KAPITOLA 7

KOMUNIKÁCIE MEDZI PROCESMI

Procesy, vykonávané v operačnom systéme majú oddelené adresné priestory, ktoré sú chránené pred prístupom iných procesov. V prípadoch, keď je potrebná komunikácia medzi procesmi, operačný systém poskytuje prostriedky pre tento účel. V Linux-e k týmto prostriedkom patria rúry, fronty správ, zdieľaná pamäť, semafory a signály. Zdieľaná pamäť, fronty správ a semafory sa v literatúre označujú pod spoločným anglickým názvom Interprocess Communication - IPC. Treba poznamenať, že semafory a signály nie sú typickými komunikačnými prostriedkami, pretože neprenášajú dáta.

7.1 Rúry

7.1.1 Systémové volanie `pipe()`

Rúra (pipe) je komunikačný prostriedok, pomocou ktorého sa výstup z jedného procesu pripája k vstupu druhému. V kapitole o príkazovom jazyku `bash` bolo popísané použitie rúr na úrovni príkazov (pomocou zvislej čiary). Napríklad:

```
ls  –l | less
```

Komunikácia pomocou rúry je **jednosmerná** a komunikačný kanál môže byť vytvorený **len medzi príbuznými procesmi** (rodič - potomok). Posielanie dát medzi procesmi je založené na koncepte bajtových prúdov, čo znamená, že správy nie sú obmedzene vo veľkosti. Dáta zapisované do rúry nie sú interpretované systémom. Ak je potrebná interpretácia, čítajúci proces ju musí zabezpečiť.

V aplikáciách, napísaných v C jazyku je možné používať rúry pomocou volania `pipe()`.

```
#include <unistd.h>

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

Po vykonaní volania `pipe()`, systém vytvorí rúru a vráti 2 deskriptory - `pipe_desc[1]` pre zápis a `pipe_desc[0]` pre čítanie. Návratová hodnota z tohto volania je 0 pri úspešnom vykonaní, alebo –1 pri neúspešnom vykonaná a je nastavená hodnota chyby v premennej `errno`.

Príklad 7.1: Vytvorenie a použitie rúry

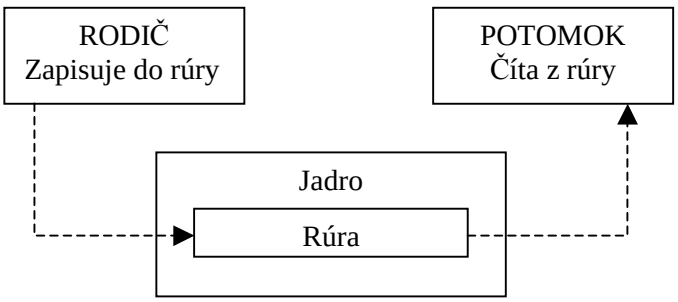
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main (int argc, char *argv[]) {
    int data_out;
    int pipe_desc[2];
    const char data[] = "AHOJ SVET!";
    char buffer[BUFSIZ + 1];

    memset(buffer, '\0', sizeof(buffer));
    if (pipe(pipe_desc) == 0) { // vytvorenie rúry
      // zápis do rúry pomocou deskriptora výstupu
        data_out = write(pipe_desc[1], data, strlen(data));
        printf("Zapisane %d bytov\n", data_out);
    // čítanie z rúry pomocou deskriptora vstupu
        data_out = read(pipe_desc[0], buffer, BUFSIZ);
        printf("Precitane %d bytes: %s\n", data_out, buffer);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

Kapacita rúry je obmedzená, jej veľkosť sa líši v jednotlivých implementáciách Linuxu. Od Linux 2.6.11 kapacita rúry je 65536 bajtov. Pri operácií `write()`, ak rúra je plná, operácia bude zablokovaná, alebo sa volanie vráti s chybou v závislostí od toho, či je nastavený príznak `O_NONBLOCK` alebo nie je nastavený. Neblokujúcu operáciu je možne nastaviť pomocou funkcie `fcntl()`s príkazom `F_SETFL` a príznakom `O_NONBLOCK`.

Komunikácia medzi dvoma procesmi je znázornená na nasledujúcom obrázku.



Obrázok 7.1: Komunikácia medzi dvomi procesmi pomocou rúry

Príklad 7.2: Komunikácia medzi dvomi procesmi – rodič –potomok

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

int main (int argc, char *argv[]){
int data_out;
int pipe_desc[2];
const char data[] = "Horuci letny den!!!";
char buffer[BUFSIZ + 1];
int fork_result;

memset(buffer, '\0', sizeof(buffer));
if (pipe(pipe_desc) == 0) {
fork_result = fork();
if (fork_result == -1) {
            fprintf(stderr, "Zlyhanie forku");
            exit(EXIT_FAILURE);
        }
        if (fork_result == 0) {        /* Potomok */
            data_out = read(pipe_desc[0], buffer, BUFSIZ);
            printf("\n Precitane %d bajtov: %s\n", data_out,
buffer);
            exit(EXIT_SUCCESS);
        } else {                       /* Rodic */
            data_out= write(pipe_desc[1],data, strlen(data));
            printf("\n Zapisane %d bajtov\n", data_out);
        }
    }
    exit(EXIT_SUCCESS);
}
```

V predchádzajúcom príklade aj rodič aj potomok používajú deskriptory rúry, ktoré potomok zdedí od rodiča.

Ak deskriptor, ktorý odkazuje na vstup do rúry bol uzatvorený, pri pokuse čítať z rúry, sa prečíta koniec súboru, t.j. operácia `read()` vráti 0.

Ak deskriptor, ktorý odkazuje na výstup z rúry bol uzatvorený, pri pokuse zapísať do rúry, operácia `write()` spôsobí vyslanie signálu `SIGPIPE`. Ak volajúci proces ignoruje tento signál, operácia `write()` skončí s chybou `EPIPE`. Aplikácia, ktorá používa systémové volania `pipe()` a `fork()` by mala uzatvoriť pomocou funkcie `close()` nepotrebné duplicitné deskriptory, čo zaistí to, že koniec súboru alebo stavy `SIGPIPE/EPIPE` sa objavia len v oprávnených situáciách.

Štandard POSIX.1-2001 hovorí, že operácia `write()`, ktorá zapisuje menej bajtov ako je konštanta `PIPE_BUF` (používa sa pri operáciach zápisu do rúr, ale nie priamo v programoch, v Linux-e je 4096 bajtov) musí byť atomická (nedeliteľná), t.j. dáta musia

byť zapísané do rúry súvisle. Zápis dát, ktoré sú dlhšie ako `PIPE_BUF` nemusí byť atomickou operáciou, jadro môže poprekladať dáta s dátami zapísanými iným procesom, na čo si treba dať pozor pri programovaní.

Možne kombinácie:

- `O_NONBLOCK` nie je nastavený, *n* <= `PIPE_BUF`

  Všetkých n bajtov je zapisaných do rúry atomicky. Zápis môže byť zablokovaný ak nie je dostatočný priestor pre zápis n bajtov naraz,

- `O_NONBLOCK` je nastavený, *n* <= `PIPE_BUF`

  Ak je dostatočný priestor pre zápis n bajtov do rúry, funkcia sa úspešne vráti ihneď, inák sa vráti s chybou `EAGAIN`, nastavenou v *errno*,

- `O_NONBLOCK` nie je nastavený, *n* > `PIPE_BUF`

  Zápis nie je atomický, dáta múžu byť poprekladané zápismi od iných procesov, funkcia `write()` sa zablokuje kým je všetkých *n* bajtov zapisaných,

- `O_NONBLOCK` je nastavený, *n* > `PIPE_BUF`

  Ak rúra je plná, zápis končí neúspešne s chybou v *errno* nastavenou na `EAGAIN`. Inák môže byť zapisano od 1 po n bajtov, presný počet je možne zístiť podľa návratovej hodnory z funkcie `write()`. Ako už bolo povedané, tieto bajty môžu byť poprekladané zápismi od iných procesov.

7.1.2 Systémové volanie `dup()`

```
#include <unistd.h>

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

Systémové volanie `dup()` duplikuje existujúci deskriptor súboru (rúry). Starý deskriptor zostáva nezmenený a obidva deskriptory ukazujú na ten istý súbor alebo rúru. Jediný význam tohto duplikovania je, že nový deskriptor môže byť z určitých dôvodov výhodnejší pre proces ako starý. Pri duplikovaní deskriptora sa aplikuje pravidlo, že sa prideľuje najnižšie voľné číslo deskriptora. Toto pravidlo môže byť využité tak, že sa získa číslo 0 pre deskriptor, ktorý popisuje koniec rúry určený na čítanie nasledovným spôsobom: uzatvorí sa deskriptor 0, ktorý popisuje systémový vstup (tím sa deskriptor 0 uvoľní) a následne sa zavolá `dup()` s čítacím deskriptorom rúry. Keďže najnižšie voľné číslo pre deskriptor je 0, to bude priradené deskriptoru rúry pre čítanie. To znamená, že čítanie sa bude vykonávať z rúry, namiesto zo systémového vstupu. Tento postup je zrejmý z nasledujúceho príkladu.

Príklad 7.3: Využitie duplikovania deskriptora rúry

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

int main (int argc, char *argv[]){
    int data_out;
    int file_pipes[2];
    char data[4096];
    int fork_result, status;

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Zlyhanie forku");
            exit(EXIT_FAILURE);
        }
        if (fork_result == 0) {
           write(file_pipes[1], data, strlen(data));
           close(1);              // uzatvorenie standardneho vystupu
           dup(file_pipes[1]); // duplikovanie popisovaca
           close(file_pipes[0]);
                                  // uzatvorenie nepotrebneho popisovaca
           close(file_pipes[1]);
           execl( "/bin/cat", "cat","/etc/profile", (char *)0);
                                 //výstup z cat  pojde do rúry
           exit(EXIT_FAILURE);
        } else  {             // Rocič
            memset(data, '\0', sizeof(data));
            data_out = read(file_pipes[0], data, sizeof(data));
            printf("\n Rodic s cislom %d precital %d
                       bajtov\n\n",
                    getpid(), data_out);
            printf(" %s", data);
            wait(&status);
        }
    }
    exit(EXIT_SUCCESS);
}
```

Uzatvorenie štandardného výstupu a duplikovanie deskriptora rúry v tomto prípade umožní poslať potomkovi výpis súboru */etc/profile* na vstup rodiča, ktorý ho prípadne môže ďalej spracovávať.

