Meziprocesová komunikace

mnohé aplikace sestávají z mnoha navzájem spolupracujících procesů, které mezi sebou komunikují a sdílejí informace

jádro musí poskytovat mechanizmy, které toto umožní

nazýváme je prostředky meziprocesové komunikace

jejich účelem je

- přenos údajů
- sdílení dat
- oznámení vzniku událostí
- sdílení prostředků
- sledování a řízení běhu procesu, např. při ladění programů

signály

umožňují oznámit procesům asynchronní události

roury (*pipes*)

umožňuje zapisovat data na konec roury a číst je ze začátku

nepojmenované roury

vytvářejí se systémovým voláním **`pipe()`**, které vrátí dva deskriptory jeden pro čtení a druhý pro zápis

deskriptory roury jsou při vytváření procesů děděné, do roury může zapisovat a číst z ní více procesů, přičemž data jsou čtena v pořadí v jakém byla zapsána

procesy můžou komunikovat prostřednictvím roury byla-li vytvořena společným předchůdcem, po skončení všech procesů roura přestává existovat

pojmenované roury, FIFO soubory

jsou perzistentní, existují jako soubory i když je nepoužívají žádné procesy

FIFO musí být explicitně zrušen, jako obyčejné soubory - **`unlink`**

na rozdíl od obyčejných souborů přečtená data jsou odstraněna a z pohledu komunikace mají stejnou sémantiku jako nepojmenované roury

vytvoření FIFO souboru

`mknode(cesta, mód, zařízení)`

`mkfifo(cesta, mód)`

`mód` obsahuje obvyklá oprávnění

v případě **mknode** obsahuje disjunkci **S_IFIFO** a oprávnění

třetí parametr slouží pro vytváření speciálních souborů pro zařízení

```
dev_t dev;
int   status;
...
status  = mknod("/home/cnd/mod_done",
S_IFIFO | S_IWUSR |
    S_IRUSR | S_IRGRP | S_IROTH, dev);
```

```
int status;
...
status = mkfifo("/home/cnd/mod_done",
S_IWUSR | S_IRUSR | S_IRGRP   | S_IROTH);
```

FIFO jsou potom jako obvykle otevřena systémovým voláním **open(),** které vrátí deskriptor souboru a do FIFO můžeme zapisovat systémovým voláním **write()** anebo z něho číst systémovým voláním **read()**

sledování procesů

`ptrace(příkaz, pid, adr, data);`

- umožňuje sledovat a řídit běh procesu **`pid`**
- **`příkaz == 0,`** slouží na informování jádra, že proces je sledován (**`PTRACE_TRACEME`**)
ostatní příkazy používá sledující proces na řízení vykonávání sledovaného procesu

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>  /* pro ORIG_EAX */

int main()
{   pid_t child;
    long orig_eax;
    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "ls", NULL);
    }
    else {
        wait(NULL);
        orig_eax = ptrace(PTRACE_PEEKUSER,
                          child, 4 * ORIG_EAX,
                          NULL);
        printf("Potomek volal "
               "sluzbu %ld\n", orig_eax);
        ptrace(PTRACE_CONT, child, NULL, NULL);
    }
    return 0;
}
```

struktura ladícího programu

