IPC

Contents

Windows

Sistemul de operare Windows pune la dispozitie o serie de mecanisme de comunicare si schimb de date intre aplicatii. Cazul de care ne vom ocupa este doar cel in care aceste aplicatii sunt procese care ruleaza pe aceeasi masina.

Inainte de a fi prezentate mecanismele de comunicare in 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 si mai complexe decat cele din Linux. Pentru sincronizare sunt necesare unul sau mai multe *obiecte de sincronizare* (synchronization objects) folosite impreuna cu o *functie de asteptare* (wait function).

ATENTIE! Obiectele de sincronizare nu pot fi folosite fara functii de sincronizare.

Odata parcurse mecanismele de sincronizare, vor fi prezentate doar doua din mecanisme de comunicare puse la dispozitie de Windows :

- ***File Mappings*** (memorie/fisiere partajate)
- ***Mail Slots*** (cozi de mesaje).

Obiecte de sincronizare

Mutex-uri

Un *mutex* este un obiect de sincronizare care poate fi detinut (posedat, acaparat) doar de un singur proces (sau thread) la un moment dat. Drept urmare, operatiile de baza cu mutex-uri sunt cele de obtinere si de eliberare.

Odata obtinut de un proces, un mutex devine indisponibil pentru orice alt proces. Orice proces care incearca sa acapareze un mutex indisponibil, se va bloca (un timp definit sau nu) asteptand ca el sa devina disponibil.

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

In continuare vor fi prezentate operatiile cu mutex-uri.

Crearea/deschiderea sunt operatii prin care se pregateste un mutex. Dupa cum am spus mai sus, pentru a opera cu orice obiect de sincronizare este necesar un `HANDLE` al acelui obiect. Scopul functiei de creare si a celei de deschidere este acela de a obtine un `HANDLE` al obiectului mutex. Prin urmare, este necesar doar *un singur apel*, fie el de creare sau de deschidere (se presupune ca alt proces a creat deja mutex-ul). Acest apel este efectuat *o singura data* la initializare; odata ce avem `HANDLE`-ul putem obtine/elibera mutex-ul de cate

ori avem nevoie.

Pentru a crea un mutex se foloseste functia CreateMutex cu sintaxa :

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

Pentru a **deschide** un mutex deja existent este definita functia OpenMutex cu sintaxa :

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

Obtinerea unui mutex se realizeaza folosind *functiile de asteptare* care sunt tratate intr-o sectiune urmatoare.

Incercarea de acaparare a unui mutex presupune urmatorii pasi :

1. este mutex-ul disponibil?
2. daca da, il pot acapara si devine indisponibil, si functia intoarce succes
3. daca nu, *astept* sa devina disponibil, dupa care il acaparez, si functia intoarce succes
4. time-out, si functia intoarce eroare (atentie! e posibil sa nu existe time-out)

Incercarea de obtinere se poate face cu sau fara timp de expirare (time-out) in functie de parametrii dati functiilor de asteptare. Cea mai des folosita **functie de asteptare** este **`WaitForSingleObject()`**.

Folosind functia ReleaseMutex se **cedeaza posesia** mutex-ului, el devenind iar disponibil. Functia are urmatoarea sintaxa :

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

Functia va esua daca procesul nu detine mutex-ul.

ATENTIE! pentru a putea folosi aceasta functie `HANDLE`-ul trebuie sa aiba dreptul de acces `MUTEX_MODIFY_STATE`.

Operatia de **distrugere** a unui mutex este aceeasi ca pentru orice `HANDLE`. Se foloseste functia CloseHandle. Dupa ce toate `HANDLE`-urile unui mutex au fost inchise, mutexul este distrus si resursele ocupate de acesta eliberate.

ATENTIE! La terminarea executiei unui program toate `HANDLE`-urile folosite de acesta sunt automat inchise. Deci spre deosebire de semafoarele IPC din Linux, este imposibil ca un mutex (sau semafor) in Windows sa mai existe in sistem dupa ce programele care l-au folosit/creat s-au terminat.

Functii de astepare

Asteptare dupa un singur obiect

Aceste functii asteapta dupa un singur obiect de sincronizare. Executia lor se termina cand una din urmatoarele conditii este adevarata :

- Obiectul de sincronizare este in starea 'signaled'
- Timpul de asteptare (time-out) a expirat. Acest timp poate fi setat ca `INFINITE` - timpul de asteptare nu expira niciodata.