7.2 Pomenované rúry

Pomenovaná rúra alebo špeciálny súbor FIFO, je prostriedok pre komunikáciu medzi procesmi podobný rúre s tým rozdielom, že je súčasťou súborového systému. Môže byť otvorený súčasne viacerými procesmi pre čítanie alebo zápis. Keď si procesy vymieňajú dáta pomocou pomenovanej rúry, jadro ich odovzdáva bez zápisu do súborového systému. Pomenovaná rúra nemá dáta v súborovom systéme, meno špeciálneho FIFO súboru slúži len ako referenčný bod a pomocou mena rúry **môžu nadviazať spojenie aj nepríbuzné procesy**.

Jadro udržuje jednu rúru pre každý FIFO súbor, ktorý je otvorený aspoň jedným procesom. FIFO súbor musí mať otvorené obidva konce (aj na čítanie, aj na zápis) predtým, ako sa môžu posielať dáta. Obyčajne otvorenie pomenovanej rúry je blokujúca operácia kým sa neotvorí aj druhý koniec rúry.

Proces môže otvoriť pomenovanú rúru aj v neblokujúcom režime. V tomto prípade otvorenie iba pre čítanie bude pokračovať, aj keď druhý koniec rúry nie je otvorený. Otvorenie pre čítanie bude končiť s chybou `ENXIO` (*no such device or address*) pokiaľ druhý koniec rúry bol už otvorený.

V Linux-e volanie operácie otvorenia pomenovanej rúry sa vráti vždy, nezávislé či sa otvára v blokujúcom či neblokujúcom režime.

Príklad 7.4: Kód prvého z komunikujúcich procesov – fifoserver.c

```
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"
/* Meno, ktoré poznajú komunikujúce procesy */

int main (int argc, char *argv[]){
        FILE *fp;
        char readbuf[80];
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

/* Vytvorí FIFO ak neexistuje */
        while(1) {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Doruceny retazec: %s\n", readbuf);
                fclose(fp);
        }
        return(0);
}
```

Príklad 7.5: Kód druhého z komunikujúcich procesov – fifoclient.c

```
#include <stdio.h>
#include <stdlib.h>
#define FIFO_FILE       "MYFIFO"

int main(int argc, char *argv[]){
    FILE *fp;

    if ( argc != 2 ) {
        printf("USAGE: fifoclient [retazec]\n");
        exit(1);
        }
    if((fp = fopen(FIFO_FILE, "w")) == NULL) {
        perror("fopen");
        exit(1);
        }
        fputs(argv[1], fp);
        fclose(fp);
        return(0);
  }
```

Proces server, ktorý číta z rúry, musí bežať v okamihu, keď sa do pomenovanej rúry zapisuje. Preto sa musí najskôr spustiť na pozadí príkazom `fifoserver&` a následne sa spustí aj klient.

Pomenovanú rúru môžeme vytvárať aj v príkazovom jazyku pomocou príkazu *mkfifo*.

7.3 Súbory, mapované do pamäte

```
#include <sys/mman.h>

 void *mmap(void *start, size_t length, int prot, int flags, int
           fd, off_t offset);
 int munmap(void *start, size_t length);
```

Funkcia `mmap()` požaduje mapovanie určitého počtu bajtov do pamäte. Význam argumentov je nasledovný:

- *start* – preferovaná počiatočná adresa mapovania v pamäti, obyčajne sa zadáva 0, skutočná počiatočná adresa je v návratovej hodnote funkcie,
- *length* – počet mapovaných bajtov,
- *fd* – deskriptor mapovaného súboru,
- *offset* – posuv od začiatku súboru pre začiatok mapovania,
- *prot* – požadovaná ochrana pamäte, je to buď `PROT_NONE` (neprístupná) alebo je bitový súčet príznakov:
  - `PROT_EXEC` - stránky môžu byť vykonávané,
  - `PROT_READ` - stránky môžu byť čítané,

- PROT_WRITE - stránky môžu byť modifikované,
- PROT_NONE - k stránkam sa nemôže pristupovať.

- *flags* - špecifikuje typ mapovaného objektu, voľby mapovania a tiež voľbu, či zmeny v mapovaných stránkach budú vlastníctvom procesu, alebo budú zdieľané s ostatnými referenciami na mapovaný súbor. Tento parameter má bity:
  - MAP_FIXED – nedovoľuje inú počiatočnú adresu ako tú, ktorá je špecifikovaná v parametri *start*. Použitie tohto bitu sa neodporúča.
  - MAP_SHARED – zdieľanie mapovaného objektu s inými procesmi. Ukladanie do oblasti je ekvivalentné so zápisom do súboru. Súbor v skutočnosti nemusí byť aktualizovaný až do zavolania funkcie na synchronizáciu, msync() alebo funkcie na zrušenie mapovania munmap().
  - MAP_PRIVATE – vytvorí privátne mapovanie typu „*copy-on-write*“ () (kópia sa vytvorí len pri zmene obsahu stránky). Ukladanie do pamäťovej oblasti neovplyvňuje originálny súbor. Nie je špecifikované či zmeny urobené v súbore po jeho namapovaní do pamäte sú viditeľné v mapovanej oblasti.

Musí byť špecifikovaný práve jeden z príznakov MAP_SHARED a MAP_PRIVATE.

Pri úspešnom vykonaní mmap() vracia ukazovateľ na namapovanú oblasť v pamäti. Pri chybe vracia MAP_FAILED (t.j. (void *) -1) a nastaví globálnu premennú errno.

Príklad 7.6: Zápis do súboru, mapovaného do pamäte

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

/* Vracia náhodné číslo z rozsahu [low, high]. */

int random_range (unsigned const low, unsigned const high) {
  unsigned const range = high - low + 1;
  return low + (int) (((double) range) * rand () / (RAND_MAX +
                        1.0));
}
int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;

 /* Násada generátora náhodných čísiel. */
  srand (time (NULL));
```

```
  /* Príprava dostatočne veľkého súboru pre zápis celého čísla bez  znamienka.  */
    fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    lseek (fd, FILE_LENGTH+1, SEEK_SET);
    write (fd, "", 1);
    lseek (fd, 0, SEEK_SET);

  /* Mapovanie do pamäte.  */
    file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED,
                        fd, 0);
    close (fd);
   /* Zápis náhodného čísla do pamäťovej oblasti */
    sprintf((char*) file_memory, "%d\n", random_range (-100,
            100));
  /* Uvoľnenie pamäte, nepotrebné keď program končí */
    munmap (file_memory, FILE_LENGTH);
    return 0;
  }
```

Príklad 7.7: Čítanie zo súboru, mapovaného do pamäte

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;
  int integer;

 /* Otvorenie súboru  */
  fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
 /* Mapovanie do pamäte.  .  */
  file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
                                      MAP_SHARED, fd, 0);
  close (fd);

 /* Prečítanie čísla, vypísanie, násobenie 2 a opätovný zápis */
  sscanf (file_memory, "%d", &integer);
  printf ("Hodnota: %d\n", integer);
  sprintf ((char*) file_memory, "%d\n", 2 * integer);
 /* Uvoľnenie pamäte, nepotrebné keď program končí */
  munmap (file_memory, FILE_LENGTH);
  return 0;
}
```

7.3.1 Systémové volanie popen()

Funkcia popen() sa využíva na odovzdávanie dát z vykonania príkazu v bash-i na vstup procesu.

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

Funkcia popen() spustí proces tak, že vytvorí rúru, vytvorí nový proces a zavolá príkazový jazyk shell. Pretože rúra je jednosmerná, v argumente *type* sa musí zadať otvorenie len na čítanie alebo na zápis, nie oboje.

Argument *command* je ukazovateľ na reťazec, obsahujúci príkazový riadok shell-u. Tento riadok sa odovzdá príkazovému iterpreteru s modifikátorom -c (formát príkazu pre spustenie interpretera je bash –c command).

Návratová hodnota z popen() je normálny V/V prúd, ktorý musí byť uzatvorený volaným pclose() (pozor nie ako súbor cez fclose()!).

7.4 Základné ustanovenia pre prostriedky medziprocesnej komunikácie

Skupina prostriedkov – fronty správ, zdieľaná pamäť a semafory sú často v anglicky písanej literatúre označované pod spoločným názvom Interprocess Communication, v skratke IPC. Systémové volania, ktoré súvisia s týmito prostriedkami sa riadia podobnými pravidlami pri vytváraní, použití, získavaní informácií o príslušnom prostriedku a jeho zrušení. Každý prostriedok je označený jedinečným identifikátorom, ktorý sa získava na základe použitia kľúča. Komunikujúce procesy si kľúč musia dopredu dohodnúť.

Príkazový jazyk bash poskytuje príkazy, ktoré sú tiež spoločné pre túto skupinu prostriedkov. Sú to ipcs a ipcrm.