```
...
if ((pid = fork()) == 0)
{
    /*potomek – sledovaný proces*/
    ptrace(0,0,0,0);
    exec("jméno sledovaného programu");
}
/*tady pokračuje ladící proces*/
for(;;)
{
    wait((int *) 0);
    read(vstup ladícího příkazu);
    ptrace(příkaz,...);
    if(konec ladění)
        break;
}
```

`ptrace` potomkovi nastaví trace bit v deskriptoru potomka

potomek vykoná **`exec`**, jádro zjistí, že trace bit je nastaven a pošle potomkovi TRAP signál

při návratu z **`exec`** jádro obslouží TRAP signál, který poslalo

vzbudí rodiče a potomek přejde do stavu trace (obdoba sleep)

rodič zadává systémovým voláním **`ptrace()`** požadované příkazy

jádro vzbudí sledovaného potomka, ladící proces se uspí, potomek vykoná příkaz a vzbudí ladící proces

Systém V IPC

předcházející mechanizmy meziprocesové komunikace nejsou pro mnohé aplikace postačující

System V poskytl tři mechanizmy (objekty)

- semafory (*semaphores*)
- fronty zpráv (*message queues*)
- sdílenou paměť (*shared memory*)

postupně byly implementovány v dalších systémech - BSD, Linux, Solaris

uvedené mechanizmy mají podobné uživatelské rozhraní a podobnou implementaci

- existují na jednom počítači
- žijí tak dlouho jako jádro (reboot)
- jsou identifikovány IPC ***klíčem*** (obdoba cesty k souboru)
- jsou spřístupňovány ***identifikátory*** (obdoba deskriptoru souboru)
- identifikátory nejsou vázány na proces, nemění se v průběhu života objektu
- nemají i-uzly, nemůžeme použít **`open, unlink, stat, read, write`**

proces získa identifikátor IPC prostředku voláním

`semget()`
`msgget()`
`shmget()`

prvním parametrem **`xxxget`** je klíč, kterým procesy identifikují prostředek a uvedené funkce vrátí identifikátor prostředku, který procesy dál používají pro přístup k prostředku

identifikátor může proces dále získat jako argument **`exec`**, posláním zprávou, přečtením ze souboru

každý mechanizmus má tabulku jejíž položky obsahují všechny jeho prostředky

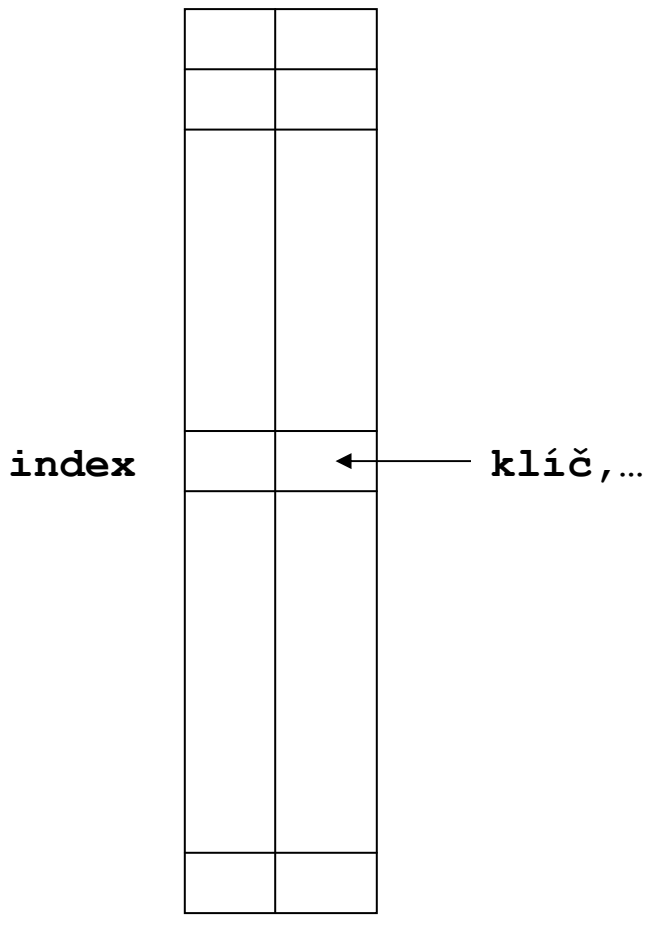
každá položka obsahuje klíč a pořadové číslo použití položky

identifikátor prostředku, vzhledem ke své perzistentnosti, není přímo index do tabulky, vypočte se podle vztahu

```
id = seq * velikost_tabulky + index
```

naopak, jádro z identifikátoru prostředku, který je parameterem dalších systémových volání, určí index prostředku v tabulce podle vztahu

```
index = id % velikost_tabulky
```



seq je inicalizován na nulu a inkrementován po každém uvolnění položky tabulky

Příklad

```
velikost: 100
index: 2
seq:     0      1      2      3   ...
id :     2    102    202    302   ...
```

společné data IPC (semafor, fronta zpráv, sdílená paměť) jsou uložena v záznamu **`ipc_perm`**, který obsahuje položky:

`key` - klíč, 32 bitová hodnota zadaná uživatelem, který identifikuje konkrétní prostředek
`uid` - uživatelský ID vlastníka prostředku
`gid` - skupinový ID vlastníka prostředku
`cuid` - uživatelský ID tvůrce prostředku
`cgid` - skupinový ID tvůrce prostředku
`mode` - oprávnění rwx pro vlastníka, skupinu a ostatních
`seq` - pořadové číslo použití položky tabulky

jsou uloženy položkách tabulky každého mechanizmu spolu s daty specifickými pro typ prostředku

správa klíčů mezi aplikacemi není, klíč můžeme vytvořit voláním **`ftok`** z cesty k souboru

