Laborator 05 - IPC

Materiale ajutătoare

- lab05-slides.pdf
- lab05-refcard.pdf

Nice to read

- TLPI - Chapter 51, Introduction to POSIX IPC
- TLPI - Chapter 52, POSIX messages queues
- TLPI - Chapter 53, POSIX semaphores
- TLPI - Chapter 54, POSIX shared memory

Mecanisme IPC

Comunicarea se poate realiza între proces de pe aceeaşi maşină sau de pe maşini diferite. Exemple de mecanimse IPC:

- obiecte de sincronizare (mutex, semafor)
- cozi de mesaje
- memorie partajată
- pipe-uri
- socketi

Pipe-urile au fost prezentate deja în laboratorul de procese. În acest laborator ne vom concentra pe primele trei categorii.

Mecanisme de sincronizare

- **Mutex** (Windows)

Este un obiect de sincronizare care poate fi deţinut (posedat, acaparat) doar de un singur proces/thread (în funcţie de implementare) la un moment dat. Drept urmare, operaţiile de baza cu mutex-uri sunt cele de obţinere şi de eliberare.

Odată obţinut de un proces/thread, un mutex devine indisponibil pentru orice alt proces/thread. Orice proces/thread care încearcă să acapareze un mutex indisponibil, se va bloca (un timp definit sau nu) aşteptând ca el să devină disponibil.

Mutex-urile sunt cel mai des folosite pentru a permite unui singur proces la un moment dat să acceseze o resursă.

- **Semafor** (Linux, Windows)

Semafoarele sunt resurse IPC folosite pentru sincronizarea între procese/thread-uri (e.g. pentru controlul accesului la resurse). Un semafor poate fi privit ca un contor ce poate fi incrementat şi decrementat, dar a cărui valoare nu poate scădea sub 0. Atât timp cât semaforul (contorul) are valori strict pozitive el este considerat disponibil. Când valoarea semaforului a ajuns la 0 el devine indisponibil şi următoarea încercare de decrementare va duce la o blocare a threadului/procesului de pe care s-a făcut apelul până când semaforul devine disponibil.

- **Cozi de mesaje** (Linux, Windows)

Sunt folosite de procese pentru a comunica între ele prin mesaje. Aceste mesaje îşi păstrează ordinea în interiorul cozii de mesaje. Sunt mecanisme de comunicare unidirecţionale atât pe Linux, cât şi pe Windows.

- **Memorie partajată** (Linux, Windows)

Acest mecanism permite comunicarea între procese prin accesul direct şi partajat la o zonă de memorie bine determinată. Este un mod mai rapid de comunicare între procese decât celelalte mijloace IPC, dar are un mare dezavantaj, procesele ce comunică trebuie să fie pe aceeaşi maşină (spre deosebire de socketi, pipe-urile cu nume şi cozile de mesaje din Windows)

Linux

Linux pune la dispoziţie 2 seturi de API-uri referitoare la mecanismele de comunicare interprocese, ce ţin de standarde diferite:

- System V Inter-Process Communication, derivat din distribuţia de Unix System V release 4 AT&T
- POSIX (Portable Operating System Interface for Unix)

Ambele standarde specifică 3 mecanisme:

- `mesaje` (messages) - realizează schimbul de mesaje cu orice proces sau server
- `semafoare` (semaphores) - realizează sincronizarea execuţiilor proceselor
- `memorie partajată` (shared memory) - realizează partajarea memoriei între procese

API-ul studiat in acest laborator este cel POSIX.

Obiectele de tip IPC pe care se concentrează laboratorul de faţă sunt gestionate global de sistem şi rămân în viaţă chiar dacă procesul creator moare. Faptul că aceste resurse sunt globale în sistem are implicaţii contradictorii:

- Dacă un proces se termină, datele plasate în obiecte IPC pot fi accesate ulterior de alte procese
- Pe de altă parte, procesul proprietar trebuie să se ocupe şi de dealocarea resurselor, altfel ele rămân în sistem până la ştergerea lor manuală sau până la repornirea sistemului.

Faptul că obiectele IPC sunt globale în sistem poate duce la apariţia unor probleme: cum numărul de mesaje care se afla în cozile de mesaje din sistem e limitat global, un proces care trimite multe asemenea mesaje poate bloca toate celelalte procese.

ATENŢIE Pentru folosirea API-ului trebuie să includeţi la linking biblioteca `rt` (-lrt).

Semafoare POSIX

Semafoarele sunt resurse IPC folosite pentru sincronizarea între procese (e.g. pentru controlul accesului la resurse). Operaţiile asupra unui semafor pot fi de *`setare`*, *`verificare`* a valorii (care poate fi >= 0), *`test and set`*. Un semafor poate fi privit ca un contor ce poate fi incrementat şi decrementat, dar a cărui valoare nu poate scădea sub 0.

Semafoarele POSIX sunt de 2 tipuri:

- `cu nume` - folosite în general pentru sincronizare între procese distincte;
- `fără nume` - ce pot fi folosite pentru sincronizarea între firele de execuţie ale aceluiaşi proces, sau între procese - cu condiţia ca semaforul să fie într-o zonă de memorie partajată.

În continuare vor fi luate în discuţie semafoarele `cu nume`. Diferenţele faţă de cele fără nume constă în funcţiile de creare şi distrugere, celelalte funcţii fiind identice.

- ambele tipuri de semafoare sunt reprezentate în cod prin tipul `sem_t`.
- semafoarele cu nume sunt idenficate la nivel de sistem printr-un şir de forma `"/nume"`.
- fişierele antet necesare sunt , şi .