- ipcs – poskytuje informáciu o ipc prostriedkoch. Modifikátory [-asmq] sú voliteľné (a – všetky, s – semafory, m – zdieľaná pamäť, q – fronty správ). Príkaz bez modifikátora vypíše informáciu o všetkých prostriedkoch.
- ipcrm – umožňuje zrušenie príslušného prostriedku po zadaní jeho identifikátora (získa sa pomocou ipcs). Syntax zadávania príkazu je :

```
ipcrm [ -M kľúč | -m id | -Q kľúč | -q id | -S kľúč | -s id  ] ...
```

Príklad 7.8: Príkazy ipcs a ipcrm

```
$ ipcs
```

```
------ Shared Memory Segments --------
key      shmid    owner    perms    bytes    nattch   status

------ Semaphore Arrays --------
key      semid    owner    perms    nsems

------ Message Queues --------
key        msqid     owner      perms      used-bytes   messages
0x00000017 32768      marti      777        256          2

$ ipcrm -q 32768
$ ipcs

------ Shared Memory Segments --------
key      shmid    owner    perms    bytes    nattch   status

------ Semaphore Arrays --------
key      semid    owner    perms    nsems

------ Message Queues --------
key    msqid      owner    perms    used-bytes   messages
```

Pri vytváraní každého z IPC prostriedkov sa zadáva kľúč. Často sa tento kľúč vytvára pomocou funkcie `ftok()`. Syntax je nasledovná:

```
# include <sys/types.h>
# include <sys/ipc.h>

key_t ftok(const char *pathname, int proj_id);
```

Funkcia `ftok()` používa identitu súboru, ktorý je zadaný v argumente *pathname* (musí ukazovať na existujúci, prístupný súbor) a 8 najmenej významných bitov z čísla projektu *proj_id* a vygeneruje kľúč typu `key_t` , ktorý je vhodný pri volaní `msgget()`, `semget()` a `shmget()`.

Návratová hodnota z úspešného volania je kľúč, z neúspešného je -1.

7.5 Fronty správ

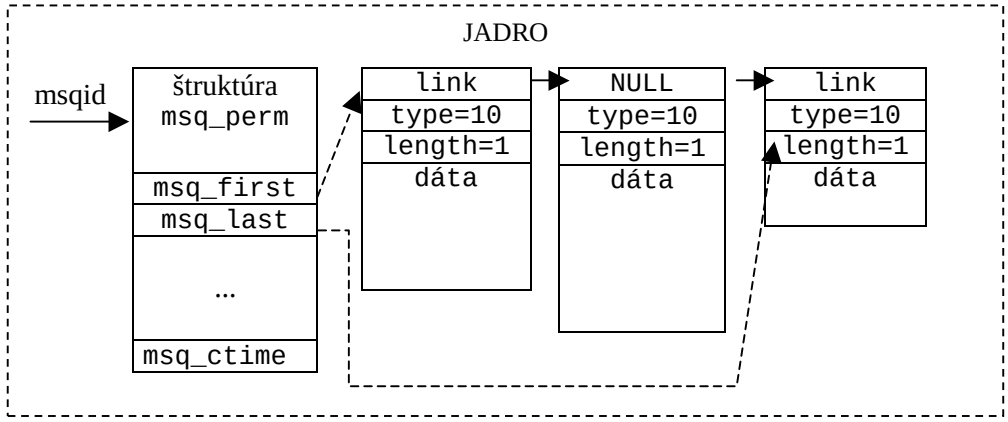
Fronty správ dovoľujú procesom vymieňať dáta vo forme správ. Fronty správ sú umiestnené v jadre a procesy pristupujú k ním pomocou identifikátorov. Procesy môžu čítať a zapisovať správy do/z každého frontu, ku ktorému majú prístup.

Každá správa má svoj typ, dĺžku a samozrejme dáta. Pre každú správu systém udržuje štruktúru s potrebnými informáciami:

```
struct msqid_ds {
    struct ipc_perm msg_perm;      /*  Vlastnictvo a prístupové práva */
```

```
time_t msg_stime;              /* čas posledného  msgsnd() */
time_t msg_rtime;              /* čas posledného  msgrcv() */
time_t msg_ctime;              /* čas poslednej  zmeny */
unsigned long __msg_cbytes;    /* Akt. počet bajtov vo fronte */
msgqnum_t msg_qnum;            /* akt. počet správ  vo fronte */
msglen_t msg_qbytes;           /* max. počet bajtov, dovolených vo fronte */
pid_t msg_lspid;               /* PID posledného  msgsnd() */
pid_t msg_lrpid;               /* PID posledného  msgrcv() */
};
```

Štruktúra `msg_perm` obsahuje prístupové práva pre daný front.



Obrázok 7.2: Štruktúra frontu správ v pamäti

7.5.1 Tvorba frontu správ

Nový front správ sa vytvára pomocou systémového volania `msgget()`.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

Argumenty volania majú nasledovný význam:

- `key` – kľuč vytváraného frontu. Procesy, ktoré chcú komunikovať musia poznať tento kľuč,
- `msgflg` – je to príznak, ktorý má jednu z týchto hodnôt:
  - `IPC_CREAT` – nastavenie bitu s touto hodnotou znamená, že sa vytvorí nový front pre zadaný kľúč, ak už neexistuje,

– IPC_PRIVATE – garantuje sa unikátny komunikačný kanál,

– IPC_EXCL – v kombinácií s IPC_CREAT spôsobuje chybu, ak front už existuje.

Návratová hodnota z tohto volania je identifikátor frontu s daným kľúčom. Ak takýto front existoval predtým, volajúci proces dostane identifikátor, ale front sa nevytvorí druhý krát. Treba podotknúť, že každý z procesov, ktorý chce komunikovať pomocou tohto frontu musí uviesť vo volaní msgget() príznak IPC_CREAT. Tento príznak sa obyčajne kombinuje (pomocou logickej operácie OR) s prístupovými právami k frontu. Prístupové práva musia obsahovať jeden z týchto režimov prístupu: O_RDONLY, O_WRONLY alebo O_RDWR. Prístupové práva sa môžu zadať symbolicky alebo číslom v osmičkovej sústave.

7.5.2 Zaslanie správy – msgsnd()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int
msgflg);
```

Toto systémové volanie slúži na zaslanie správy do frontu s identifikátorom msqid, ktorý bol získaný z volania msgget(). Ďalšie argumenty sú:

- msgp – ukazovateľ na štruktúru, v ktorej je uložená správa. Typ tejto správy je uložený v <sys/msg.h> a jej definícia je nasledovná:

```
struct msgbuf {
               long mtype;       /* typ správy, musí byť > 0 */
               char mtext[1];    /* dáta správy*/
        };
```

Typ správy musí byť väčší ako 0, pretože typ 0 má špeciálny význam pri prijímaní správ. Dáta správy nemusia byť typu char, dovolené sú aj binárne alebo textové dáta. Systém správu neinterpretuje. Ak si procesy potrebujú vymieňať dáta s inou štruktúrou, musia si ju zadefinovať.

- msgsz – udáva dĺžku správy v bajtoch – je to dĺžka dát, ktoré si zadefinoval používateľ a nasledujú za typom správy,
- msgflg – je to buď IPC_NOWAIT alebo 0. Hodnota IPC_NOWAIT vráti volanie okamžite, ak front správ je zaplnený. Ak IPC_NOWAIT je špecifikovaný a front je plný, návratová hodnota z msgsnd() je –1 a premenná errno je nastavená na EAGAIN.

Blokujúce volanie msgsnd() môže zlyhať ak front je zrušený, alebo je zachytený signál.

7.5.3 Prijatie správy - `msgrcv()`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
               int msgflg);
```

Prvé tri argumenty tohto volania sú rovnaké ako u `msgsnd()`. Ďalšie sú interpretované odlišným spôsobom:

- `msgtyp` - jeho hodnota určuje aký typ správy sa prijme:
  - ak `msgtyp` = 0, prijme sa prvá správa vo fronte,
  - ak `msgtyp` > 0, prijme sa prvá správa vo fronte s daným typom. Ak `msgflg` je nastavený na MSG_EXCEPT, prijme sa prvá správa, ktorej typ nie je rovný zadanému.
  - ak `msgtyp` < 0, prijme sa prvá správa vo fronte, ktorej typ je menší alebo rovný absolútnej hodnote argumentu `msgtyp`.
- `msgflg` – tento argument je bitovou maskou skonštruovanou pomocou operácie OR zo žiadneho alebo jedného z príznakov:
  - IPC_NOWAIT – vráti sa okamžite, ak vo fronte nie je správa požadovaného typu,
  - MSG_EXCEPT – používa sa s `msgtyp` > 0, ako bolo vysvetlené vyššie,
  - MSG_NOERROR – ak je nastavený, nevznikne chyba ak správa je dlhšia ako je `msgsz`.

Ak správa požadovaného typu sa nenachádza vo fronte a príznak IPC_NOWAIT nie je špecifikovaný, volajúci proces je zablokovaný, kým sa nevyskytne jedna z týchto udalostí:

- príde správa požadovaného typu,
- front je zrušený, v tom prípade vznikne chyba EIDRM,
- volajúci proces zachytí signál, v tomto prípade volanie zlyhá s chybou EINTR.

Pri úspešnom ukončení volania, jeho návratová hodnota je počet bajtov, ktoré boli doručené.

7.5.4 Kontrolné operácie nad frontom správ

Systémové volanie `msgctl()` umožňuje vykonávanie rôznych riadiacich operácií nad frontami správ.

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Argumenty volania majú nasledovný význam:

- `msqid` – identifikátor frontu, získaný pri jeho tvorbe,
- `cmd` – príkaz, ktorý určí aká kontrolná operácia sa uskutoční,
- `buf` – je to štruktúra typu `msqid_ds`. Každý front v systéme má svoju `msqid_ds` štruktúru. Je definovaná v `<sys/ipc.h>` takto:

```
struct msqid_ds {
    struct ipc_perm msg_perm;  /* vlastnictvo a prístupové
                                  práva*/
    time_t    msg_stime;       /* čas posledného msgsnd() */
    time_t    msg_rtime;       /* čas posledného msgrcv() */
    time_t    msg_ctime;       /* čas poslednej zmeny */
    unsigned long  __msg_cbytes;      /* počet bajtov vo fronte */
    msgqnum_t   msg_qnum;             /* počet správ vo fronte */
    msglen_t    msg_qbytes;    /* max. dovolený počet bajtov vo
                                  fronte */
    pid_t       msg_lspid;     /* PID posledného msgsnd() */
    pid_t       msg_lrpid;     /* PID posledného msgrcv() */
};
```

Štruktúra `ipc_perm` je definovaná v `<sys/ipc.h>` takto:

```
struct ipc_perm {
      key_t key;          /* kľúč pre msgget() */
      uid_t uid;          /* efektívne UID vlastníka */
      gid_t gid;          /* efektívne GID vlastníka */
      uid_t cuid;         /* efektívne UID tvorca */
      gid_t cgid;         /* efektívne GID tvorca */
      unsigned short mode;     /* prístupové práva */
      unsigned short seq;      /* poradové číslo */
  };
```

7.5.4.1 Získanie informácií

Pre získanie informácií o stave frontu správ sa ako **argument** `cmd` používa **IPC_STAT**. Tento príkaz skopíruje informácie o fronte do bufra, zadaného ako argument `buf`. Ten musí byť typu `msqid_ds` . Príklad volania:

```
…   if( msgctl( qid, IPC_STAT, qbuf) == -1)…
```

Volajúci proces musí mať právo čítania.

