

# 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 procese de pe aceeași mașină sau de pe mașini diferite. Exemple de mecanisme IPC:

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

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 deținut (posedat, acaparat) doar de un singur proces/thread (în funcție de implementare) la un moment dat. Drept urmare, operațiile de baza cu mutex-uri sunt cele de obținere și de eliberare.

Odată obținut 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) așteptând 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 și 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 și 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 își păstrează ordinea în interiorul cozii de mesaje. Sunt mecanisme de comunicare unidirecționale atât pe Linux, cât și pe Windows.

- **Memorie partajată** ([Linux](#), [Windows](#))

Acest mecanism permite comunicarea între procese prin accesul direct și 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 aceeași mașină (spre deosebire de socket-uri, pipe-urile cu nume și cozile de mesaje din Windows)

## Linux

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

- [System V](#) Inter-Process Communication, derivat din distribuția 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 execuțiilor proceselor
- memorie partajată (shared memory) - realizează partajarea memoriei între procese

API-ul studiat în acest laborator este cel [POSIX](#).

Obiectele de tip IPC pe care se concentrează laboratorul de față sunt gestionate global de sistem și rămân în viață chiar dacă procesul creator moare. Faptul că aceste resurse sunt globale în sistem are implicații 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 și de dealocarea resurselor, altfel ele rămân în sistem până la ștergerea lor manuală sau până la repornirea sistemului.

Faptul că obiectele IPC sunt globale în sistem poate duce la apariția 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.

**Atenție!** Pentru folosirea API-ului trebuie să includeți la linking biblioteca `rt (-lrt)`.

### Semafoare POSIX

Semafoarele sunt resurse IPC folosite pentru sincronizarea între procese (e.g., pentru controlul accesului la resurse). Operațiile asupra unui semafor pot fi de *setare*, *verificare* a valorii (care poate fi  $\geq 0$ ), *test and set*. Un semafor poate fi privit ca un contor ce poate fi incrementat și 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 execuție ale aceluiași proces, sau între procese - cu condiția ca semaforul să fie într-o zonă de memorie partajată.

În continuare vor fi luate în discuție semafoarele cu nume. Diferențele față de cele fără nume apar în funcțiile de creare și distrugere, celelalte funcții fiind identice.

- ambele tipuri de semafoare sunt reprezentate în cod prin tipul `sem_t`.
- semafoarele cu nume sunt identificate la nivel de sistem printr-un șir de forma `"/nume"`.
- fișierele antet necesare sunt `sem.h` și `sys/sem.h`.

### Crearea și deschiderea

Un proces poate crea sau deschide un semafor existent cu funcția [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 fișierelor. Dacă flag-ul `O_CREAT` este prezent, trebuie folosită cea de-a doua formă a funcției, specificând permisiunile și valoarea inițială.

Singurele posibilități pentru al doilea argument sunt:

- 0 - se deschide semaforul dacă există

- 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 și aflarea valorii

Valoarea unui semafor este decrementată cu funcția [sem\\_wait](#):

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

Dacă semaforul are valoarea 0, funcția 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, funcția va întoarce -1, iar errno va fi setat la EAGAIN.

Un semafor este incrementat cu funcția [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 referință de pvalue.

### Închiderea și distrugerea

Un proces închide (notifică faptul că nu mai folosește) un semafor printr-un apel [sem\\_close](#):

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

Un proces poate șterge un semafor printr-un apel [sem\\_unlink](#):

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

Distrugerea efectivă a semaforului are loc după ce toate procesele care îl au deschis apelează [sem\\_close](#) sau se termină. Totuși, chiar și 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"` și va avea numele `"sem.my_semaphore"`

## Cozi de mesaje

Acestea permit proceselor să interschimbe date sub formă de mesaje

- la nivel de sistem sunt identificabile printr-un string de forma `"/nume"`.
- la nivel codului, o coada de mesaje este reprezentată de un descriptor de tipul `mqd_t`.
- fișierele antet necesare sunt:  `, și .`

## Crearea și deschiderea

Funcțiile de creare și deschidere sunt similare ca formă și 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 funcție de flag-uri (unul din cele de mai jos trebuie specificat), coada poate fi deschisă pentru:

- recepționare (`O_RDONLY`)
- trimitere (`O_WRONLY`)
- recepționare și trimitere (`O_RDWR`)

Dacă `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 și recepționarea de mesaje

Pentru a trimite un mesaj (de lungime cunoscută, stocat într-un buffer) în coadă se apelează [mq\\_send](#):

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

Mesajele sunt ținute în coadă în ordinea descrescătoare a priorității.