Rezultatul intors de aceste functii poate fi :

- `WAIT_OBJECT_0` - Succes
- `WAIT_ABANDONED` - Obiectul specificat este un mutex care a fost abandonat, adica thread-ul care-l detinea s-a terminat fara sa-l elibereze. In acest caz threadul curent va deveni detinatorul mutexului iar starea mutexului va fi *nonsignaled* (mutex ocupat).
- `WAIT_IO_COMPLETION` - Asteptarea a fost intrerupta de un apel asincron de procedura.
- `WAIT_TIMEOUT` - Timpul de expirare s-a scurs.
- `WAIT_FAILED` - Functia a esuat. Informatii despre eroare pot fi obtinute folosind functia **`GetLastError()`**.

In continuare sunt prezentate pe larg functiile care fac parte din aceasta categorie :

SignalObjectAndWait semnalizeaza un obiect si asteapta dupa altul. Functia are sintaxa :

```
DWORD  SignalObjectAndWait(
HANDLE  hObjectToSignal,
HANDLE  hObjectToWaitOn,
DWORD  dwMilliseconds,
BOOL  bAlertable
 );
```

WaitForSingleObject asteapta dupa un singur obiect si are sintaxa :

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

WaitForSingleObjectEx permite o asteptare alertabila dupa un singur obiect si are sintaxa :

```
DWORD  WaitForSingleObjectEx(
HANDLE  hHandle,
DWORD  dwMilliseconds,
BOOL  bAlertable
 );
```

Asteptare dupa mai multe obiecte

Aceste functii asteapta dupa mai multe obiecte de sincronizare. Executia lor se termina cand una din urmatoarele conditii este adevarata:

- Starea unui obiect de sincronizare SAU starea tuturor obiectelor de sincronizare este 'signaled' (depinde de parametri)
- Timpul de asteptare (time-out) a expirat. Acest timp poate fi setat ca `INFINITE` pentru a specifica ca timpul de asteptare nu va expira niciodata

WaitForMultipleObjects asteapta dupa mai multe obiecte si are sintaxa :

```
DWORD  WaitForMultipleObjects(
DWORD  nCount,
const  HANDLE*  lpHandles,
BOOL  bWaitAll,
DWORD  dwMilliseconds
 );
```

WaitForMultipleObjectsEx permite o asteptare alertabila dupa mai multe obiecte si are sintaxa :

```
DWORD  WaitForMultipleObjectsEx(
DWORD  nCount,
const  HANDLE*  lpHandles,
BOOL  bWaitAll,
DWORD  dwMilliseconds,
BOOL  bAlertable
 );
```

Asteptare alertabila si asteptare inregistrata

Functiile de asteptare alertabila sunt :

- `WaitForSingleObjectEx()`
- `WaitForMultipleObjectsEx()`
- `SignalObjectAndWait()`

Aceste functii ofera posibilitatea de a efectua operatii de asteptare alertabile. O operatie de asteptare alertabila se poate termina cand :

- conditiile specificate sunt adevarate
- sistemul programeaza o rutina de tratare a operatiilor de I/O terminate
- sistemul programeaza o rutina de tratare a unui apel asincron terminat

Controlul alertabilitatii se realizeaza prin parametrul `BOOL bAlertable` pe care aceste functii il accepta.

Functiile de asteptare inregistrata sunt folosite de programele cu thread-uri si vor fi explicate in laboratoarele care trateaza thread-urile.

Semafoare

Un *semafor* (semaphore) este un obiect de sincronizare care are intern un contor ce ia doar valori pozitive. Atat timp cat semaforul (contorul) are valori strict pozitive el este considerat disponibil (*signaled*). Cand valoarea semaforului a ajuns la zero el devine indisponibil (*nonsignaled*) si urmatoarea incercare de decrementare va duce la o blocare a threadului de pe care s-a facut apelul (si a procesului, daca acesta foloseste un singur thread) pana cand semaforul devine disponibil.

Operatia de decrementare se realizeaza doar cu *o singura unitate* (la fel ca in API-ul POSIX, dar spre deosebire de API-ul SysV unde se poate face decrementarea atomica a unui semafor cu mai multe unitati o data), in timp ce incrementarea se poate realiza cu orice valoare in limita maxima.