Návratová hodnota po úspešnom volaní je 0, inak je -1.

7.5.4.2 Nastavenie členov informačnej štruktúry frontu

Pre nastavenie hodnôt v štruktúre `msqid_ds` pre daný front správ, sa ako **argument** `cmd` používa **IPC_SET.** Požadované hodnoty sa zoberú zo štruktúry n ktorú ukazuje argument buf.

Nastavované môžu byť `msg_qbytes`, `msg_perm.uid`, `msg_perm.gid` a `msg_perm.mode` (najmenej významných 9 bitov). Efektívne UID volajúceho procesu musí byť rovnaké ako je UID vlastníka (`msg_perm.uid`) alebo tvorcu (`msg_perm.cuid`) frontu.

Príklad 7.9: Funkcia na zmenu prístupových práv frontu

```
int change_queue_mode( int qid, char *mode )
{
        struct msqid_ds tmpbuf;

    /* Získanie momentalného obsahu štruktúry */
        if( msgctl( qid, IPC_STAT, &tmpbuf) == -1)
        {
                return(-1);
        }

    /* Zmena prístupových práv */
        sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);

    /* Aktualizácia vnútornej štruktúry */
        if( msgctl( qid, IPC_SET, &tmpbuf) == -1)
        {
                return(-1);
        }
        return(0);
}
```

7.5.4.3 Zrušenie frontu

Pre zrušenie daného frontu správ sa ako **argument** `cmd` používa **IPC_RMID.** Je potrebné pripomenúť, že po ukončení komunikácie medzi procesmi, používaný front sa **musí zrušiť!** V opačnom prípade po určitom čase bude prekročený maximálny počet frontov v systéme.
Posledný argument volania `msgctl` v tomto prípade je nulový:

```
... if( msgctl( qid, IPC_RMID, 0) == -1)…
```

Príklad 7.10 : Proces vytvorí front a pošle doňho správu

```
/* Program vytvorí front sprav a pošle nejaký text */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define   KEY   ((key_t) 23)

#define   COUNT 5
#define   BUFFSIZE      128
#define   PERMS        0777


int main (int argc, char *argv[]){
    register int i, msqid;
    struct {
      long  m_type;
      char  m_text[BUFFSIZE];
    } msgbuff;
     struct msqid_ds moj_ds;

    if ((msqid = msgget(KEY, PERMS | IPC_CREAT))< 0)
                          printf("msgget error\n");
         else {
          if ((msgctl(msqid,IPC_STAT, &moj_ds) < 0))
              printf("Chyba IPC_STAT\n");
          else
          {
             printf(" Pocet bytov vo fronte:%d\n",
                     moj_ds.msg_qbytes);
             printf(" Pristupove prava
                                    %o\n",moj_ds.msg_perm.mode);
             }
     msgbuff.m_type = 1;
     strcpy(msgbuff.m_text, "Dobra sprava – su VIANOCE!!!");
    if (msgsnd(msqid,(char *)&msgbuff, BUFFSIZE,0) < 0)
          printf("Chyba msgsnd \n");
       else printf("Posielam spravu\n");
      }
    exit(0);
}
```

Príklad 7.11: Proces prečíta správu a zruší front

```
#include        <sys/types.h>
#include        <sys/ipc.h>
```

```
#include        <sys/msg.h>

#define KEY     ((key_t) 23)

#define COUNT   5
#define BUFFSIZE          128
#define PERMS            0777

int main (int argc, char *argv[]){
    int i, msqid;
        struct {
          long  m_type;
          char  m_text[BUFFSIZE];
        } msgbuff;
    struct msqid_ds moj_ds;

        if ((msqid = msgget(KEY, PERMS | IPC_CREAT))< 0)
                        printf("msgget error\n");
/* Každý proces, ktorý pracuje s frontom musí získať identifikátor pomocou tohoto volania */
        else {
         if ((msgctl(msqid,IPC_STAT, &moj_ds) < 0))
             printf("IPC_STAT error\n");
         else
           {
            printf(" pocet bytov vo fronte:%d\n",
                    moj_ds.msg_qbytes);
            printf(" PERMISSIONS  %o\n",moj_ds.msg_perm.mode);
            }

        msgbuff.m_type = 1;
        if (msgrcv(msqid,(char *)&msgbuff, BUFFSIZE,0, 0) < 0)
             printf("msgsnd error\n");
          else printf("Sprava:%s\n", msgbuff.m_text);
      }
       if (msgctl(msqid,IPC_RMID, 0)<0) // zrušenie frontu po prečítani správy
           printf("IPC_RMID error -front sa nepodarilo zrusit");
        exit(0);
}
```

7.6 Zdieľaná pamäť

Zdieľaná pamäť je najrýchlejším spôsobom komunikácie medzi procesmi v jednom systéme. Za normálnych okolností procesy nemôžu čítať alebo zapisovať údaje do adresných priestoroch iných procesov. Za týmto účelom systém poskytuje možnosť vytvorenia segmentu v pamäti, ktorý je prístupný viacerým procesom, ktoré poznajú dopredu dohovorený kľúč. Tento segment je potom pripojený k adresným priestorom komunikujúcich procesov a ti môžu pracovať s nim bežným spôsobom.

Prístup k zdieľanému segmentu nie je synchronizovaný. Ak programátor nezaisti synchronizáciu súbežného prístupu, konzistencia dát nie je zaručená!

7.6.1 Získanie zdieľaného segmentu

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

Systémové volanie `shmget ()` alokuje zdieľaný pamäťový segment.

Argumenty volania majú následovný význam:

- `key` – kľuč vytváraného segmentu. Procesy, ktoré chcú komunikovať musia poznať tento kľuč,
- `size` – rozmer zdieľaného segmentu v bajtoch, zaokrúhlený na násobok konštanty PAGE_SIZE (veľkosť stránky),
- `shmflg` – je to príznak, ktorý môže nadobudnúť jednu alebo viacej z týchto hodnôt:

  - IPC_CREAT – nastavenie bitu s touto hodnotou znamená, že sa vytvorí nový segment pre zadaný kľúč, ak už neexistuje. Ak tento príznak nie je použitý, `shmget` pohľadá či existuje segment k zadanému kľúču a či volajúci proces má právo prístupu k nemu. Ak sú špecifikované IPC_CREAT a IPC_EXCL a zdieľaný segment už existuje, funkcia sa vráti s chybou a `errno` bude nastavená na EEXIST,

  - IPC_EXCL – v kombinácií s IPC_PRIVATE spôsobuje chybu, ak segment už existuje

  - príznaky prístupu špecifikujú prístupové práva pridelené vlastnikovi, skupine a ostatným.

Návratová hodnota z tohto volania je identifikátor segmentu s daným kľúčom. Nový segment sa vytvorí ak predtým neexistoval ak argument key má hodnotu IPC_PRIVATE alebo ak nemá túto hodnotu je zadaný príznak IPC_CREAT a segment s takým kľúčom neexistuje.

7.6.2 Pripojenie zdieľaného segmentu k procesu

```
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Funkcia `shmat()` pripája zdieľaný segment k adresnému priestoru procesu. Argumenty volania sú:

- `shmid` - identifikátor, získaný z volania shmget,
- `shmaddr` - adresa v pamäti, na ktorej chceme pripojiť segment. Ak táto adresa je NULL, systém vyberie vhodnú adresu, kde prípoji segment. Ak tento argument nie je NULL a príznak je nastavený na SHN_RND, pripojenie sa uskutoční na adresu, ktorá je rovná požadovanej s tým, že sa zaokrúhli k najbližšiemu násobku konštanty SHMLBA.
- `shmflg` - ak je zadaný príznak SHM_RDONLY, segment je pripojený na čítanie aproces musú mať čítanie povolenú. Ináč segment sa pripojí pre čítanie a zápis a proces musí tieto práva mať.

Po úspešnom volaní proces `shmat()` dostane adresu pripojeného segmentuako návratovú hodnotu a príslušné štruktúry v `shmid_ds` (pozri popis v oddieli o `shmctl`) sa zmenia.

7.6.3 Odpojenie zdieľaného segmentu od procesu

```
int shmdt(const void *shmaddr);
```

Po ukončení práce so zdieľaným segmentom, ten sa odpojí od adresného priestoru procesu. Jediným argumentom tohto volania je adresa `shmaddr`, na ktorej bol segment pripojený pomocou `shmat()`.

7.6.4 Kontrolné operácie nad zdieľaným segmentom

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Systémové volanie `shmctl()` vykonáva kontrolné operácie nad zdieľaným segmentom podobne ako u frontov správ ako získania informácií, nastavenie položiek do štruktúry `shmid_ds` a zrušenie zdieľaného segmentu. Argumenty volania sú:

- `shmid` - identifikátor segmentu, získaný zo shmget(),
- `cmd` - požadovaná operácia nad segmentom,
- `buf` - ukazovateľ na štruktúru typu `shmid_ds`, kde sa nastavujú alebo ukladajú požadované informácie (pre príkazy IPC_GET a IPC_SET).

Argument `buf` ukazuje na štruktúru, definovanú v `<sys/shm.h>` takto:

```
struct shmid_ds {
    struct ipc_perm    shm_perm;  /* vlastnictvo a prístupové práva*/
    size_t      shm_segsz;        /* rozmer segmentu v bajtocj */
    time_t      shm_atime;        /* čas posledného pripojenia */
    time_t      shm_dtime;        /* čas posledného odpojenia */
    time_t      shm_ctime;        /* čas poslednej zmeny */
    pid_t       shm_cpid;         /* PID tvorcu */
    pid_t       shm_lpid;         /* PID posledného shmat()/shmdt() */
    shmatt_t    shm_nattch;       /* Počet pripojení v danom momente */
           };
```

Štruktúra `ipc_perm` je definovanú v `<sys/shm.h>` takto:

```
struct ipc_perm {
    key_t key;              /* kľúč pre shmget() */
    uid_t uid;              /* efektívne UID vlastníka */
    gid_t gid;              /* efektívne GID vlastníka */
    uid_t cuid;             /* efektívne UID tvorca */
    gid_t cgid;             /* efektívne GID tvorca */
    unsigned short mode;    /* prístupové práva plus priznáky SHM_DEST a
                               SHM_LOCKED */
           };
```

Platí, že úspešne vykonané volanie vráti hodnotu 0, neúspešne –1.

7.6.4.1 Získanie informácií o zdieľanom segmente

Použije sa volanie `shmctl`, kde argument `cmd` je IPC_STAT. Systém skopíruje informácie do bufra, ktorého ukazovateľ je v argumente `buf`. Volajúci proces musí mať právo čítania.

7.6.4.2 Nastavenie členov informačnej štruktúry segmentu

Použije sa volanie `shmctl()`, kde argument `cmd` je IPC_SET. Systém skopíruje informácie nastavené v bufri, ktorého ukazovateľ je v argumente `buf` do systémovej štruktúry. Polia štruktúry, ktoré sa môžu zmeniť sú: `shm_perm.uid`, `shm_perm.gid`, a (najmenej významných 9 bitov zo) `shm_perm.mode`.

7.6.4.3 Zrušenie zdieľaného segmentu

Použije sa volanie `shmctl`, kde argument `cmd` je IPC_RMID. Segment bude označený na zrušenie, ale skutočne bude zrušený až keď posledný proces ho odpojí od svojho adresného priestoru (alebo bude zrušený). Volajúci proces musí byť vlastník, tvorca alebo privilegovaný.

Príklad 7.12: Process, ktorý číta zo zdieľaného segmentu

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

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MEM_SZ 4096   // veľkosť zdieľaneho segmentu

struct sharedmem_t {
    int input;
    char text[BUFSIZ];
};

int main (int argc, char *argv[])
{
    int running = 1;
    void *shared_memory = (void *)0;
    struct sharedmem_t *shared_data;
    int shmid;

    srand((unsigned int)getpid());
   /* Vytvorenie zdielaneho segmentu */
   shmid = shmget((key_t)1234, MEM_SZ, 0666 | IPC_CREAT);
    if (shmid == -1) {
        fprintf(stderr, "Chyba shmget\n");
        exit(EXIT_FAILURE);
    }

    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "Chyba shmat \n");
        exit(EXIT_FAILURE);
    }

    printf("Pamatovy segment připojeny na %X\n",
(int)shared_memory);
    shared_data = (struct sharedmem_t *)shared_memory;
    shared_data->input = 0;
    while(running) {
       if (shared_data->input) {
          printf("Výpis zo zdieľanej pamäte: %s", shared_data-
>text);
            sleep( rand() % 4 ); /* Chvilku ostatne procesy budu cakat */
            shared_data->input = 0;
            /* citanie sa ukončí po zadaní slova „koniec“ */
```

```
            if (strncmp(shared_data->text, "koniec", 5) == 0) {
                running = 0;
            }
        }
    }
    if (shmdt(shared_memory) == -1) {   // odpojenie zdieľaného segmentu
        fprintf(stderr, "Chyba shmdt \n");
        exit(EXIT_FAILURE);
    }
    if (shmctl(shmid, IPC_RMID, 0) == -1) {    //zrušenie segmentu
        fprintf(stderr, "Chyba shmctl(IPC_RMID)\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

Príklad 7.13: Proces ktorý číta zo vstupu a zapisuje do zdieľanej pamäte

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

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MEM_SZ 4096

struct sharedmem_t {        /* vlastná štruktúra zdieľanej pamäte */
    int input;              /* pomocná premenná */
    char text[BUFSIZ];      /* samotné dáta */
};

int main (int argc, char *argv[])
{
    int running = 1;
    void *shared_memory = (void *)0;
    struct sharedmem_t *shared_data;
    char buffer[BUFSIZ];
    int shmid;
    /* vytvorenie zdieľaného segmentu */
    shmid = shmget((key_t)1234, MEM_SZ, 0666 | IPC_CREAT);

    if (shmid == -1) {
        fprintf(stderr, "Chyba shmget \n");
        exit(EXIT_FAILURE);
    }
    /* pripojenie segmentu */
```

```
    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "Chyba shmat \n");
        exit(EXIT_FAILURE);
    }

    printf("Pamatovy segment pripojeny na %X\n",
(int)shared_memory);

    shared_data = (struct sharedmem_t *)shared_memory;
    while(running) {
        while(shared_data->input == 1) {
            sleep(1);
            printf("Cakam na vstup...\n");
        }
        printf("Napis nieco: ");
        fgets(buffer, BUFSIZ, stdin);

        strcpy(shared_data->text, buffer); // zápis do zdieľanej pamäte
        shared_data->input = 1;

        if (strncmp(buffer, "koniec", 5) == 0) {
                running = 0;
        }
    }

    if (shmdt(shared_memory) == -1) { // odpojenie zdieľaného segmentu
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

Poznámka: Keďže procesy majú fungovať súčasné, je potrebné jeden z nich spustiť na pozadí (pomocou &).

7.7 Semafory

Semafory sú synchronizačnými prostriedkami. Sú zaradené medzi IPC prostriedky, ale nie sú komunikačnými prostriedkami, t.j. pomocou semaforov sa nedajú prenášať dáta. Ich funkciou je synchronizovať prístup paralelne bežiacich procesov k zdieľaným prostriedkom (súbory, premenné, zdieľaná pamäť a ďalšie), čím sa zachová konzistencia dát.

Semafor je prostriedok, ktorý má svoju hodnotu a má definované operácie, ktoré sa nad touto hodnotou dajú vykonať. Podstatou synchronizačných schopností semaforov je, že tieto operácie sú **atomické,** t.j. sú vykonávané bez prerušenia.

Hodnota semafora sa mení len operáciami. V literatúre operácia, ktorá **odpočítava** určitú hodnotu od hodnoty semafora sa nazýva **`wait`** a operácia, ktorá **pripočítava** určitú

hodnotu k hodnote semafora sa nazýva **`signal`**. Pri prístupe k zdieľanému prostriedku proces volá operáciu **`wait`**, pri uvoľnení prostriedku – operáciu **`signal.`**

Podľa toho ako pokračuje vo svojom vykonaní volajúci proces v prípade, že zdieľaný prostriedok je využívaný iným procesom rozlišujeme semafory **s aktívnym čakaním** alebo **s pasívnym čakaním**.

V prvom prípade – s aktívnym čakaním, proces zostáva aktívnym (prideľuje sa mu čas procesora), ale neustále testuje hodnotu semafora.

V prípade pasívneho čakania k semaforu je priradený aj front čakajúcich procesov a volajúci proces je zablokovaný kým sa hodnota semafora nezvýši (neprideľuje sa mu čas procesora). Formálne môžeme vyjadriť semaforové operácie nasledovne:

`wait` (sem): sem.hodnota = sem.hodnota – 1;

if sem.hodnota < 0 → zablokuj proces do frontu čakajúcich procesov;

`signal` (sem): sem.hodnota = sem.hodnota + 1;

if sem.hodnota > 0 → odblokuj jeden z čakajúcich procesov;

Procesy, ktoré synchronizujú svoju činnosť musia dodržať určitú postupnosť vykonania:

- **pred prístupom** k zdieľanému prostriedku zavolať operáciu `wait`(sem),
- **po ukončení** svojej činností so zdieľaným prostriedkom zavolať operáciu `signal` (sem).

Doteraz popísaná činnosť semaforových operácií platí pre **binárne semafory**, t.j. semafory, ktorých hodnota je buď 1 alebo 0.

Implementácia semaforov v Linix-e je podľa štandardu POSIX a oproti predchádzajúcej definície je rozšírená nasledovne:

- semafor je realizovaný ako sada semaforov, pričom operácie nad celou sadou sú atomické,
- hodnota semafora nie je obmedzená na 0 a 1, semaforové operácie môžu nadobúdať aj iné hodnoty.

Implementácia semaforov pod Linux-om je rozsiahla a využitie všetkých možností vyžaduje opatrnú implementáciu synchronizačných úloh. Najčastejšia chyba je použitie kontrolnej operácie GETVAL (vo volaní `semctl())` na synchronizáciu. Táto operácia bola navrhnutá **len pre informáciu, nie pre synchronizáciu.**

Druhý problém, ktorý môže vzniknúť pri použití semaforov je uviaznutie procesov. Pre ilustráciu tejto situácie si predstavme systém s dvomi procesmi - P_0 a P_1, ktoré používajú dva semafory *S* a *Q*, nastavené na hodnotu 1:

```
    P0                  P1
wait(S);            wait(Q);
wait(Q);            wait(S) ;
    .                   .
    .                   .
    .                   .
signal(S);          signal(Q);
signal(Q);          signal(S);
```

Predpokladajme, že P_0 vykonal operáciu `wait(S)` a potom P_1 vykonal operáciu `wait(Q)`. Keď P_0 vykoná `wait(Q)`, bude musieť počkať, kým P_1 vykoná `signal(Q)`. Podobne keď P_1 vykoná `wait(S)`, bude musieť počkať, kým P_0 vykoná `signal(S)`. Pretože tieto operácie sa nemôžu vykonať, P_0 a P_1 uviaznu.

7.7.1 Tvorba sady semaforov

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Systémové volanie `semget()` vracia sadu semaforov. Argumenty volania sú:

- `key` - kľúč, ktorý identifikujúci sadu semaforov. Nová sada sa vytvorí ak hodnota kľúča je IPC_PRIVATE, alebo sada s týmto kľúčom neexistuje a argument `semflg` je nastavený na IPC_PRIVATE. Ak sú zadané IPC_CREAT a IPC_EXCL a sada semaforov pre tento kľúč už existuje, volanie zlyhá a do `errno` sa nastaví hodnota EEXIST,
- `nsems` - počet semaforov v sade,
- `semflg` - najmenej významných 9 bitov tohto argumentu určuje prístupové práva pre sadu (vlastník, skupina, ostatní). Právo vykonania samozrejme v tomto prípade nemá význam. Právo zápisu znamená právo meniť hodnotu semafora.

Pri tvorbe sady semaforov `semget()` vytvorí a inicializuje štruktúru `semid_ds`, ktorá ju popisuje.

Návratová hodnota – identifikátor vytvorenej sady semaforov, pri neúspešnom vykonaní návratová hodnota je –1. **Po tomto volaní semafory v sade nie sú inicializované**, za týmto účelom je treba použiť volanie `semctl()` (pozri ďalej).

7.7.2 Operácie nad sadou semaforov

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned
          nsops);

int  semtimedop(int semid, struct sembuf *sops, unsigned
                nsops, struct time spec *timeout);
```

Systémové volanie `semop()` vykonáva operácie nad semaformi v sade. Argumenty volania sú:

- `semid` - identifikátor sady, získaný z volania `semget()`,
- `*sops` - ukazovateľ na pole s prvkami typu `sembuf`, ktoré popisujú každý semafor v sade,
- `nsops` - počet prvkov v `*sops`. Prvky sú číslované od 0 po počet semaforov-1,

Štruktúra `sembuf` je definovanú následovne:

```
struct sembuf {
     unsigned short sem_num;        /* číslo semafora v sade */
     short  sem_op;                 /* operácia nad semaforom */
     short  sem_flg;                /* priznak operácie  */
}
```

Sada operácií v `*sops` **je vykonaná atomicky**, t.j. operácie sú vykonané súčasne a to len vtedy, ak je možne vykonať všetky naraz. Správanie volania v prípade, že operácie sa nedajú vykonať naraz je určené od prítomností príznaku `IPC_NOWAIT` v `sem_flg`. Príznaky, ktoré sa môžu použiť v `sem_flg` sú:

- `IPC_NOWAIT` - ak je zadaný tento príznak, pri nemožnosti vykonať všetky semaforové operácia naraz, volanie sa vráti. Ak príznak nie je zadaný, volajúci proces bude zablokovaný kým sa hodnota semafora nezmení,
- `SEM_UNDO` - pri zadaní tohto príznaku sa pri nenormálnom ukončení procesu automaticky vykoná operácia s opačnou hodnotou ako je zadaná v `sem_op`. Tým sa eliminuje vplyv ukončeného procesu a predchádza sa uviaznutiu ostatných procesov, ktoré používajú tento semafor.

Každý semafor v sade je popísaný štruktúrou typu sem:

```
sruct sem {
     unsigned short  semval; /* hodnota semafora */
     nsigned short  semzcnt; /* čakaúci na znulovanie */
     nsigned short  semncnt;  /* čakaúci na zvyšenie hodnoty */
     id_t   sempid;           /* proces, ktorý urobil poslenú operáciu */
};
```

Operácie nad semaforom sú definované, ako už bolo uvedene vyššie, v `sem_op`. Hodnota tejto premennej určuje či ta operácia bude `wait` alebo `signal` v zmysle klasickej definície semaforov. Hodnota `sem_op` môže byť kladná, záporná alebo 0.

- sem_op > 0 - hodnota operácie sa pridá k momentálnej hodnote semafora `sem_val` . Ak bol špecifikovany SEN_UNDO, aktualizuje sa ja hodnota `semadj` (pre prípadne vykonanie opačnej operácie pri nenormálnom ukončení procesu)
- `sem_p` = 0 to znamená, že volajúci proces chce počkať kým hodnota semafora bude 0. Výsledok závisí od momentálnej hodnoty `sem_val`. Ak tá je 0, proces môže pokračovať, ak nie je 0, bude zablokovaný až kým hodnota semafora nebude 0. `semzcnt` je počet procesov, ktoré čakajú na nulovú hodnotu semafora,
- `sem_p` < 0 - význam zápornej hodnoty operácie je, že volajúci proces chce počkať až bude `sem_val` >= |`sem_op`| (absolútna hodnota).

7.7.3 Kontrolné operácie nad sadou semaforov

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

Systémové volanie semctl() vykonáva rôzne kontrolné operácie nad semaformi. Argumenty tohto volania sú:

- `semid` - identifikátor sady semaforov,
- `semnum` - čislo semafora zo sady na ktorého sa príkaz `cmd` vzťahuje,
- `cmd` - príkaz, ktorý sa vykoná,
- ďalší argument - celkový počet argumentov môže byť 3 alebo 4 a to závisí od zadaného príkazu.

Ak funkcia použije 4 argumenty, je potrebne definovať union typu `semun`. Jeho definícia je nasledovná:

```
union semun {
    int  val;                   /* hodnota pre SETVAL */
    struct semid_ds *buf;       /* bufer pre IPC_STAT, IPC_SET */
    unsigned short  *array;     /* pole GETALL, SETALL */
    struct seminfo  *__buf;     /* bufer pre  IPC_INFO
                                   (Linux specific) */
};
```

Dátová štruktúra `semid_ds` je definovaná v `<sys/sem.h>` takto:

```
struct semid_ds {
    struct ipc_perm sem_perm;  /* vlastnictvo a prístupové práva*/
    time_t          sem_otime; /* čas poslednej semop */
    time_t          sem_ctime; /* čas poslednej zmeny*/
    unsigned short  sem_nsems; /* počet semaforov v sade */
};
```

Príkazy, ktoré sa môžu použiť v argumente `cmd` sú:

- `IPC_STAT` - skopíruje informáciu z štruktúry jadra do 4-tého argumentu typu `semid_ds`. Argument `semnum` je ignorovaný.
- `IPC_SET` - nastavuje hodnoty zo štruktúry `semid_ds` do štruktúry jadra toho istého typu. Efektívne UID volajúceho procesu a vlastníka alebo tvorcu musia byť rovnaké, alebo proces musí byť privilegovaný. Argument `semnum` je ignorovaný.
- `IPC_RMID` - Okamžite zruší sadu semaforov a odblokuje všetky procesy, čakajúcich na navrat z volania semop().Efektívne UID volajúceho procesu a vlastníka alebo tvorcu musia byť rovnaké, alebo proces musí byť privilegovaný. Argument `semnum` je ignorovaný.
- `GETVAL` - `semctl()` vráti hodnotu semafora č. `semnum`. POZOR! Táto operácia **nie je atomická** a je **NEVHODNÁ NA SYNCHRONIZÁCIU!!!**
- `GETALL` - vráti hodnoty `semval` všetkých semaforov v sade do poľa `arg.array` zadaného ako 4-tý argument. Počet prvkov tohto poľa je rovný počtu semaforov v sade a každý prvok je typu `ushort`.
- `SETVAL` - nastaví hodnotu semafora č. `semnum` na hodnotu zadanú ako
  4-tý argument (typ `int`) a aktualizuje príslušné štruktúry. Ak zmenená hodnota semafora dovolí, odblokuje sa jeden proces z čakajúcich na zvýšenie hodnoty semafora. Volajúci proces musí mať právo zmeny.
- `SETALL` - nastaví hodnoty všetkých semaforov naraz. 4-tý argument je ukazovateľ na pole prvkov typu `unsigned short`.
- `GETZCNT` - vracia počet procesov čakajúcich aby príslušný semafor nadobudol hodnotu 0.
- `GETNCNT` - vracia počet procesov čakajúcich na zvýšenie hodnoty príslušného semafora.
- `GETPID` - vracia PID procesu, ktorý vykonal poslednú operáciu nad semaforom alebo sadou semaforov.

Príklad 7.14: Práca so semaformi

```
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/sem.h>
#include <stdio.h>
#include <unistd.h>
#define COUNT   10

struct sembuf   signal[1]   = { 0,  1, 0 };
struct sembuf   wait[1] = { 0, -1, 0 };
union  semun  {
      int val;
      struct semid_ds *semo;
      ushort *array;
} mojun;

int main (int argc, char *argv[]){

   register int  semid;
   int s,i,status;
   extern int errno;
   char *t[20];

    if ( (semid = semget((key_t)123, 1, 0666 | IPC_CREAT)) < 0)
               printf("semget error\n");
    semctl(semid,0,SETVAL,1 );/* Inicializácia semafora */
    printf("Semafor id = %d, proces %d \n",semid, getpid());
    printf(" Hodnota semafora: %d\n",semctl(semid,0,GETVAL ) );
    printf("\nPROCES  %d\n", getpid());
    if (semop(semid, &wait[0], 1) < 0)
            printf("semop up error%\n");
    printf(" \n WAIT, hodnota je ");
    printf("   -  %d\n",semctl(semid,0,GETVAL));
    for (i=1;i<10;i++){
           printf("Proces %d vykonava nejaku pracu\n",getpid());
    }
    printf("\n SKONCIL SOM - volam signal\n");
    if (semop(semid, &signal[0], 1) < 0)
           printf("chyba pri UP\n");
    return 0;
}
```