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

Pentru a primi un mesaj dintr-o coadă (și anume: cel mai vechi mesaj cu cea mai mare prioritate) se folosește [mq\\_receive](#):

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

Dacă `priority` este non-`NULL`, zona de memorie către care face referire va reține 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](#).

**Atenție!** La primirea unui mesaj, lungimea buffer-ului trebuie să fie **cel puțin 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 și ștergerea

Închiderea (eliberarea "referinței") unei cozi este posibilă prin apelul [mq\\_close](#):

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

Ștergerea se realizează cu un apel [mq\\_unlink](#):

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

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

### 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");

        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;
}
```

&lt; a

### Memoria partajată

Acest mecanism permite comunicarea între procese prin accesul direct și 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).
- fișierele antet necesare sunt: , și .

### Crearea și 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 realizează 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 sau se termină.

## 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

Conținutul cozilor (conținutul mesajelor) nu poate fi vizualizat, însă informații statistice pot fi obținute prin montarea unui pseudo-sistem de fișiere:

```

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

```

## Windows

Sistemul de operare Windows pune la dispoziție o serie de mecanisme de comunicare și schimb de date între aplicații. Cazul de care ne vom ocupa este doar cel în care aceste aplicații sunt procese care rulează pe aceeași mașină.

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 și mai complexe decât cele din Linux. Pentru sincronizare sunt necesare:

- unul sau mai multe [obiecte de sincronizare \(Synchronization Objects\)](#)
- o [funcție de așteptare \(Wait Functions\)](#).

### Funcții de așteptare

#### Așteptare după un singur obiect

Aceste funcții așteaptă după un singur obiect de sincronizare. Execuția lor se termină când una din următoarele condiții este adevărată :

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

[WaitForSingleObject](#) așteaptă după un singur obiect și are sintaxa :

```

DWORD WaitForSingleObject(
    HANDLE hHANDLE,
    DWORD dwMilliseconds
);

```

Funcții de așteptare

**Atenție!** Obiectele de sincronizare nu pot fi folosite fără funcții de sincronizare.

### Obiecte de sincronizare

## Mutex-uri

### Crearea și deschiderea

Sunt operații prin care se obține 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 folosește funcția [CreateMutex](#) cu sintaxa :

<pre>HANDLE CreateMutex(     LPSECURITY_ATTRIBUTES lpAttributes,     BOOL bInitialOwner,     LPCTSTR lpName );</pre>	<pre>hMutex = CreateMutex(     NULL, /* default security attributes */     FALSE, /* initially not owned */     NULL, /* unnamed mutex */ );</pre>
--	--

Pentru a deschide un mutex deja existent este definită funcția [OpenMutex](#) cu sintaxa :

<pre>HANDLE OpenMutex(     DWORD dwDesiredAccess,     BOOL bInheritHandle,     LPCTSTR lpName );</pre>	<pre>hMutex = OpenMutex(     MUTEX_ALL_ACCESS, /* request full access */     FALSE, /* handle not inheritable */     "MyMutex" /* object name */ );</pre>
--	---

### Obținerea

Obținerea unui mutex se realizează folosind una din funcțiile de așteptare care sunt tratate anterior.

Încercarea de acaparare a unui mutex presupune următorii pași:

- se verifică dacă mutex-ul este disponibil
- dacă da, îl pot acapara și devine indisponibil, și funcția întoarce succes
- dacă nu, aștept să devină disponibil, după care îl acaparez, și funcția întoarce succes
- la time-out funcția întoarce eroare (atenție: e posibil să nu existe time-out)

Încercarea de obținere se poate face cu sau fără timp de expirare (time-out) în funcție de parametrii dați funcțiilor de așteptare. Cea mai des folosită funcție de așteptare este [WaitForSingleObject](#).

### Cedarea

Folosind funcția [ReleaseMutex](#) se cedează posesia mutex-ului, el devenind iar disponibil. Funcția are următoarea sintaxa :

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

Funcția va eșua dacă procesul nu deține mutex-ul.

**Atenție!** pentru a putea folosi această funcție HANDLE-ul trebuie să aibă cel puțin dreptul de acces MUTEX\_MODIFY\_STATE.

### Distrușgerea

Operația de **distrușgere** a unui mutex este aceeași ca pentru orice HANDLE. Se folosește funcția [CloseHandle](#). După ce



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

**Atenție!** La terminarea execuției 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*) și următoarea încercare de decrementare va duce la o blocare a threadului/procesului de pe care s-a făcut apelul până când semaforul devine disponibil.

Operația 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 și deschiderea

Funcția de creare a semafoarelor este [CreateSemaphore](#) și are sintaxa :

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

Această funcție se poate folosi și pentru deschiderea unui semafor deja existent. Alternativ, pentru a folosi un semafor deja existent, este necesar obținerea HANDLE-ului semaforului, operație ce se realizează folosind funcția [OpenSemaphore](#) cu următoarea sintaxă :

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

### Decrementarea (așteptarea)

Operația de decrementare a semaforului cu sau fără așteptare se realizează folosind una din funcțiile de așteptare. Cea mai des folosită este funcția [WaitForSingleObject](#).

### Incrementarea

Incrementarea semaforului se realizează folosind funcția [ReleaseSemaphore](#) cu sintaxa :

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

### Distrugerea

Operația de distrugere a unui semafor este similară cu cea de distrugere a unui mutex. Se folosește funcția [CloseHandle](#). După ce toate HANDLE-urile unui semafor au fost închise, semaforul este distrus și resursele ocupate de acesta eliberate.

## Cozi de mesaje (Mailslots)

Cozile de mesaje sunt un fel de pseudo-fișiere care rezidă în memorie. De aceea, pot fi gestionate prin intermediul funcțiilor standard de acces la fișiere. Fiind păstrate în memorie, toate aceste date au un caracter volatil, spre deosebire de fișiere, 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 unidrecționale.
- Pot exista mai mulți cititori și mai mulți scriitori, dar cel mai frecvent se folosește o arhitectură one-to-many.

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

Detalii despre limitări

Un exemplu tipic de folosire este următorul:

- serverul mailslot creează coada folosind `CreateMailslot`, apoi așteaptă 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>
```