```
key_t ftok(
    const char *cesta, int id
);
```

pro jednu cestu může generovat různé klíče pro různá id

hodnota **`IPC_PRIVATE`** parametru klíč funkcí **`xxxget`** zajistí vytvoření nového prostředku

dalším společným parametrem funkcí **`xxxget`** je parametr příznaky

příznak **`IPC_CREAT`** způsobí vytvoření prostředku jádrem, pokud ještě neexistuje

ve spojení s příznakem **`IPC_EXCL`** jádro oznámí chybu jestliže prostředek se zadaným klíčem existuje

příkazy **`IPC_SET`** a **`IPC_STAT`** funkcí **`semctl()`, `msgctl()`** a **`shmctl()`** umožňují nastavit a zjistit stavové informace prostředků

příkaz **`IPC_RMID`** v **`xxxctl`** odstraní IPC prostředek

Semafory

```
semid = semget (klíč, počet, příznak);
```

vrátí identidikátor pole semaforů o velikosti **počet**

```
stav = semop(semid, sops, nsops);
```

atomicky vykoná operace nad polem semaforů

sops je ukazatel na pole s **nsops** prvky typu **sembuf**

```
struct sembuf {
    unsigned short sem_num;
    short sem_op;
    short sem_flg;
};
```

specifikují operaci **sem_op** nad semaforem s indexem **sem_num**

`sem_op < 0` je-li hodnota semaforu větší nebo rovná absolutní hodnotě **`sem_op`,** absolutní hodnota **`sem_op`**, se odečte od hodnoty semaforu
je-li menší, proces je blokován (spící) dokud není zvýšena hodnota semaforu

`sem_op > 0` zvýší se hodnota semaforu o hodnotu **`sem_op`** a vzbudí se procesy čekající na její zvýšení

`sem_op = 0` proces je blokován dokud hodnota semaforu není 0

jde tedy o zobecněný semafor

je-li proces spící uprostřed operace, je spící na přerušitelné úrovni

```
#define SEMKEY  52
int semid;
struct sembuf psembuf, vsembuf;

short init[2];

/*inicializace*/

semid = semget(SEMKEY,2,IPC_CREAT |
    S_IRUSR | S_IWUSR | S_IRGRP |
    S_IWGRP | S_IROTH | S_IWOTH);

init[0] = init[1] = 1;
semctl(semid,2,SETALL,init);

/*P operace*/
psembuf.sem_op = -1;
psembuf.sem_flg = 0;

/*V operace*/
vsembuf.sem_op = 1;
vsembuf.sem_flg = 0;

/*proces 1*/
...
psembuf.sem_num = 0;
semop(semid, &psembuf, 1);
psembuf.sem_num = 1;
semop(semid, &psembuf, 1);
...
vsembuf.sem_num = 1;
semop(semid, &vsembuf, 1);
vsembuf.sem_num = 0;
semop(semid, &vsembuf, 1);
```

```
/*proces 2*/
...
psembuf.sem_num = 1;
semop(semid, &psembuf, 1);
psembuf.sem_num = 0;
semop(semid, &psembuf, 1);
...
```

možnost uvíznutí

řešení

```
struct sembuf psembuf[2];