Príklad 7.15: Paralelný prístup dvoch procesov ku kritickej sekcií (KS)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
union semun {
            int val;
            struct semid_ds *buf;
            ushort *array;
                 };
int semid; /* semid sady semaforov */
struct sembuf *sops;
int s=0;
int status;

int wait_sem(int index, int pid){
    fprintf(stderr, "------- Proces %d vykonava operaciu
            wait (-1)\n",pid);
       sops[0].sem_num = index;
       sops[0].sem_op = -1; /* Ziska semafor */
       sops[0].sem_flg = SEM_UNDO ;
       /* ak proces skonci skor, eliminuje jeho poslednu operaciu nad semaforom */

    if  (semop(semid, sops, 1)<0){
             perror("zlyhanie semop pri wait");
             return 1;
      }
      else
             return 0;
  }

 int signal_sem(int index, int pid){
      fprintf(stderr, "++++++ Proces %d vykonava operaciu
              signal (1)\n",pid);
      sops[0].sem_num = index;
      sops[0].sem_op = 1; /* opusta KS, zvysuje hodnotu semafora */
      sops[0].sem_flg = SEM_UNDO;
      if  (semop(semid, sops, 1)<0){
            perror("zlyhanie semop pri wait");
            return 1;
        }
        else return 0 ;
  }

 int main (int argc, char *argv[]){
          int i,j;
          int pid;

          key_t key = 1234; /* kluc pre volanie semget() */
          int semflg = IPC_CREAT | 0666; /* semflg pre volanie semget */
          int nsems = 1 /* nsems pre volanie semget */
          int nsops; /* počet operacii so semaforom */

          sops =  malloc(sizeof(struct sembuf););
```

```
        /* do sembuf sa ulozia argumenty volania semop */

       /* ziskanie sady semaforov */
  if ((semid = semget(key, 1, semflg)) == -1) {
      perror("semget: semget failed");
      return 1;}
    else
       fprintf(stderr, "semget: semget uspesne: semid = %d\n",
              semid);

    if (semctl(semid, 0,SETVAL,1)==-1)  /* inicializácia semafora */

         perror("Zlyhanie nastavenia hodnoty semafora\n");

    if ((pid = fork()) < 0) {    /* tvorba procesu potomka */
         perror("fork");
            return 1;
            }

    switch (pid)
     {
     case 0:    /* potomok */
            i = 0;
            while (i  < 3)
           {
              /* Pokus ziskat semafor, ak je obsadeny, bude cakat */
             if (j = wait_sem(0,getpid())>0)
                 perror("   semop: zlyhanie semop");
              else
              {
                fprintf(stderr, "\n\n   Potomkovy  proces %d
                        vchadza do KS: %d/3 krat\n",getpid(),
                sleep(5);
/* Nerobi nic 5 sekund  - tu sa vloží práca so zdieľaným prostriedkom, ku  ktorému
synchronizujeme prístup*/
              }

              if (j = signal_sem(0,getpid())>0)
                     perror("   semop: zlyhanie semop ");
                else {
                     fprintf(stderr, "    Proces potomok %d
                             opusta KS: %d/3 krat\n",getpid(),
               }
                ++i;
             }
             break;
    default: /* rodic */
            i = 0;
            while (i  < 3) { /* 3 krat zopakuje ziskanie a uvolnenie semafora */
```

```
                if (j = wait_sem(0,getpid()) >0) {
                    /* Volanie semop()*/
                   perror("* 1 semop: semop failed");
                  }
                  else
                  {
                    fprintf(stderr, "\n*  Process RODIC vchadza
                             do KS: %d/3 krat\n", i+1);
                    sleep(5);
/* Nerobi nic 5 sekund  - tu sa vloží práca so zdieľaným prostriedkom, ku  ktorému
synchronizujeme prístup*/

                    nsops = 1;

                    if (j = signal_sem(0, getpid())>0) {
                       perror("* semop: zlyhalnie semop ");
                    }
                    else
                        fprintf(stderr, "Proces RODIC opusta KS:
                        %d/3 krat\n", i+1);
                        sleep(5);
                  }
                ++i;
              }
          int s;
          s=wait(&status);
          /* Caka na ukoncenie potomka a az potom zrusi sadu semaforov */
          semctl(semid,0, IPC_RMID,0);
          break;
}
return 0;
}
```

7.8 Signály

Signály sa používajú pre upozornenie procesu alebo vlákna na určitú udalosť. Signály by sa mohli porovnať s hardvérovými prerušeniami, ktoré sa vyskytujú keď napríklad vstupno/výstupný systém generuje prerušenie pri ukončení V/V operácie. Táto udalosť spôsobí to, že riadenie sa odovzdáva obslužného programu prerušenia. Príslušný obslužný program sa nájde podľa čísla prerušenia.

Nie každý proces môže poslať inému procesu signál, môže len jadro a administrátor. Normálne používateľské procesy môže poslať signály procesom s rovnakým `UID` a `GID`, alebo procesom z ich skupiny.

Keď proces alebo vlákno dostane signál, riadenie sa odovzdá príslušnému handleru (toto je zaužívané pomenovanie obslužnej rutiny pre signály). Ktorý handler to bude závisí od nastaveniach procesu– buď obsluha bude štandardná alebo proces si ustanoví svoj handler (v závislostí od signálu).

Štandard POSIX., ktorý je implementovaný pre signály v Linux-e definuje rozhranie pre použitie signálov v kóde.

Výskyt signálu môže byť **synchrónny** alebo **asynchrónny** k behu procesu alebo vlákna, v závislostí od zdroja signálu a príčiny jeho vyslania.

Synchrónny signál sa vyskytuje ako dôsledok vykonávania toku inštrukcií pri ktorom sa vyskytne neopraviteľná chyba ako napr. ilegálna inštrukcia alebo ilegálny odkaz na pamäť. Takáto udalosť vyžaduje okamžite ukončenie procesu. Synchrónnym signálom sa v anglicky písanej literatúre často hovorí aj trap (lapač), pretože riadenie sa odovzdáva „trap handlera“ v jadre.

Asynchrónny signál je externý (niekedy nesúvisí) s momentálnym kontextom vykonávania procesu. Týmto signálom sa niekedy hovorí prerušenia.

Každý signál má svoje unikátne meno, ktoré začína skratkou SIG (napr. SIGINT pre prerušovací signál) a odpovedajúcim číslom. Tieto mená sa nachádzajú v `<signal.h>`. Navyše pre každý signál systém definuje štandardnú akciu, ktorá sa vykoná pri výskyte signálu. Existuje 5 možných akcií:

- `Term`: proces je donútený sa ukončiť,
- `Core`: proces je donútený sa ukončiť a navyše sa vytvorí core súbor (dump pamäte),
- `Stop`: proces sa zastaví,
- `Ign`: signál je ignorovaný, nevykoná sa nič pri jeho výskyte.
- `Cont`: štandardná akcia je že proces pokračuje, ak bol zastavený.

Akcia, ktorá sa vykoná při výskyte signálu je definovaná v kontexte procesu. Všetky vlákna a LWP procesy v rámci jedného procesu zdieľajú toto nastavenie. Jednotlivé vlákna nemôžu mať rozdielne nastavenia.

Tabuľka 7.1: Zoznam najdôležitejších signálov definovaných v štandarde POSIX.1 a ich význam

Signál	**Hodnota**	**Akcia**	**Popis**
SIGHUP	1	Term	„Zavesenie“ zístené z riadiacého terminálu alebo zrušenie riadiaceho procesu
SIGINT	2	Term	Prerušenie z klávesnice
SIGQUIT	3	Core	Quit z klávesnice
SIGILL	4	Core	Ilegálna inštrukcia
SIGABRT	6	Core	Signál Abort z abort(3)

SIGFPE	8	Core	Výnimka „Plavajúca čiarka“
SIGKILL	9	Term	Signál Kill
SIGSEGV	11	Core	Neplatný odkaz na pamäť
SIGPIPE	13	Term	Prerušená rúra : zápis do rúry, ktorú nikto nečíta
SIGALRM	14	Term	Signál Timer z alarm(2)
SIGTERM	15	Term	Signál na ukončenie (Termination)
SIGUSR1	30,10,16	Term	Používateľom definovaný signál 1
SIGUSR2	31,12,17	Term	Používateľom definovaný signál 2
SIGCHLD	20,17,18	Ign	Potomok zastavený alebo ukončený
SIGCONT	19,18,25	Cont	Pokračovanie, ak proces bol zastavený
SIGSTOP	17,19,23	Stop	Stop procesu
SIGTSTP	18,20,24	Stop	Stop zadány z tty
SIGTTIN	21,21,26	Stop	Vstup z tty pre proces na pozadí
SIGTTOU	22,22,27	Stop	Výstup z tty pre proces na pozadí

Akcia, ktorá sa vykoná pri zachytení signálu sa môže zmeniť a proces môže zaistiť zachytenie signálu a jeho obslúženie vlastnou obslužnou rutinou alebo opačne, proces môže ignorovať signál, ktorého štandardná akcia nie je `Ign`. Jedinými výnimkami sú signály SIGKILL a SIGSTOP, pretože ich štandardné akcie sa nedajú zmeniť. Interfejs pre definovanie a zmenu štandardnej obsluhy signálu sú knižnice `signal` a `sigset` a systémové volania `sigaction()` a `signal()`.

Signály môžu byť aj blokované (anglický termín - pending), čo znamená že proces dočasne zakazuje doručenie signálu. Generovanie takého signálu bude blokované kým sa neodblokuje, alebo akcia sa zmení na `Ign`.

Systémové volanie `sigprocmask` nastaví alebo získa signálnu masku procesu. Táto maska je bitové pole, ktorého jadro skúma, aby zistilo či je signál blokovaný alebo nie. `thr_setsigmask` a `pthread_sigmask` sú ekvivalentným interfejsom pre nastavenie a získanie nastavenia pre vlákna na úrovni používateľa (opak vlákien implementovaných na úrovni jadra).

Signál môže pochádzať z rôznych zdrojov a môže mať rôzne príčiny. Napr.

- `SIGHUP`, `SIGINT` a `SIGQUIT` sú generované z klávesnice**,**
- `SIGINT` a `SIGHUP` sú generované pri odpojení terminálu,
- `SIGSTOP`, `SIGTTIN`, `SIGTTOU` a `SIGTSTP` - sú orientované tiež na terminálový vstup/výstup.

Pre signály, pochádzajúce z príkazov z klávesnice príslušná postupnosť klavies, ktoré generujú daný signál je nadefinovaná v parametroch terminálu a dá sa zistiť príkazom `stty`:

```
$ stty -a
speed 38400 baud; rows 42; columns 84; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol =
<undef>; eol2 = <undef>;
swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr
icrnl ixon -ixoff -iuclc
-ixany -imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0
tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -
tostop -echoprt echoctl
echoke
```

Napríklad bežne používaná kombinácia `CTRL/C` vlastne generuje signál SIGINT a jeho štandardná akcia je `Exit`.

7.8.1 Zmena reakcie na zachytenie signálu

Proces môže zmeniť štandardnú reakciu na zachytenie signálu pomocou systémového volania `sigaction()` alebo pomocou menej portabilného volania `signal()`. Pomocou týchto volaní, proces si môže vybrať svoje chovanie pri zachytená signálu:

- vykonanie štandardnej reakcie na signál,
- ignorovanie signálu,
- zachytenie a obslúženie signálu vo vlastnom handleri, ktorý za vyvolá automaticky po zachytení signálu.

7.8.1.1 Systémové volanie `sigaction()` - robustné signálové rozhranie

Funkcia `sigaction()` je doporučovaná ako modernejšia a robustnejšia jako je funkcia `signál()`, ktorá bude popísaná neskôr.