Crearea şi deschiderea

Un proces poate crea sau deschide un semafor existent cu funcţia sem_open:

```
/* open semaphore */
sem_t* sem_open(const char *name, int oflag);

/* create semaphore */
sem_t* sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

Comportamentul este similar cu cel de la deschiderea fişierelor. Dacă flag-ul `O_CREAT` este prezent, trebuie folosită cea de-a doua formă a funcţiei, specificând permisiunile şi valoarea iniţială.

Singurele posibilităţi pentru al doilea argument sunt:

- 0 - se deschide semaforul dacă exista

- O_CREAT - se creează semaforul dacă nu exista; se deschide dacă exista
- O_CREAT | O_EXCL - se creează semaforul **numai** dacă nu exista; se întoarce eroare dacă exista

Decrementare, incrementare şi aflarea valorii

Valoarea unui semafor este decrementată cu funcţia sem_wait:

```
int sem_wait(sem_t *sem);
```

Dacă semaforul are valoarea 0, funcţia blochează până când un alt proces "deblochează" (incrementează) semaforul.

Pentru a încerca decrementarea unui semafor fără riscul de a rămâne blocat la acesta, un proces poate apela sem_trywait:

```
int sem_trywait(sem_t *sem);
```

În cazul în care semaforul are deja valoarea zero, funcţia va întoarce -1, iar `errno` va fi setat la `EAGAIN`.

Un semafor este incrementat cu funcţia sem_post:

```
int sem_post(sem_t *sem);
```

În cazul în care semaforul avea valoarea zero, un proces blocat în sem_wait pe acesta va fi deblocat.

Valoarea unui semafor (a contorului) se poate afla cu sem_getvalue:

```
int sem_getvalue(sem_t *sem, int *pvalue);
```

În cazul în care există procese blocate la semafor, implementarea apelului pe Linux va returna **zero** în valoarea referinţă de `pvalue`.

Închiderea şi distrugerea

Un proces închide (notifica faptul că nu mai foloseşte) un semafor printr-un apel sem_close:

```
int sem_close(sem_t *sem);
```

Un proces poate şterge un semafor printr-un apel sem_unlink:

```
int sem_unlink(const char *name);
```

Distrugerea efectiva a semaforului are loc după ce toate procesele care îl au deschis apelează sem_close sau se termină. Totuşi, chiar şi în acest caz, apelul sem_unlink nu se va bloca!

Exemplu de utilizare

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>

#include "utils.h"

#define SEM_NAME        "/my_semaphore"

int main(void)
{
        sem_t *my_sem;
        int rc, pvalue;

        /* create semaphore with initial value of 1 */
        my_sem = sem_open(SEM_NAME, O_CREAT, 0644, 1);
        DIE(my_sem == SEM_FAILED, "sem_open failed");


        /* get the semaphore */
        sem_wait(my_sem);

        /* do important stuff protected by the semaphore */
        rc = sem_getvalue(my_sem, &pvalue);
        DIE(rc == -1, "sem_getvalue");
        <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("sem is %d\n", pvalue);

        /* release the lock */
        sem_post(my_sem);

        rc = sem_close(my_sem);
        DIE(rc == -1, "sem_close");
```

```
        rc = sem_unlink(SEM_NAME);
        DIE(rc == -1, "sem_unlink");

        return 0;
}
```

Semaforul va fi creat în "/dev/shm" şi va avea numele "sem.my_semaphore"

Cozi de mesaje

Acestea permit proceselor interschimbare de date sub formă de mesaje

- la nivel de sistem sunt indentificabile printr-un string de forma "`/nume`".
- la nivel codului, o coada de mesaje este reprezentată de un descriptor de tipul `mqd_t`.
- fişierele antet necesare sunt , şi .

Crearea şi deschiderea

Functiile de creare şi deschidere sunt similare ca forma şi semantică celor de la semafoare (mq_open):

```
/* open */
mqd_t mq_open(const char *name, int oflag);

/* create */
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
```

În funcţie de flag-uri (unul din cele de mai jos trebuie specificat), coada poate fi deschisă pentru:

- recepţionare (`O_RDONLY`)
- trimitere (`O_WRONLY`)
- recepţionare şi trimitere (`O_RDWR`)

Daca `attr` e `NULL`, coada va fi creată cu atribute implicite. Structura `mq_attr` arată astfel:

```
struct mq_attr {
    long mq_flags;       /* 0 or O_NONBLOCK */
    long mq_maxmsg;      /* Max. number of messages on queue */
    long mq_msgsize;     /* Max. message size (bytes) */
    long mq_curmsgs;     /* number of messages currently in queue */
};
```

Trimiterea şi recepţionarea de mesaje

Pentru a trimite un mesaj (de lungime cunoscută, stocat într-un buffer) în coadă se apelează mq_send:

```
mqd_t mq_send(mqd_t mqdes, const char *buffer, size_t length, unsigned priority);
```

Mesajele sunt ţinute în coadă în ordinea descrescătoare a priorităţii.

În cazul în care coada este plină, apelul blochează. Dacă este o coadă non-blocantă (`O_NONBLOCK`), funcţia va întoarce -1, iar `errno` va fi setat la `EAGAIN`.

Pentru a primi un mesaj dintr-o coadă (şi anume: cel mai vechi mesaj cu cea mai mare prioritate) se foloseşte mq_receive:

```
ssize_t mq_receive(mqd_t mqdes, char *buffer, size_t length, unsigned *priority);
```

Daca `priority` este non-NULL, zona de memorie către care face referire va reţine prioritatea mesajului extras.

În cazul în care coada este vidă, apelul blochează. Dacă este o coadă non-blocantă (`O_NONBLOCK`), comportamentul este similar cu cel al mq_send.

ATENŢIE La primirea unui mesaj, lungimea buffer-ului trebuie să fie **cel puţin egală** cu dimensiunea maximă a mesajelor pentru coada respectivă, iar la trimitere **cel mult egală**. Dimensiunea maximă implicită se poate afla pe Linux din `/proc/sys/kernel/msgmax`.

Închiderea şi ştergerea

Închiderea (eliberarea "referinţei") unei cozi este posibilă prin apelul mq_close:

```
mqd_t mq_close(mqd_t mqdes);
```

Ştergerea se realizează cu un apel mq_unlink:

```
mqd_t mq_unlink(const char *name);
```

Semantica este similară cu cea de la semafoare: coada nu va fi ştearsă efectiv decât după ce restul proceselor implicate o închid.