psembuf[0].sem_num = 0;
psembuf[1].sem_num = 1;
psembuf[0].sem_op = -1;
psembuf[1].sem_op = -1;
semop(semid,psembuf,2);
```

v příznacích operace **semop** může proces nastavit příznak **IPC_NOWAIT,** jádro namísto blokování vrátí chybu

když po zamknutí přístupu k prostředku proces skončí, další procesy ho nemůžou získat, příznak **SEM_UNDO** v operaci **semop** způsobí, že jádro anuluje vykonané operace

implementace (Linux)

jednotlivý semafor implementuje záznam **`sem`** obsahující položky

`semval -` hodnota počítadla
`sempid -` PID posledního procesu pracujícího se semaforem

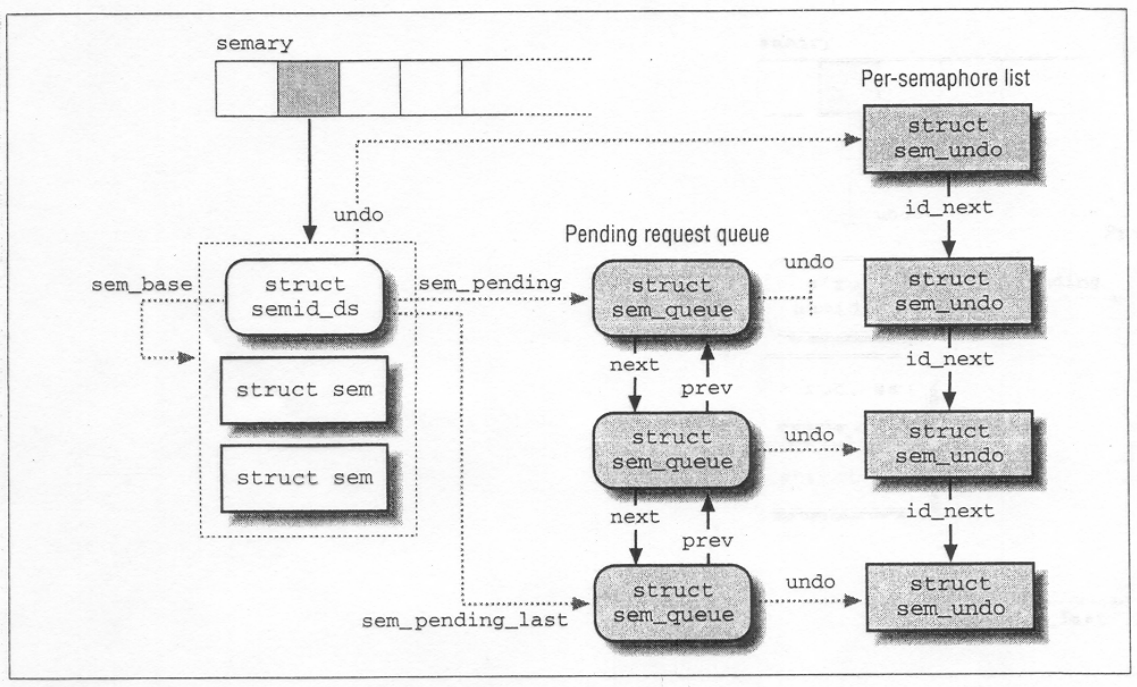
položky tabulky semaforů jsou záznamy **`semid_ds`** obsahující položky

`sem_perm - ipc_perm`
`sem_otime -` čas poslední operace
`sem_base -` ukazatel na první **`sem`** záznam
`sem_pending -` nevyřízené operace
`sem_pending_last -` poslední nevyřízená operace
`undo -` seznam operací na anulování
`sem_nsems -` počet semaforů v poli

seznam všech anulovatelných operací se udržuje:

pro proces jehož prvky obsahují upravující hodnotu (adjustment value) pro jednotlivé semafory, s kterými proces pracuje – je použit když proces skončí

pro semafor jenž se použije při nastavení hodnoty funkcí **`semctl()`** a při odstranění (zrušení) semaforu



IPC semaphore data structures

Zdroj: Bovet P.D., Cesati M.C.: Understanding the LINUX KERNEL, O'REILLY 2001

Fronty zpráv

procesy komunikují prostřednictvím zpráv

zpráva vytvořená procesem je zaslána do fronty zpráv dokud ji jiný proces nepřečte

zpráva obsahuje 32 bitový typ zprávy a data zprávy

typ zprávy umožňuje selektivně vybírat zprávy z fronty

proces získá nebo vytvoří frontu zpráv voláním
`msgqid = msgget(klíč, příznak);`

zpráva se uloží do fronty voláním
`msgsnd(msgqid, msgp, počet, příznak);`

`msgp` - je ukazatel na zprávu obsahující typ zprávy následován daty

`počet` - velikost zprávy včetně typu v bytech

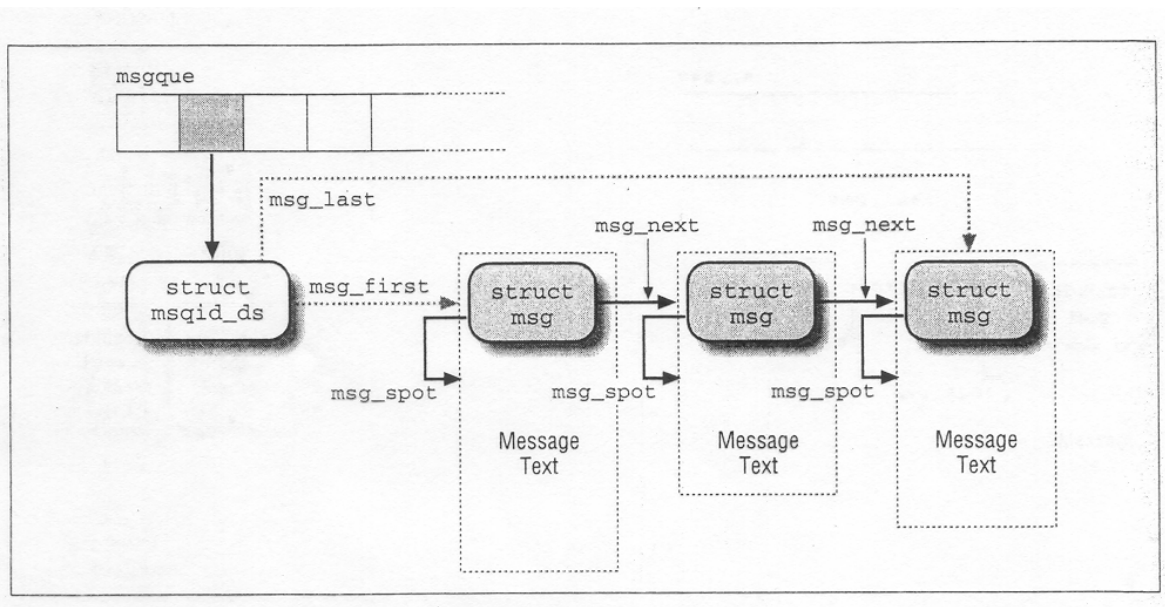
zprávy jsou ve frontě v pořadí jejich příchodu

zprávy jsou vybírány voláním
počet = msgrcv(msgqid, msgp, maxpct, msgtyp, příznak);

je-li čtená zpráva delší než **maxpct** zpráva je useknutá

msgp - je ukazatel na zprávu obsahující typ zprávy následován daty

- je-li **msgtyp** nulový, vráti se první zpráva z fronty
- je-li **msgtyp** kladný, vrátí první zprávu typu **msgtyp**
- je-li **msgtyp** záporný, vráti se první zpráva nejnižšího typu než je absolutní hodnota **msgtyp**



IPC message queue data structures

Zdroj: Bovet P.D., Cesati M.C.: Understanding the LINUX KERNEL, O'REILLY 2001

Sdílená paměť

sdílená paměť je oblast paměti, která je sdílená více procesy

proces sdílenou oblast pamětí vytvoří nebo ji získá voláním

