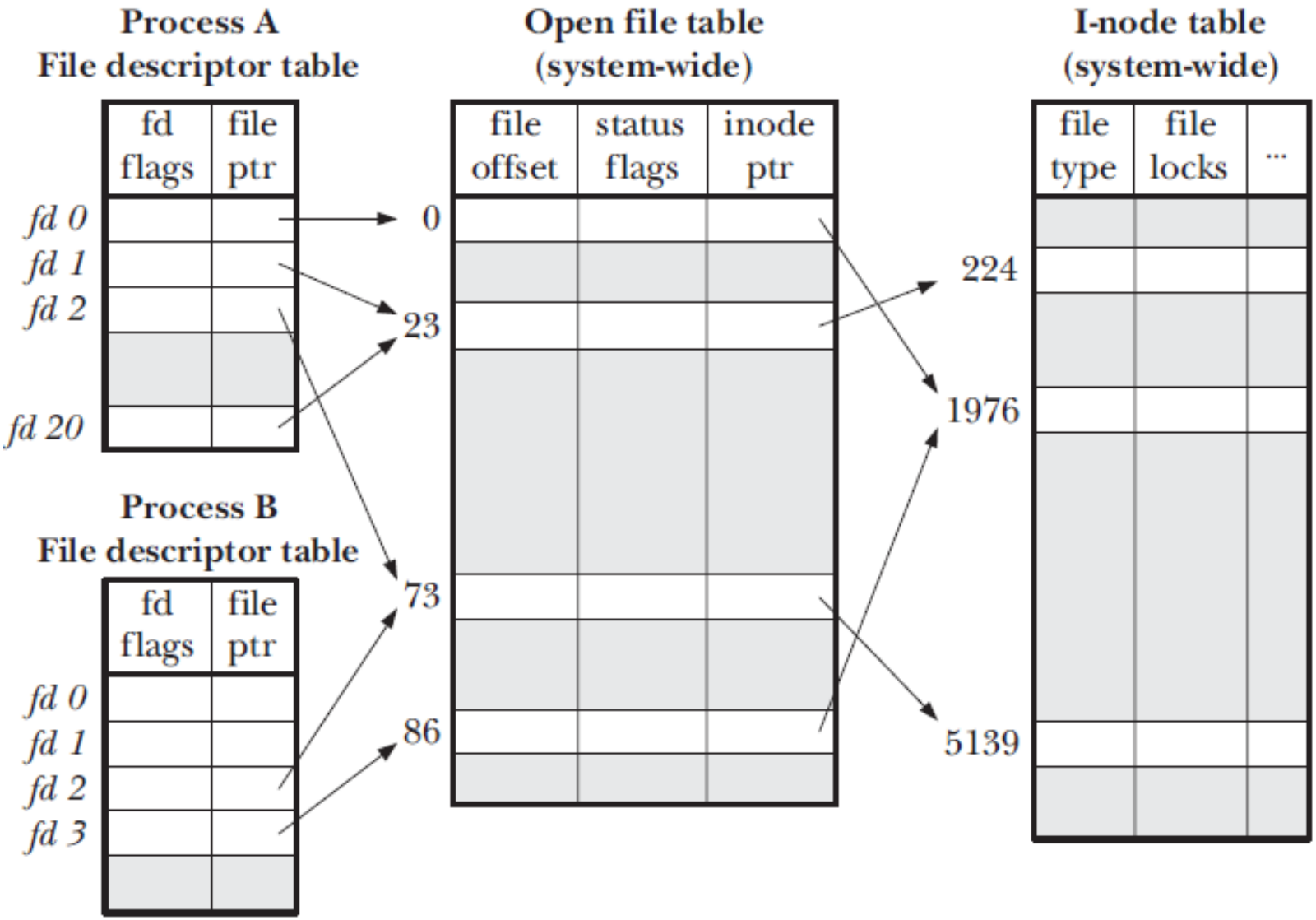
```
#include <unistd.h>
int pipe(int pipefd[2]);
```

- 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!)

Odbočka – soubory



Příklad – pipe() I

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

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

    child_pid = fork();
    if (child_pid == -1) { perror("fork() failed"); return 1;
    } else if (child_pid == 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_pid = fork();
    if (child_pid == -1) { perror("fork() failed"); return 1;
    } else if (child_pid == 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 (|)
- Váš program dostane jako parametry dva příkazy ke spuštění – nezapomeňte, že součástí příkazu mohou být parametry spouštěného programu
- Tyto dva příkazy budou odděleny pomocí dvojtečky (:)
- Program pak zařídí spuštění zadaných příkazů a propojení standardního výstupu první aplikace se standardním vstupem druhé pomocí volání `dup2(2)`.
- Příklad spuštění:

```
$ echo -ne "Ahoj\nSvete\n" | ./myshell tr '[a-z]' '[A-Z]' : sort -r
SVETE
AHOJ
```

- Stejné chování, jako v shellu, zkuste např. `./myshell cat : head -n 0` nebo `./myshell echo hello : cat`

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()`
- `open(2)`, `man sys_stat.h`

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

- vytvořte systémového daemona[1], který přijímá signály
- prvním parametrem daemona je název souboru, do kterého bude zapisovat události ve tvaru `<time>: <signal_description>`
- co událost, to řádek
- 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)`, `O_APPEND`, `fflush(3)`, `strftime(3)`

[1] využijte funkci daemon()

Zdroje

signály

- www.win.tue.nl/∼aeb/linux/lk/lk-5.html
- "Real time" signals on Linux

roury

- www.tldp.org/LDP/lpg/node9.html
- www.tldp.org/LDP/lpg/node15.html