**Atenție!** 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 fișierelor. Un exemplu valid: "\\.\mailslot\test\commands".

Pentru a crea o coadă de mesaje, se folosește funcția [CreateMailslot](#) care are următoarea sintaxă și 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`.

**Atenție!** Handle-ul întors de această funcție poate fi folosit pentru a efectua doar operații de citire (nu și de scriere) cu coada de mesaje.

## Deschiderea unei cozi existente

Pentru a deschide o coadă de mesaje pentru scriere, se folosește funcția [CreateFile](#) care va primi în loc de numele fișierului denumirea cozi de mesaje care se dorește a fi deschisă și flagul `FILE_SHARE_READ`. Pentru a permite accesul concomitent al mai multor clienți, trebuie adăugat și flagul `FILE_SHARE_WRITE`.

## Scrierea și citirea

Citirea, respectiv scrierea din/în cozile de mesaje sunt asemănătoare cu operațiile cu fișiere, folosindu-se aceleași funcții:

- [ReadFile](#) și [ReadFileEx](#) - pentru citire
- [WriteFile](#) și [WriteFileEx](#) - pentru scriere

## Obținerea de informații despre o coadă de mesaje

Pentru a obține informații despre o coadă de mesaje, se folosește funcția [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");

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

        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) strlen(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 fișier ca și când fișierul ar fi o zonă de memorie. Astfel se pot folosi toate operațiile aplicabile asupra memoriei, inclusiv pointeri.

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

**Atenție!** 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 funcții care trebuie apelate în această ordine :

1. [CreateFileMapping](#) - este o funcție pregătitoare care creează un obiect de tipul File Mapping, reprezentat de un HANDLE.
2. [MapViewOfFile](#) - pentru a mapa efectiv zona de memorie. Funcția întoarce un pointer la zona de memorie partajată.

[CreateFileMapping](#) creează o resursă (un obiect) de tipul FileMapping și are următoarea sintaxă :

```

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

```

Dacă există un obiect cu același nume, dar de alt tip, funcția va eșua și va întoarce NULL.