Exemplu de utilizare

mqueue.c

```
#include <mqueue.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

#include "utils.h"

/* Set buffer size at least the default maxim size of the queue
 * found in/proc/sys/kernel/msgmax */
#define BUF_SIZE        (1<<13)
#define TEXT            "test message"
#define NAME            "/test_queue"

char buf[BUF_SIZE];

int main(int argc, char **argv)
{
        unsigned int prio = 10;
        int rc;
        mqd_t m;

        m = mq_open(NAME, (argc>1 ? O_CREAT : 0) | O_RDWR, 0666, NULL);
        DIE(m == (mqd_t)-1, "mq_open");

        if (argc > 1) {
                /* server sending message */

                rc = mq_send(m, TEXT, strlen(TEXT), prio);
                DIE(rc == -1, "mq_send");

                rc = mq_close(m);
                DIE(rc == -1, "mq_close");

        } else {
                /* client receiving message */

                rc = mq_receive(m, buf, BUF_SIZE, &prio);
                DIE(rc == -1, "mq_recvieve");

                                                                                                    <a
href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("received: %s\n", buf);

                rc = mq_close(m);
                DIE(rc == -1, "mq_close");

                rc = mq_unlink(NAME);
                DIE(rc == -1, "mq_unlink");
        }

        return 0;
}
```

Memoria partajată

Acest mecanism permite comunicarea între procese prin accesul direct şi partajat la o zonă de memorie bine determinată.

- la nivelul sistemului, o zonă este identificată printr-un string de forma "`/nume`";
- la nivelul codului, o zonă este reprezentată printr-un file descriptor (`int`).
- fişierele antet necesare sunt , şi .

Crearea şi deschiderea

Apelul de creare/deschidere este similar cu semantica apelului open pentru fişiere "obişnuite":

```
int shm_open(const char *name, int flags, mode_t mode);
```

Ca flag de acces trebuie specificat fie `O_RDONLY`, fie `O_RDWR`.

Redimensionarea

O zonă de memorie partajată nou creată are dimensiunea iniţială zero. Pentru a o dimensiona se foloseşte ftruncate:

```
int ftruncate(int fd, off_t length);
```

Maparea şi eliberarea

Pentru a putea utiliza o zonă de memorie partajată după deschidere, aceasta trebuie mapată în spaţiul de memorie al procesului. Maparea se realizează printr-un apel mmap:

```
void *mmap(void *address, size_t length, int protection, int flags, int fd, off_t offset);
```

Valoarea întoarsă reprezintă un pointer către începutul zonei de memorie sau `MAP_FAILED` în caz de eşec.
Acest apel are o largă aplicabilitate şi va fi discutat în cadrul laboratorului de memorie virtuală. Momentan, pentru a mapa întregul conţinut al unei zone (`shm_fd`) de dimensiune cunoscută (`shm_len`), recomandăm folosirea apelului:

```
mem = mmap(0, shm_len, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

Când maparea nu mai este necesară, prin apelul munmap se realiează demaparea:

```
int munmap(void *address, size_t length);
```

Închiderea şi ştergerea

Închiderea unei zone de memorie partajată este identică cu închiderea unui fişier - apelul close.

Odată ce o zonă de memorie a fost demapată şi închisă în toate procesele implicate, se poate şterge prin shm_unlink:

```
int shm_unlink(const char *name);
```

Semantica este identică cu cea de la funcţiile `*_unlink` anterioare - ştergerea efectivă este amânată până ce toate procesele implicate închid zona în cauză.

Exemplu de utilizare

```
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */
#include <unistd.h>

#include "utils.h"

#define SHM_NAME        "my_shm"
#define SHM_SIZE        1024

int main(void)
{
        void *mem;        /* map address */
        int shm_fd;        /* memory descriptor */
        int rc;

        /* create shm */
        shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0644);
        DIE(shm_fd == -1, "shm_open");

        /* resize shm to fit our needs */
        rc = ftruncate(shm_fd, SHM_SIZE);
        DIE(rc == -1, "ftruncate");

        mem = mmap(0, SHM_SIZE, PROT_WRITE | PROT_READ, MAP_SHARED, shm_fd, 0);
        DIE(mem == MAP_FAILED, "mmap");

        /* write number in shm */
        ((int*)mem)[0] = 2011;
          <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("mem[0] = %d\n",
((int*)mem)[0]);

        /* unmap shm */
```

```
        rc = munmap(mem, SHM_SIZE);
        DIE(rc == -1, "munmap");

        /* close descriptor */
        rc = close(shm_fd);
        DIE(rc == -1, "close");

        rc = shm_unlink(SHM_NAME);
        DIE(rc == -1, "unlink");

        return 0;
}
```

Depanare POSIX IPC

Memoria partajată

În Linux, zonele pot fi regăsite în `/dev/shm`, ca intrări formate din numele dat la creare + suffixul ".shm".

Cozi de mesaje

Conţinutul cozilor (conţinutul mesajelor) nu poate fi vizualizat, însă informaţii statistice pot fi obţinute prin montarea unui pseudo-sistem de fişiere:

```
so@spook$ sudo mkdir cozi
so@spook$ sudo mount -t mqueue none /mnt/cozi/
so@spook$ cat q_name
QSIZE:12         NOTIFY:0     SIGNO:0     NOTIFY_PID:0
```

Windows

Sistemul de operare Windows pune la dispoziţie o serie de mecanisme de comunicare şi schimb de date între aplicaţii. Cazul de care ne vom ocupa este doar cel în care aceste aplicaţii sunt procese care rulează pe aceeaşi maşină.

Înainte de a fi prezentate mecanismele de comunicare în sine trebuie introduse **mecanismele de sincronizare**, care sunt folosite pentru controlul accesului la resurse.

Mecanismele de sincronizare oferite de sistemul de operare Windows sunt mai multe şi mai complexe decât cele din Linux. Pentru sincronizare sunt necesare:

- unul sau mai multe obiecte de sincronizare (Synchronization Objects)
- o funcţie de aşteptare (Wait Functions).

Funcţii de aşteptare

Aşteptare după un singur obiect

Aceste funcţii aşteaptă după un singur obiect de sincronizare. Execuţia lor se termina când una din următoarele condiţii este adevărată :

- Obiectul de sincronizare este în starea `signaled`
- Timpul de aşteptare (time-out) a expirat. Acest timp poate fi setat ca `INFINITE` - timpul de aşteptare nu expiră niciodată.

WaitForSingleObject aşteaptă după un singur obiect şi are sintaxa :

```
 DWORD  WaitForSingleObject(
        HANDLE  hHANDLE,
        DWORD  dwMilliseconds
 );
```

Atenţie Obiectele de sincronizare nu pot fi folosite fără funcţii de sincronizare.

Obiecte de sincronizare

Mutex-uri

Crearea şi deschiderea

Sunt operaţii prin care se obţine un HANDLE al unui obiect de tip mutex. Este necesar doar un singur apel, fie el de creare sau de deschidere (se presupune ca alt proces a creat deja mutex-ul).

Pentru a crea un mutex se foloseşte funcţia CreateMutex cu sintaxa :

```
HANDLE  CreateMutex(
     LPSECURITY_ATTRIBUTES lpAttributes,
     BOOL bInitialOwner,
     LPCTSTR lpName
);
```

```
hMutex = CreateMutex(
     NULL,  /* default security attributes */
     FALSE, /* initially not owned */
     NULL,  /* unnamed mutex */
);
```

Pentru a deschide un mutex deja existent este definită funcţia OpenMutex cu sintaxa :

```
HANDLE  OpenMutex(
       DWORD  dwDesiredAccess,
       BOOL  bInheritHandle,
       LPCTSTR  lpName
);
```

```
hMutex = OpenMutex(
        MUTEX_ALL_ACCESS,  /* request full access */
        FALSE,             /* handle not inheritable */
        "MyMutex"          /* object name */
);
```

Obţinerea

Obţinerea unui mutex se realizează folosind una din funcţiile de aşteptare care sunt tratate anterior.

Încercarea de acaparare a unui mutex presupune următorii paşi:

- verificarea daca mutex-ul este disponibil
- daca da, îl pot acapara şi devine indisponibil, şi funcţia întoarce succes
- daca nu, aştept să devină disponibil, după care îl acaparez, şi funcţia întoarce succes
- la time-out funcţia întoarce eroare (atenţie! e posibil să nu existe time-out)

Încercarea de obţinere se poate face cu sau fară timp de expirare (time-out) în funcţie de parametrii daţi funcţiilor de aşteptare. Cea mai des folosită funcţie de aşteptare este WaitForSingleObject.

Cedarea

Folosind funcţia ReleaseMutex se cedează posesia mutex-ului, el devenind iar disponibil. Funcţia are următoarea sintaxa :

```
BOOL  ReleaseMutex(
       HANDLE  hMutex
);
```

Funcţia va eşua dacă procesul nu deţine mutex-ul.

Atenţie pentru a putea folosi această funcţie HANDLE-ul trebuie să aibă cel puţin dreptul de acces MUTEX_MODIFY_STATE.

Distrugerea

Operaţia de **distrugere** a unui mutex este aceeaşi ca pentru orice HANDLE. Se foloseşte funcţia CloseHandle. După ce

toate HANDLE-urile unui mutex au fost închise, mutexul este distrus şi resursele ocupate de acesta eliberate.

Atenţie La terminarea execuţiei unui program toate HANDLE-urile folosite de acesta sunt automat închise. Deci spre deosebire de semafoarele IPC din Linux, este imposibil ca un mutex (sau semafor) în Windows să mai existe în sistem după ce programele care l-au folosit/creat s-au terminat.

Semafoare

Un semafor este un obiect de sincronizare care are intern un contor ce ia doar valori pozitive. Atât timp cât semaforul (contorul) are valori strict pozitive el este considerat disponibil (*signaled*). Când valoarea semaforului a ajuns la zero el devine indisponibil (*nonsignaled*) şi următoarea încercare de decrementare va duce la o blocare a threadului/procesului de pe care s-a făcut apelul până când semaforul devine disponibil.

Operaţia de decrementare se realizează doar cu o singură unitate (la fel ca în API-ul POSIX), în timp ce incrementarea se poate realiza cu orice valoare în limita maximă.

Crearea şi deschiderea

Funcţia de creare a semafoarelor este CreateSemaphore şi are sintaxa :

```
HANDLE  CreateSemaphore(
       LPSECURITY_ATTRIBUTES  lpSemaphoreAttributes,
       LONG  lInitialCount,
       LONG  lMaximumCount,
       LPCTSTR  lpNAME
);
```

Această funcţie se poate folosi şi pentru deschiderea unui semafor deja existent. Alternativ, pentru a folosi un semafor deja existent, este necesar obţinerea HANDLE-ului semaforului, operaţie ce se realizează folosind funcţia OpenSemaphore cu următoarea sintaxă :

```
HANDLE  OpenSemaphore(
       DWORD  dwDesiredAccess,
       BOOL  bInheritHandle,
       LPCTSTR  lpNAME
);
```

Decrementarea (aşteptarea)

Operaţia de decrementare a semaforului cu sau fără aşteptare se realizează folosind una din funcţiile de aşteptare. Cea mai des folosită este funcţia WaitForSingleObject.

Incrementarea

Incrementarea semaforului se realizează folosind funcţia ReleaseSemaphore cu sintaxa :