Crearea si deschiderea

Functia de creare a semafoarelor este CreateSemaphore si are sintaxa :

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

Se observa ca functia se poate folosi si pentru deschiderea unui semafor deja existent.

Alternativ, pentru a folosi un semafor **deja existent** este necesara obtinerea `HANDLE`-ului semaforului, operatie ce se realizeaza folosind functia **`OpenSemaphore()`** cu urmatoarea sintaxa :

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

Decrementarea/asteptarea si incrementarea

Operatia de decrementare a semaforului cu sau fara asteptare se realizeaza folosind una din functiile de asteptare. Cea mai des folosita este functia **`WaitForSingleObject()`**.

Incrementarea semaforului se realizeaza folosind functia ReleaseSemaphore cu sintaxa :

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

Distrugerea

Operatia de distrugere a unui semafor este similara cu cea de distrugere a unui mutex. Se foloseste functia **`CloseHandle()`**. Dupa ce toate `HANDLE`-urile unui semafor au fost inchise, semaforul este distrus si resursele ocupate de acesta eliberate.

Cozi de mesaje (Mailslots)

Cozile de mesaje sunt folosite de procese pentru a comunica între ele prin mesaje. Aceste mesaje îi păstrează ordinea în interiorul cozii de mesaje.

Mailslots sunt un fel de pseudo-fisiere care rezidă în memorie, i deci pot fi folosite prin intermediul funciilor standard de acces la fisiere. Datele dintr-un astfel de mailslot pot avea orice formă, atât timp cât nu depăesc limita de 64 Kbytes. Fiind păstrate în memorie, toate aceste date au un caracter volatil, spre deosebire de fiiere, iar când toate handle-urile la un mailslot sunt distruse, acesta la rândul său, este distrus împreună cu datele, iar memoria este eliberată.

Sistemul de cozi de mesaje bazate pe mailslots este de tipul client-server. Astfel :

- serverul mailslot

Este procesul care creează i deine coada de mesaje, obinând astfel un handle. Numai cu ajutorul acestui handle se poate citi din (dar **nu** se poate scrie în) coada de mesaje. Există i posibilitatea ca handle-ul să fie motenit.

- clientul mailslot

Este un proces care pune mesaje în coadă. Pot exista mai muli clieni simultan.

Limitări :

- Mesajele de tip broadcast sunt limitate la maximum 424 bytes, iar încercarea de a trimite un mesaj broadcast mai mare va eua, iar funcia va întoarce eroare.
- NU pot fi trimise mesaje de lungime 425 bytes sau 426 bytes.
- Lungimea maximă a unui mesaj este 64 Kbytes.

Denumirea Mailslot-urilor

Când un proces creează un mailslot, trebuie să-i atribuie o denumire, care are următoarea formă :

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

Atenie! Prefixul "\\.\mailslot\" trebuie sa 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 fiierelor. **Un exemplu valid : "\\.\mailslot\test\commands"** .

Un proces client, pentru a scrie într-un mailslot, va folosi aceeai denumire.

Cozile de mesaje pot fi folosite i pentru a comunica cu procese care rulează pe alte calculatoare. În acest sens, clientul va folosi denumiri care au structura :

```
\\<ComputerName>\mailslot\[path]<Nume>
```

Pentru a trimite mesaje unui întreg domeniu, denumirea va avea structura :

```
\\<DomainName>\mailslot\[path]<Nume>
```

Pentru a trimite mesaje tuturor, denumirea va avea structura :

```
\\*\mailslot\[path]<Nume>
```

Crearea

Pentru a crea un mailslot, se folosete funcia CreateMailslot care are următoarea sintaxă i întoarce un handle :

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

În cazul în care se încearcă crearea unui mailslot cu o denumire care deja există, se va întoarce INVALID_HANDLE_VALUE.

ATENIE! Handle-ul întors de această funcie poate fi folosit pentru a efectua doar operaii de citire (nu i de scriere) cu mailslot-ul.

Deschiderea unei cozi existente

Pentru a deschide un mailslot pentru scriere, se folosete funcia **`CreateFile()`** care va primi în loc de numele fiierului denumirea cozii de mesaje care se dorete a fi deschisă si flagul `FILE_SHARE_READ`. Pentru a permite accesul concomitent al mai multor clienți, trebuie adăugat şi flagul `FILE_SHARE_WRITE`.