[MapViewOfFile](#) întoarce un pointer la zona de memorie partajată și 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 funcții (în ordinea specificată) :

1. `OpenFileMapping` - o funcție 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` și 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 funcția `MapViewOfFile()`, se folosește funcția [UnmapViewOfFile](#) care are următoarea sintaxă :

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

## Exemple de utilizare

Serverul creează o zonă de memorie partajată, iar apoi așteaptă un interval de timp. Clientul deschide zona de memorie partajată și scrie un mesaj la începutul ei. Serverul termină așteptarea și afișează conținutul 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;
}
```

## Exerciții de laborator

În rezolvarea laboratorului folosiți arhiva de sarcini [lab05-tasks.zip](#)

Platforma pe care rezolvați exercițiile este la alegere

### Linux

#### Atenție!

- Asigurați-vă că în timpul rulării exercițiilor nu există resurse IPC cu același nume create anterior. Folosiți comanda `make clean` care va șterge resursele alocate (și eventual prost eliberate)
- Nu confundați variabila globală `errno` cu valoarea de retur a unei funcții. Un apel de sistem va întoarce o anumită valoare de return în caz de eroare și va seta variabila globală `errno` astfel încât să indice motivul erorii.

Întrucât la toate exercițiile veți avea nevoie de două console deschise în paralel, pe Linux, este recomandat să vă instalați **terminator**.

```
sudo apt-get install terminator
Ctrl+Shift+0 -> 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)** Intrați în directorul `1-fun/`:
  - Programul `sem.c` crează un semafor și îl incrementează de fiecare dată când apăsați o tastă.
  - Rulați programul și observați cum se schimbă conținutul fișierului `/dev/shm/sem.my_sem` la apăsarea unei taste. Ce

- se întâmplă cu fișierul /dev/shm/sem.my\_sem când terminați normal programul? Dar când îl terminați folosind 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'
- o Programul shm.c crează o zonă de memorie partajată și scrie în ea un șir de caractere. Analizați conținutul fișierului /dev/shm/my\_shm. Observați ce se întâmplă cu zona de memorie partajată când programul shm.c se încheie normal (apăsați orice tastă) sau când este întrerupt folosind Ctrl+C. console1\$ ./shm
- Press any key to continue... console2\$ cat /dev/shm/my\_shm | hexdump -c
- o Programul mq.c crează o coadă de mesaje și pune un mesaj în coadă la fiecare apăsare a unei taste. În Linux cozile de mesaje sunt create într-un sistem de fișiere virtual. Acest sistem de fișiere poate fi montat în ierarhia voastră de fișiere astfel: \$ sudo mkdir /dev/mqueue
  - \$ sudo mount -t mqueue none /dev/mqueue
  - o Rulați programul mq și observați cum crește 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 între calculatoare

1. Initial setup
  - Deschideți mașina virtuală Windows 2003 Server din VMware (cea locală, nu de pe server)
  - Setați numele mașinii cu numele vostru și workgroup-ul la S0
    - Click dreapta Computer Properties
    - Alegeți Computer Name, domain and workgroup settings Change settings
    - Alegeți Computer Name Change
  - Închideți mașina virtuală - Shut Down (**NU** restart)
  - Asigurați-vă ca mașina virtuală este conectată la rețea
    - Edit setting Network Addapter Bridged connection
  - Reporniți mașina virtuală
2. Porniți proiectul win\lab05.sln
  - Compilați proiectele 1-fun-client și 1-fun-server
  - Alegeți un coleg cu care să faceți echipă:
    - Unul din voi va rula **serverul** din proiectul 1-fun-server
      - o serverul poate primi mesaje de la orice client
    - Celălalt va rula **clientul** din proiectul 1-fun-client și va introduce numele stației colegului care a pornit serverul
      - o Clientul poate transmite și mesaje broadcast (la tot workgroup-ul) setând '\*' ca nume al stației server
  - Test it! :)

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

## Linux / Windows

Să se implementeze un protocol simplu client - server folosind mecanisme IPC. Serverul întreține o tabelă de dispersie (hashtable), conținând cuvinte, în care se fac inserări și ștergeri comandate de mesajele primite de la clienți. Inserarea într-un tablou (bucket) se face la finalul acestuia.

Clienții primesc operațiile prin argumentele primite în linia de comandă la lansarea în execuție. Exemplu:

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

În acest caz, clientul va trimite serverului, în ordine, mesajele: *a vincent* pentru adăugarea în hashtable a cuvântul "vincent", *c* pentru golirea tablei, *a test* pentru adăugarea cuvântului "test" și *p* pentru afișarea conținutului tablei.