```
BOOL  ReleaseSemaphore(
       HANDLE  hSemaphore,
       LONG  lReleaseCount,
       LPLONG  lpPreviousCount
);
```

Distrugerea

Operaţia de distrugere a unui semafor este similară cu cea de distrugere a unui mutex. Se foloseşte funcţia CloseHandle. După ce toate HANDLE-urile unui semafor au fost închise, semaforul este distrus şi resursele ocupate de acesta eliberate.

Cozi de mesaje (Mailslots)

Cozile de mesaje sunt un fel de pseudo-fisiere care rezidă în memorie. De aceea pot fi folosite prin intermediul funcţiilor standard de acces la fişiere. Fiind păstrate în memorie, toate aceste date au un caracter volatil, spre deosebire de fişiere, iar când toate handle-urile la un mailslot sunt distruse, acesta la rândul său, este distrus împreună cu datele, iar memoria este eliberată. (spre deosebire de cozile de mesaje de pe Linux)

Au următoarele caracteristici:

- Sunt unidirecţionale.
- Pot exista mai mulţi cititori şi mai mulţi scriitori, dar cel mai frecvent se foloseşte o arhitectură one-to-many .

- Un scriitor nu ştie sigur dacă mesajul său a ajuns la cititor.
- Dimensiunea mesajelor e limitată.
- Datorită modului de numire, se pot transmite mesaje prin reţea.

Detalii despre limitări:

Un exemplu tipic de folosire este următorul:

- `serverul mailslot` creează coada folosind `CreateMailslot`, apoi aşteaptă să primească un mesaj folosind un apel `ReadFile`
- `clientul mailslot` deschide coada folosind `CreateFile`, apoi transmite un mesaj folosind un apel `WriteFile`.

Crearea

Când un proces creează un mailslot, trebuie să-i atribuie o **denumire** de forma :

```
 \.\mailslot\[path]<nume>
```

Atenţie Prefixul "\\.\mailslot\" trebuie să existe exact în această formă, el fiind urmat de un nume, care eventual va fi precedat de o cale. Calea este asemănătoare cu cea a fişierelor. Un exemplu valid : "\\.\mailslot\test\commands" .

Pentru a crea o coadă de mesaje, se foloseşte funcţia CreateMailslot care are următoarea sintaxă şi întoarce un handle :

```
HANDLE  CreateMailslot(
        LPCTSTR  lpName,
        DWORD  nMaxMessageSize,
        DWORD  lReadTimeout,
        LPSECURITY_ATTRIBUTES  lpSecurityAttributes
);
```

În cazul în care se încearcă crearea unei cozi de mesaje cu o denumire deja existentă, se va întoarce `INVALID_HANDLE_VALUE`.

Atenţie! Handle-ul întors de această funcţie poate fi folosit pentru a efectua doar operaţii de citire (nu şi de scriere) cu coada de mesaje.

Deschiderea unei cozi existente

Pentru a deschide o coada de mesaje pentru scriere, se foloseşte funcţia CreateFile care va primi în loc de numele fişierului denumirea cozii de mesaje care se doreşte a fi deschisă si flagul `FILE_SHARE_READ`. Pentru a permite accesul concomitent al mai multor clienţi, trebuie adăugat şi flagul `FILE_SHARE_WRITE`.

Scrierea şi citirea

Citirea, respectiv scrierea din/în cozile de mesaje sunt asemănătoare cu operaţiile cu fişiere, folosindu-se aceleaşi funcţii :

- ReadFile şi ReadFileEx - pentru citire
- WriteFile şi WriteFileEx - pentru scriere

Obţinerea de informaţii despre o coada de mesaje

Pentru a obţine informaţii despre o coadă de mesaje, se foloseşte funcţia GetMailslotInfo ce are următoarea sintaxă:

```
BOOL  GetMailslotInfo(
        HANDLE  hMailslot,
        LPDWORD  lpMaxMessageSize,
        LPDWORD  lpNextSize,
        LPDWORD  lpMessageCount,
        LPDWORD  lpReadTimeout
);
```

Detalii despre schimbarea timpului de expirare:

Exemplu de utilizare

MailslotServer.c

```
#include <windows.h>
#include "utils.h"

LPSTR lpszSlotName = "\\.\mailslot\sample_mailslot";

int main(void)
{
        DWORD cbMessage, cMessage, cbRead, dwRet;
        HANDLE hMailslot;
        BOOL bRet;
        LPSTR lpszBuffer;

        /* Create Mailslot */
        hMailslot = CreateMailslot(
                lpszSlotName,
                0,                     /* no maximum message size */
                MAILSLOT_WAIT_FOREVER, /* no expiration period */
                NULL);                 /* no security attributes */
        DIE(hMailslot == INVALID_HANDLE_VALUE, "CreateMailSlot");

        /* Timeout - waiting for clients */
        Sleep(5000);

        /* Get number of messages form Mailslot */
        bRet = GetMailslotInfo(
                hMailslot,             /* mailslot handle */
                (LPDWORD) NULL,        /* no maximum message size */
                &cbMessage,            /* size of next message */
                &cMessage,             /* number of messages */
                (LPDWORD) NULL);       /* no read time-out */
        DIE(bRet == FALSE, "GetMailslotInfo");

        /* Read all messages from Mailslot */
        while (cMessage != 0) {

                lpszBuffer = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, cbMessage);
                DIE(lpszBuffer == NULL, "HeapAlloc");

                bRet = ReadFile(
                        hMailslot,
                        lpszBuffer,
                        cbMessage,
                        &cbRead,
                        (LPOVERLAPPED) NULL);
                DIE(bRet == FALSE, "ReadFile from Mailslot");

                                                                                < a
href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Received:     %s\n",
lpszBuffer);

                HeapFree(GetProcessHeap(), 0, lpszBuffer);

                bRet = GetMailslotInfo(
                        hMailslot,     /* mailslot handle */
                        NULL,          /* no maximum message size */
                        &cbMessage,    /* size of next message */
                        &cMessage,     /* number of messages */
                        NULL);         /* no read time-out */
                DIE(bRet == FALSE, "GetMailslotInfo");
        } /* end while */

        dwRet = CloseHandle(hMailslot);
        DIE (dwRet == FALSE, "CloseHandle");

        return 0;
}
```

MailslotClient.c

```
#include <windows.h>
#include "utils.h"