```
#include <signal.h>

int  sigaction(int  signum,  const struct sigaction *act,
               struct sigaction*oldact);
```

Táto funkcia sa používa na zmenu správania procesu pri zachytení určitého signálu. Argumenty sú:

- `signum` - špecifikuje signál. Môže byť ľubovoľný platný signál okrem `SIGKILL` a `SIGSTOP`,
- `act` - ukazovateľ na štruktúru typu sigaction. Ak nie je `NULL`, nainštaluje sa uvedený handler,
- `oldact` - predchádzajúci handler. Ak nie je `NULL`, jeho ukazovateľ sa uchová v tomto argumente.

  Štruktúra `sigaction` je definovaná podobným spôsobom ako je uvedene ďalej:

```
struct sigaction {
      void (*sa_handler)(int);
      sigset_t sa_mask;
      int sa_flags;
    }
```

- `sa_handler` špecifikuje akciu, ktorá sa vykoná pri zachytení signálu s číslom `signum`. Môže byť:
  - `SIG_DFL` - pre štandardnú akciu,
  - `SIG_IGN` - pre ignorovanie signálu,
  - ukazovateľ na handler. Handler dostáva číslo signálu, ktoré je jeho jediným argumentom.
- Ak `SA_SIGINFO` je špecifikovaný v `sa_flags`, potom `sa_sigaction` (namiesto `sa_handler`) špecifikuje obslužnú funkciu pre `signum`. Funkcia získava číslo signálu ako prvý argument, ukazovateľ na `sig_info_t` ako druhý argument a tretý argument je ukazovateľ na `ucontext_t` (pretypovaný na `void`).
- `sa_mask` - predstavuje masku signálov, ktoré majú byť blokované počas vykonania handlera signálu. Navyše signál, ktorý spustil handler bude blokovaný pokiaľ nie je použitý príznak `SA_NODEFER`.
- `sa_flags` - špecifikuje sadu príznakov, ktoré modifikujú obsluhu signálu. Pozostáva z výsledku bitovej OR operácie medzi žiadnym alebo viacerými z týchto príznakov: `SA_NOCLDSTOP`, `SA_NOCLDWAIT`, `SA_RESETHAND`, `SA_ONSTACK`, `SA_RESTART`, `SA_NODEFER`, `SA_SIGINFO`. Podrobnejší popis čitateľ nájde v manuálových stránkach svojo systému.

Príklad 7.16: Použitie funkcie `sigaction()`

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

```
#include <unistd.h>

void Obsluha(int sig)
{
    printf("Dostal som signal c.%d\n", sig);
}

int main()
{
    struct sigaction act; // definicia štruktúry sigaction

    act.sa_handler = Obsluha; //obslužná funkcia
    sigemptyset(&act.sa_mask);//
    act.sa_flags = 0;

    sigaction(SIGINT, &act, 0);

    while(1) {
       printf("Ahoj, zijem!!!\n");
       sleep(1);
    }
}
```

Tento program sa vykonáva v nekonečnej slučke, pretože po každom zachytení signálu č.2 (`SIGINT`) sa jeho obsluha nastaví znova. Ukončenie je možné vygenerovaným signálu `SIGQIUT` pomocou klávesovej skratky CTRL/\).

V tomto príklade je použité volanie `sigemptyset(),` ktoré inicializuje sadu signálov, zadanú v jej argumente tak, že vylúči všetky signály zo sady. Ďalšie podobné funkcie pre prácu s všetkými signálmi sú:

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signum);

int sigdelset(sigset_t *set, int signum);

int sigismember(const sigset_t *set, int signum);
```

Tieto funkcie sú využívané funkciou `sigaction()` a ďalšími, pracujúcimi so signálmi.

- `sigemptyset(-` inicializuje sadu signálov, zadanú v jej argumente tak, že vyprázdni všetky signály zo sady.

- `sigfillset()` - inicializuje sadu signálov, zadanú v jej argumente tak, že naplní všetky signály.
- `sigaddset()` - pridá zadaný signál do sady.
- `sigdelset()` - vymaže zadaný signál do sady.
- `sigismember()`- zisťuje či daný signál je členom množiny signálov. Ak je, vráti 0, ináč –1.

Príklad 7.17: Zasielanie signálu medzi rodičom a potomkom

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

sig_atomic_t child_exit_status;

void cakanie_na_potomka (int signal_number)
{
  int status;
  wait (&status);   // čaka na ukončenie potomka
   /* Uloži stav ukončenia do globálnej premennej .  */
  child_exit_status = status;
}

int main ()
{
 /* Obslúži SIGCHLD volaním cakanie_na_potomka.  */
  struct sigaction sigchld_action;
  memset (&sigchld_action, 0, sizeof (sigchld_action));
  sigchld_action.sa_handler = & cakanie_na_potomka;
  sigaction (SIGCHLD, &sigchld_action, NULL);

   switch pid=fork(){
         case –1: printf(“Chyba forku”);
                  break;
         case  0: for (int i=1;i< 50;i++)
                      printf(“Pracuje potomok\n”);
                  break;
         default: cakanie_na_potomka(SIGCHLD);
                  printf(potomok skoncil so stavom %d“,
                         child_exit_status);
   }

  return 0;
}
```

7.8.1.2 Systémové volanie `signal()`

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

Systémové volanie `signal()` inštaluje nový handler pre signál so zadaným číslom. Argumenty volania sú:

- `signum` - číslo signálu pre ktorý sa inštaluje nový handler,
- `handler` - adresa funkcie nového handlera. Handler môže byť používateľom špecifikovaná funkcia alebo SIG_IGN (ignorovanie signálu) alebo SIG_DFL (štandardná obsluha). Použitie handlera pre obsluhu signálu sa nazýva „zachytenie“ signálu.

7.8.1.3 Systémové volanie `kill()`

Funkcia `kill()` sa používa na zaslanie signálu jednému alebo skupine procesov.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Argumenty volanie sú:

- `pid` - číslo signálu. Ak:
  - `pid` > 0 - signál je zaslaný procesu s číslom `pid`,
  - `pid` = 0 - signál je zaslaný každému procesu zo skupiny volajúceho procesu,
  - `pid` = -1 - signál je zaslaný každému procesu ktorému volajúci process má právo zasielať signal, okrem procesu 1 (init).
  - `pid` < -1 - signál je zaslaný každému procesu zo skupiny procesov s číslom `-pid`.
- `sig` - ak je 0, signál sa neposiela, ale vykoná sa chybová kontrola.

Príklad 7.18 : Zaslanie SIGUSR1 pribuznému procesu

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

int mpid,spid;
void handler(int a)
{
    signal(SIGUSR1, handler);
```

```
    printf("     HANDLER\n");
    printf("     Zachytil som signal USR1\n");
    printf("     Obsluhujem proces %d\n", getpid());
}

int main ()
{
    int im status;
    signal(SIGUSR1,handler);
    mpid = getpid ();
    printf("RODIC %d\n",mpid);
    spid = fork();

    switch (spid){
     case 0:
               printf("           POTOMOK %d\n",getpid());
               printf("           Vysielam otcovi USR1\n");
               kill(mpid,SIGUSR1);
               break;

     default: /* rodic */
              printf("Cakam na ukoncenie potomka\n");
              wait(&status);
  }
return 0;
}
```

7.8.1.4 Zaslanie signálov z príkazového riadku

Príkaz bash-u `kill` je obdoba systémového volania `kill()`.

```
kill [ -signal | -s signal ] pid
```

Číslo signálu môžeme zadať číslom alebo symbolom. Napr.

```
kill –9 PID
kill –SIGKILL PID   // používa štandardné pomenovanie signálu
kill –KILL PID      // používa štandardné pomenovanie signálu bez predpony SIG
```

Príklaz `kill –l` vypíše všetky signály, ktoré sa dajú použiť s príkazom `kill`.

Príklad 7.19: Použitie príkazu `kill`

```
$ ps            // zístime ktoré procesy bežia
 PID TTY          TIME CMD
4026 pts/1    00:00:00 bash
```

```
 4608 pts/1    00:00:00 man
 4614 pts/1    00:00:00 nroff
 4615 pts/1    00:00:00 pager
 4618 pts/1    00:00:00 ps

 $ kill -9 4608  // zrušenie procesu s týmto PID

 $ ps            // znova  zístime ktoré procesy bežia

  PID TTY          TIME CMD
 4026 pts/1    00:00:00 bash
 4619 pts/1    00:00:00 ps
[1]+  Killed                  man kill
```

Klávesové skrátky pre poslanie niektorých signálov z klávesnice:

CTRL\C - SIGINT
CTRL\- - SIGQUIT

7.9 Úlohy a zadania

1. V príklade 7.15 skúste zmeniť nastavenie semaforovej operácie pomocou príznaku `IPC_NOWAIT` a pozorujte zmenu priebehu procesov.
2. Implementujte pomocou semaforov a zdieľanej pamäte klasickú synchronizačnú úlohu procucent-konzument.