```
shmid = shmget(klíč, velikost, příznak);
```

proces připojí oblast na virtuální adresu voláním

```
adr = shmat(shmid, shmadr, shmpříznak);
```

`shmadr` je návrh adresy pro připojení oblasti

`shmpříznak SHM_RND` způsobí zaokrouhlení adresy dolů

je-li **`shmadr`** nula, jádro vybere adresu

skutečná adresa je návratová hodnota

odpojení oblasti

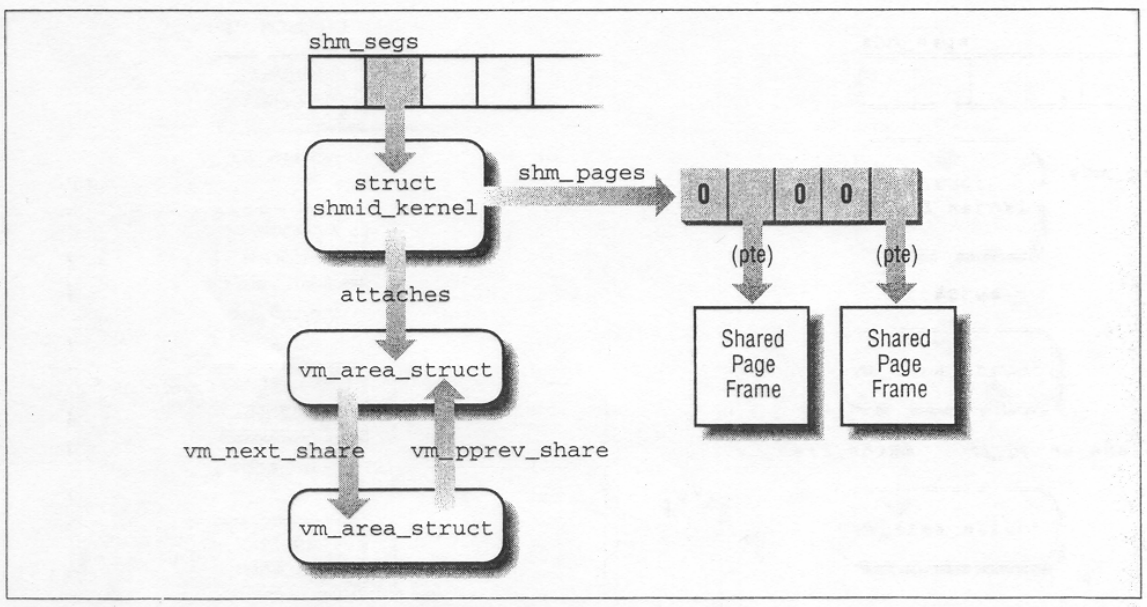
```
shmdt(shmaddr);
```