LPSTR lpszSlotName = "\\.\mailslot\sample_mailslot";

int main(void)
{
        HANDLE hMailslot;
```

```
        BOOL bRet;
        DWORD cbWritten, dwRet;
        LPSTR lpszBuffer = "Testing Mailslot";

        /* Open Mailslot */
        hMailslot = CreateFile(
                lpszSlotName,
                GENERIC_WRITE,
                FILE_SHARE_READ,
                NULL,
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL,
                NULL);
        DIE(hMailslot == INVALID_HANDLE_VALUE, "CreateFile");

        /* Send message */
        bRet = WriteFile(
                hMailslot,
                lpszBuffer,
                (DWORD) lstrlen(lpszBuffer) + 1,
                &cbWritten,
                NULL);
        DIE(bRet == FALSE, "Write file to Mailslot");

           <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Message
successfully send\n");

        /* Close Mailslot */
        dwRet = CloseHandle(hMailslot);
        DIE (dwRet == FALSE, "CloseHandle");

        return 0;
}
```

Memorie partajată (FileMapping)

Memoria partajată permite accesul mai multor procese la un fişier ca şi când fişierul ar fi o zonă de memorie. Astfel se pot folosi toate operaţiile aplicabile asupra memoriei, inclusiv pointeri.

O facilitate specială a `FileMapping` este aceea de memorie partajată identificată după nume (`named shared memory`).

ATENŢIE! Accesul la o zonă de memorie partajată trebuie reglementat folosind unul din mecanismele de sincronizare descrise mai sus!

Crearea unei zone de memorie partajată

Pentru crearea unei zone de memorie partajată se folosesc două funcţii care trebuie apelate în această ordine :

1. CreateFileMapping - este o funcţie pregătitoare care creează un obiect de tipul `File Mapping`, reprezentat de un `HANDLE`.
2. MapViewOfFile - pentru a mapa efectiv zona de memorie. Funcţia întoarce un pointer la zona de memorie partajată.

CreateFileMapping creează o resursă (un obiect) de tipul `FileMapping` şi are următoarea sintaxa :

```
HANDLE  CreateFileMapping(
        HANDLE  hFile,
        LPSECURITY_ATTRIBUTES  lpAttributes,
        DWORD  flProtect,
        DWORD  dwMaximumSizeHigh,
        DWORD  dwMaximumSizeLow,
        LPCTSTR  lpName
);
```

Dacă există un obiect cu acelaşi nume, dar de alt tip, funcţia va eşua şi va întoarce `NULL`.

MapViewOfFile întoarce un pointer la zona de memorie partajată şi are sintaxa :

```
LPVOID  MapViewOfFile(
        HANDLE  hFileMappingObject,
        DWORD  dwDesiredAccess,
        DWORD  dwFileOffsetHigh,
        DWORD  dwFileOffsetLow,
        SIZE_T  dwNumberOfBytesToMap
```

```
);
```

Accesul la o zonă de memorie partajată deja creată

Pentru a accesa o zonă de memorie partajată, creată de alt proces, se utilizează următoarele funcţii (în ordinea specificată) :

1. `OpenFileMapping` - o funcţie pregătitoare care accesează (deschide) un obiect de tipul `File Mapping`.
2. `MapViewOfFile` - pentru a mapa efectiv zona de memorie.

OpenFileMapping accesează o resursă/obiect deja existent de tipul `FileMapping` şi are sintaxa :

```
HANDLE  OpenFileMapping(
        DWORD  dwDesiredAccess,
        BOOL  bInheritHandle,
        LPCTSTR  lpName
);
```

Demaparea unei zone de memorie partajată

Pentru a demapa o zonă de memorie partajată, care a fost anterior mapată folosind funcţia `MapViewOfFile()`, se foloseşte funcţia UnmapViewOfFile care are următoarea sintaxă :

```
BOOL UnmapViewOfFile(
        LPCVOID lpBaseAddress
);
```

Exemple de utilizare

Serverul creează o zonă de memorie partajată, iar apoi aşteaptă un interval de timp. Clientul deschide zona de memorie partajată şi scrie un mesaj la începutul ei. Serverul termină aşteptarea şi afişează conţinutul zonei de memorie.

ServerSHM.c

```
#include <windows.h>
#include "utils.h"

#define BUF_SIZE 256
LPSTR szMapName = "MyFileMappingObject";
LPSTR szMsg = "Testing shared memory on windows";


int main(void)
{
        HANDLE hMapFile;
        LPVOID lpMapAddress;
        BOOL bRet;

        hMapFile = CreateFileMapping(
                INVALID_HANDLE_VALUE,         /* use swap, not a particular file */
                NULL,                              /* default security */
                PAGE_READWRITE,                      /* read/write access */
                0,                               /* maximum object size (high-order DWORD) */
                1024,                              /* maximum object size (low-order DWORD) */
                szMapName);                       /* name of mapping object */
        DIE(hMapFile == INVALID_HANDLE_VALUE, "CreateFileMapping");

        lpMapAddress = MapViewOfFile(
                        hMapFile,              /* handle to map object */
                        FILE_MAP_ALL_ACCESS,  /* read/write permission */
                        0,                      /* offset (high-order) */
                        0,                      /* offset (low-order) */
                        0);
        DIE(lpMapAddress == NULL, "MapViewOfFile");

        ZeroMemory(lpMapAddress, strlen(szMsg) + 1);
        CopyMemory(lpMapAddress, szMsg, strlen(szMsg));

        Sleep(5000);

        bRet = UnmapViewOfFile(lpMapAddress);
        DIE(bRet == FALSE, "UnampViewOfFile");

        bRet = CloseHandle(hMapFile);
```

```
        DIE(bRet == FALSE, "CloseHandle");

        return 0;
}
```

ClientSHM.c

```
#include <windows.h>
#include "utils.h"