Exercițiul se compune din 3 părți:

- **comunicarea prin mesaje** - clienții trimit comenzi serverului prin intermediul unei cozi de mesaje
- **tabela de dispersie** - serverul va menține 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

- Urmăriți sursele din proiect:
  - `server.c` - conține codul rulat de server
  - `client.c` - conține codul rulat de client
  - `common.h` - conține structurile necesare protocolului
  - `generic_queue.h` - header cu funcțiile generale pentru lucrul cu coada de mesaje
  - `generic_shm.h` - header cu funcțiile generale pentru lucrul cu memoria partajată
  - `generic_sem.h` - header cu funcțiile generale pentru lucrul cu semafoare
  - `unix_*.c` - conține implementarea unix a funcțiilor din `generic_*.h`
  - `win_*.c` - conține implementarea windows a funcțiilor din `generic_*.h`
  - `hashtable.h`, `hashtable.c` - reprezintă interfața și implementarea funcțiilor de lucru cu tabela de dispersie
  - `hash.h`, `hash.c` - reprezintă interfața și implementarea funcției de hash
- Compilați și rulați serverul și clientul în două console diferite
- **Atenție!** Fișierul `common.h` conține structurile necesare la următoarele exerciții

2. (2 puncte) Comunicare prin mesaje

- Creați un server și un client care să comunice prin următoarele comenzi:
  - 'a S': trimite serverului mesajul de adăugare în hashtable a cuvântului S (add);
  - 'c': trimite serverului mesajul de golire a conținutului tablei (clear);
  - 'p': clientul afișează la standard output conținutul tablei - formatul este precizat mai jos (print);
  - 'e': clientul îi spune serverului să își încheie execuția (exit).
- Trebuie să completați funcțiile `msgq_*` din fișierul `unix_queue.c/win_queue.c` relativ la interfața din fișierele `common.h` și `generic_queue.h`
- Funcțiile de `msgq_send` și `msgq_receive` trebuie să trimită/primească **toată** structura `message_t` primită ca parametru (nu doar unul din câmpuri).
- Aceste funcții sunt deja apelate din codul de server - `server.c`, respectiv client - `client.c`
- Hint:
  - Urmăriți în sursa `unix_queue.c/win_queue.c` comentariile *TODO 1*
  - În această fază trebuie să funcționeze doar trimiterea mesajelor
  - Reveniți la secțiunea de [Cozi de mesaje POSIX / Mailslots](#)
  - Linux:
    - în funcția `msgq_create` va trebui să creați coada de mesaje. Aveți grijă ce dimensiuni alegeți! ([man mq\\_setattr](#))
  - Testați trimitând mesaje de la client la server:
    - Porniți serverul și clientul în două console diferite: `./server ./client a test`
    - Pentru a închide serverul: `./client e`

3. (2 puncte) Tabela de dispersie

- Completați funcțiile `shm_*` din `unix_shm.c/win_shm.c` relativ la interfața din fișierele `common.h` și `generic_shm.h`.
- Aceste funcții sunt deja apelate din codul de server - `server.c`, respectiv client - `client.c`
- **Hints:**
  - Urmăriți în sursa `unix_shm.c/win_shm.c` comentariile *TODO 2*
  - Reveniți la secțiunea de [Memoria partajată POSIX / File Mappings](#)
  - Testați funcționalitatea: `./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.
  - Operațiile de 'print' și 'clear' nu trebuie să ia toate semafoarele în același timp.
- Nu este acceptabilă existența unui **singur** obiect de sincronizare pentru toată tabela.
  1. (2 puncte) Implementarea codului pentru semafoare
    - Completați funcțiile din `unix_sem.c/win_sem.c` relativ la interfața din fișierul `common.h` și `generic_sem.h`
    - Aceste funcții sunt deja apelate din codul de server - `server.c`, respectiv client - `client.c`
    - **Hints:**
      - Urmăriți comentariile cu *TODO 3*
      - Reveniți la secțiunea de [Semafoare POSIX / Semafoare Windows](#)
  2. (1.5 puncte) Realizarea accesului exclusiv
    - De data aceasta trebuie să decideți voi **unde** se aplică funcțiile mai sus implementate pentru a asigura sincronizarea
    - Testați funcționalitatea

**BONUS**



## 1. (1 so karma) Funny semaphores

- Intrați în directorul '3-funny\_sem'
- Clientul ('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 neobișnuită - numărul trebuie transmis doar prin intermediul unui semafor.

## EXTRA

### • EXTRA IPC în Python

- Rulați și citiți codul din arhiva [PySHM](#) - server în C și client în Python
- Instalați modulul de Python [ipc\\_posix](#) și studiați exemple din [arhiva de instalare](#)

## Soluții

[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](#)

From:

<http://elf.cs.pub.ro/so/wiki/> - **Sisteme de Operare**

Permanent link:

<http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-05>

Last update: **2012/04/10 14:14**