```c
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
  key_t key = 100;
  int shm_id;
  int shm_size = 1024;
  char *shm_addr;
  struct shmid_ds shm_buf;
  pid_t child_pid;

  /* vytvoreni sdilene pameti ke klici key */
  shm_id = shmget(key, shm_size, IPC_CREAT |
    IPC_EXCL | S_IRUSR | S_IWUSR);

child_pid = fork();
if (child_pid == 0) {

  /* pripojeni */
  shm_addr = (char *)shmat(shm_id, NULL, 0);
  printf("potomek: sdilena pamet pripojena na
      adresu: %p\n", shm_addr);

  printf("zapis ...\n");
  sprintf(shm_addr, "Sdilena pamet");

  /* odpojeni */
  shmdt(shm_addr);
}
```

```
else if (child_pid != -1) {
  shm_addr = (char *)shmat(shm_id, NULL,
     SHM_RDONLY);
  printf("rodic: sdilena pamet pripojena na
     adresu: %p\n", shm_addr);

  sleep(1);
  printf("cteni ...");
  printf(" %s\n", shm_addr);

  shmdt(shm_addr);

  waitpid(child_pid, NULL, 0);

  /* dealokovani*/
  shmctl(shm_id, IPC_RMID, NULL);
}
else return 1;

return 0;
}
```

Výstup:

```
potomek: sdilena pamet pripojena na adresu:
0xb7f8d000
zapis...
rodic: sdilena pamet pripojena na adresu:
0xb7f8d000
cteni: ... Sdilena pamet
```



IPC shared memory data structures

Zdroj: Bovet P.D., Cesati M.C.: Understanding the LINUX KERNEL, O'REILLY 2001

Method	100-byte msg	2000-byte msg	20,000-byte msg	100,000-byte msg
FIFO	S: 1.00 B: 1.00 D: 1.00 L: 1.00	S: 1.22 B: 1.46 D: 1.29 L: 1.51	too big	too big
System V message queue	S: 0.90 B: 0.62 L: 0.31	S: 1.82 B: 3.76 L: 0.64	too big	too big
POSIX message queue	S: 2.02	S: 2.40	S: 7.03	S: 33.39
System V shared memory	S: 1.47 B: 0.94 D: 1.04 L: 0.55	S: 1.41 B: 0.91 D: 1.07 L: 0.53	S: 1.55 B: 0.90 D: 1.02 L: 0.47	S: 1.24 B: 0.90 D: 1.06 L: 0.51
POSIX shared memory	S: 1.27	S: 1.25	S: 1.41	S: 1.37
Sockets	S: 1.84 B: 0.81 D: 1.04 L: 0.75	S: 2.15 B: 1.00 D: 1.27 L: 0.95	S: 10.15 B: 7.99 D: 5.52 L: 6.06	S: 44.13 B: 35.83 D: 25.86 L: 31.91

S: Solaris; B: FreeBSD; D: Darwin; L: Linux. All times normalized for each system so FIFO time for 100-byte messages is 1.00. Sockets used the `AF_UNIX` domain (see Chapter 8).

Zdroj: M.J. Rochkind, Advanced UNIX Programming