#define BUF_SIZE 256
LPSTR szMapName = "MyFileMappingObject";

int main(void)
{
        HANDLE hMapFile;
        LPVOID lpMapAddress;
        BOOL bRet;

        hMapFile = OpenFileMapping(
           FILE_MAP_ALL_ACCESS,                /* read/write access */
           FALSE,                              /* do not inherit the name */
           szMapName);                         /* name of mapping object */
        DIE(hMapFile == INVALID_HANDLE_VALUE, "CreateFileMapping");

        lpMapAddress = MapViewOfFile(
                        hMapFile,               /* handle to map object */
                        FILE_MAP_ALL_ACCESS,       /* read/write permission */
                        0,                       /* offset (high-order) */
                        0,                       /* offset (low-order) */
                        0);
        DIE(lpMapAddress == NULL, "MapViewOfFile");

         <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Message in shm
is: <%s>\n", lpMapAddress);

        bRet = UnmapViewOfFile(lpMapAddress);
        DIE(bRet == FALSE, "UnampViewOfFile");

        bRet = CloseHandle(hMapFile);
        DIE(bRet == FALSE, "CloseHandle");

        return 0;
}
```

Exerciţii de laborator

În rezolvarea laboratorului folosiţi arhiva de sarcini lab05-tasks.zip

Platforma pe care rezolvaţi exerciţiile este la alegere

Linux

Atenţie

- Asiguraţi-vă că în timpul rulării exerciţiilor nu există resurse IPC cu acelaşi nume create anterior. Folosiţi comanda `make clean` care va şterge resursele alocate (şi eventual prost eliberate)
- Nu confundaţi variabila globală `errno` cu valoarea de retur a unei funcţii. Un apel de sistem va întoarce o anumită valoare de return în caz de eroare şi va seta variabila globală `errno` astfel încât să indice motivul erorii.

Întrucât la toate exerciţiile veţi avea nevoie de două console deschise în paralel, pe Linux, este recomandat să vă instalaţi **terminator**.

```
sudo apt-get install terminator
Ctrl+Shift+O -> open hOrizontal tab
Ctrl+Shift+E -> open vErtical tab
Ctrl+Shift+N -> move to Next tab
Ctrl+Shift+P -> move to Previous tab
Ctrl+Shift+W -> close current tab
```

- (**2 puncte**) Intraţi în directorul `1-fun/`:

- Programul `sem.c` crează un semafor şi îl incrementează de fiecare dată când apăsaţi o tastă.
- Rulaţi programul şi observaţi cum se schimbă conţinutul fişierului `/dev/shm/sem.my_sem` la apăsarea unei taste.Ce se întâmplă cu fişierul `/dev/shm/sem.my_sem` când terminaţi normal programul? Dar când îl terminaţi cu `Ctrl+C`? console1$ ./sem
  Press any key to continue (E/e to exit) console2$ watch -n 1 -d 'cat /dev/shm/sem.my_sem | hexdump -d -n 1'
- Programul `shm.c` crează o zona de memorie partajata şi scrie în ea un şir de caractere. Analizaţi conţinutul fişierului `/dev/shm/my_shm`. Observaţi ce se întâmplă cu zona de memorie partajată când programul `shm.c` se încheie normal (apăsaţi orice tastă) sau când este întrerupt cu `Ctrl+C`.console1$ ./shm
  Press any key to continue... console2$ cat /dev/shm/my_shm | hexdump -c
- Programul `mq.c` crează o coadă de mesaje şi pune un mesaj în coadă la fiecare apăsare a unei taste. În Linux cozile de mesaje sunt create într-un sistem de fişiere virtual. Acest sistem de fisiere poate fi montat în ierarhia voastră de fişiere astfel:$ sudo mkdir /dev/mqueue
  $ sudo mount -t mqueue none /dev/mqueue
- Rulaţi programul `mq` şi observaţi cum creşte dimensiunea cozii la fiecare mesaj pus în coadă.console1$ ./mq
  Press any key to continue (E/e exit) console2$ watch -n 1 cat /dev/mqueue/my_mqueue

Windows

1. (**2 puncte**) IPC between computers:
   1. Initial setup
      - Deschideţi maşina virtuala de Windows din VMware
      - Setaţi numele maşinii cu numele vostru şi workgroup-ul la `SO`
        - Click dreapta `MyComputer Properties`
        - Alegeţi tagul `Computer Name Change`
      - Închideţi maşina virtuala - `Shut Down` (**NU** restart)
      - Asiguraţi-vă ca maşina virtuală este conectata la reţea
        - Edit setting Network Addapter Bridged connection
      - Reporniţi maşina virtuală
   2. Porniţi proiectul `win\lab05.sln`
      - Compilaţi proiectele `1-fun-client` şi `1-fun-server`
      - Alegeţi un coleg cu care să faceţi echipă - unul din voi va rula serverul, iar celălalt clientul
        - Cel care rulează **serverul** porneşte proiectul `1-fun-server`
          - serverul poate primi mesaje de la orice client
        - Cel care ruleaza **clientul** porneşte proiectul `1-fun-client` şi introduce numele staţiei colegului care a pornit serverul
          - Clientul poate transmite si mesaje broadcast (la tot workgroup-ul) setând '*' ca nume al staţiei server
      - Test it! :)

Exerciţiul următor e independent de platformă, iar platforma e la alegere.

Linux / Windows

Să se implementeze un protocol simplu client - server folosind mecanisme IPC. Serverul întreţine o tabelă de dispersie (hashtable), conţinând cuvinte, în care se fac inserări şi ştergeri comandate de mesajele primite de la clienţi. Inserarea într-un tablou (bucket) se face la finalul acestuia.

Clienţii primesc operaţiile prin argumentele primite în linia de comanda la lansarea in executie. Exemplu:

```
./client a vincent c a test p
```

În acest caz, clientul va trimite serverului, în ordine, mesajele: *adaugă* în hashtable cuvântul "vincent", *clear* pentru golirea tabelei, *adaugă* cuvântul "test" şi va afişa conţinutul tabelei.

Exerciţiul se compune din 3 părţi:

- **comunicarea prin mesaje** - clienţii trimit comenzi serverului prin intermediul unei cozi de mesaje
- **tabela de dispersie** - serverul va menţine tabela în memoria partajată, iar clientul va citi din această zonă de memorie

de fiecare dată când are nevoie să printeze
- **sincronizarea accesului la tabelă** - se va realiza prin semafoare

1. (**0.5 puncte**) Acomodarea cu codul deja existent
   - Urmăriţi sursele din proiect:
     - `sever.c` - conţine codul rulat de server
     - `client.c` - conţine codul rulat de client
     - `common.h` - conţine structurile necesare protocolului
     - `generic_queue.h` - header cu funcţiile generale pentru lucrul cu coada de mesaje
     - `generic_shm.h` - header cu funcţiile generale pentru lucrul cu memoria partajată
     - `generic_sem.h` - header cu funcţiile generale pentru lucrul cu semafoare
     - `unix_*.c` - conţine implementarea unix a funcţiilor din `generic_*.h`
     - `win_*.c` - conţine implementarea windows a funcţiilor din `generic_*.h`
     - `hashtable.h`, `hashtable.c` - reprezintă interfaţa şi implementarea funcţiilor de lucru cu tabela de dispersie
     - `hash.h`, `hash.c` - reprezintă interfaţa şi implementarea funcţiei de hash
   - Compilaţi şi rulaţi serverul şi clientul din 2 console paralele de terminator
   - **Atenţie!** Fişierul `common.h` conţine structurile necesare la următoarele exerciţii
2. (**2 puncte**) Comunicare prin mesaje
   - Creaţi un `server` şi un `client` care să comunice prin următoarele comenzi:
     - `'a S'`: trimite serverului mesajul de adăugare în hashtable a cuvântului S;
     - `'c'`: trimite serverului mesajul de golire a conţinutului tabelei (clear);
     - `'p'`: clientul afişează la standard output conţinutul tabelei (formatul este precizat mai jos);
     - `'e'`: clientul îi spune serverului să îşi încheie execuţia.
   - Trebuie să completaţi funcţiile msgq_* din fişierul *unix_queue.c/win_queue.c* relativ la intefaţa din fişierele *common.h* şi *generic_queue.h*
   - Funcţiile de `msgq_send` şi `msgq_receive` trebuie să trimită/primească **toată** structura `message_t` primită ca parametru. (nu doar unul din câmpuri)
   - Aceste funcţii sunt deja apelate din codul de server - *server.c*, respectiv client - *client.c*
   - Hint:
     - Urmăriţi în sursa *unix_queue.c/win_queue.c* comentariile *TODO 1*
     - În această fază trebuie să funcţioneze doar trimiterea mesajelor
     - Reveniţi la secţiunea de Cozi de mesaje POSIX / Mailslots
     - Linux:
       - în funcţie *msgq_create* va trebui să creaţi coada de mesaje. Aveţi grijă ce dimensiuni alegeţi! (man mq_setattr)
     - Testaţi trimiţând mesaje de la client la server:
       - Porniţi serverul şi clientul în două console diferite: ./server ./client a test
       - Pentru a închide serverul: ./client e
3. (**2 puncte**) Tabela de dispersie
   - Completaţi funcţiile shm_* din *unix_shm.c/win_shm.c* relativ la intefaţa din fişierele *common.h* şi *generic_shm.h*.
   - Aceste funcţii sunt deja apelate din codul de server - *server.c*, respectiv client - *client.c*
   - **Hints**:
     - Urmăriţi în sursa *unix_shm.c/win_shm.c* comentariile *TODO 2*
     - Reveniţi la secţiunea de Memoria partajată POSIX / File Mappings
   - Testaţi funcţionalitatea:./server ./client a test1 a test2
     ./client p
     ./client c
     ./client p
     ./client e
4. (**3.5 puncte**) Sincronizare prin semafoare
   - Sincronizarea trebuie să fie **"fine grained"**, adică la nivel de bucket.
     - Operaţiile de 'print' şi 'clear' nu trebuie să ia toate semafoarele în acelaşi timp.
   - Nu este acceptabilă existenţa unui **singur** obiect de sincronizare pentru toată tabela.
     1. (**2 puncte**) Implementarea codului pentru semafoare
        - Completaţi funcţiile din *unix_sem.c/win_sem.c* relativ la intefaţa din fişierul *common.h* şi *generic_sem.h*
        - **Hints**:
          - Urmăriţi comentariile cu *TODO 3*
          - Reveniţi la secţiunea de Semafoare POSIX / Semafoare Windows
     2. (**1.5 puncte**) Realizarea accesului exclusiv
        - De data aceasta trebuie să decideţi voi **unde** se aplică funcţiile mai sus implementate pentru a asigura sincronizarea
        - Testaţi funcţionalitatea

BONUS

1. (**1 so karma**) Funny semaphores
   - Intraţi în directorul '3-funny_sem'
   - Clientul (reprezentat print 'client.c') vrea să trimită serverului o valoare magică, considerată a fi "the answer to the Ultimate Question of Life, the Universe, and Everything"
   - Pentru asta clientul trebuie să folosească o metodă mai neobişnuită - numărul trebuie transmis doar prin intermediul unui semafor.

EXTRA

- **EXTRA** IPC în Python
  - Rulaţi şi citiţi codul din arhiva PySHM - server în C şi client în Python
  - Instalaţi modulul de Python ipc_posix şi studiaţi exemple din arhiva de instalare

Soluţii

lab05-sol.zip

Resurse utile

- Fast User-level Locking In Linux
- Windows IPC
- Linux ipc(5) man page describing System V IPC
- Beej's Guide to Unix IPC