Scrierea i citirea

Citirea i respectiv scrierea din/în cozile de mesaje sunt asemănătoare cu operaiile cu fiiere, folosindu-se aceleai funcii :

- **`ReadFile()`** i **`ReadFileEx()`** - pentru citire
- **`WriteFile()`** i **`WriteFileEx()`** - pentru scriere

Obinerea de informaii despre o coada de mesaje

Pentru a obine informaii despre o coadă de mesaje, se folosete funcia următoare :

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

Schimbarea timpului de expirare

Singura caracteristică a unei cozi de mesaje, care poate fi schimbată după ce coada a fost creată, este timpul de expirare. (Dimensiunea maximă a mesajelor acceptate de o coadă **nu** mai poate fi schimbată după ce aceasta a fost creată)

Funcia care setează această caracteristică este următoarea :

```
BOOL  SetMailslotInfo(
HANDLE  hMailslot,
DWORD  lReadTimeout
 );
```

Exemplu de utilizare

Exemplu 04-a. Crearea unei cozi de mesaje

```
HANDLE hSlot;

BOOL WINAPI CreareCoadaMesaje()
{
        LPSTR lpszSlotName = "\\\\.\\mailslot\\sample_mailslot";

        hSlot = CreateMailslot(lpszSlotName,
0,                              // fără dimensiune maximă
                MAILSLOT_WAIT_FOREVER,  // fără timp de expirare
(LPSECURITY_ATTRIBUTES) NULL); // fără atribute de securitate

if (hSlot == INVALID_HANDLE_VALUE)
{
// tratare eroare
return FALSE;
}

return TRUE;
}
```

Exemplu 04-b. Deschiderea cozii de mesaje si trimiterea unui mesaj

```
LPSTR lpszMessage = "Mesaj";
BOOL fResult;
HANDLE hFile;
DWORD cbWritten;

hFile = CreateFile("\\\\.\\mailslot\\sample_mailslot",
        GENERIC_WRITE,          // accesul dorit
        FILE_SHARE_READ,        // flag necesar pentru a scrie în mailslot
```

```
(LPSECURITY_ATTRIBUTES) NULL,  // atribute securitate
        OPEN_EXISTING,         // coada de mesaje există deja
        FILE_ATTRIBUTE_NORMAL,
(HANDLE) NULL);                // fiier template

if (hFile == INVALID_HANDLE_VALUE)
{
// tratare eroare
return FALSE;
}

fResult = WriteFile(hFile,
        lpszMessage,
(DWORD) lstrlen(lpszMessage) + 1,  // includem i null-ul care marchează sfâritul irului
        &cbWritten,
(LPOVERLAPPED) NULL);

if (!fResult)
{
// tratare eroare
        CloseHandle(hFile);
return FALSE;
}

printf("Mesajul a fost scris cu succes.\n");

fResult = CloseHandle(hFile);

if (!fResult)
{
// tratare eroare
return FALSE;
}

return TRUE;
```

Exemplu 04-c. Citirea mesajelor din coadă

```
HANDLE hFile;
BOOL fResult;
DWORD cbMessage, cMessage, cbRead;
LPSTR lpszBuffer;

cbMessage = cMessage = cbRead = 0;

// deschiderea cozii de mesaje

hFile = CreateFile("\\\\.\\mailslot\\sample_mailslot",
        GENERIC_READ,          // accesul dorit
        FILE_SHARE_READ,
(LPSECURITY_ATTRIBUTES) NULL,  // atribute securitate
        OPEN_EXISTING,         // coada de mesaje există deja
        FILE_ATTRIBUTE_NORMAL,
(HANDLE) NULL);                // fiier template

if (hFile == INVALID_HANDLE_VALUE)
{
// tratare eroare
return FALSE;
}

// determinarea numarului de mesaje din coadă
```

```
fResult = GetMailslotInfo(hFile,      // handle-ul  asociat mailslot-ului
(LPDWORD) NULL,               // fără dimensiune maximă a mesajelor
        &cbMessage,           // dimensiunea mesajului următor
        &cMessage,            // numărul de mesaje
(LPDWORD) NULL);              // fără timp de expirare

if (!fResult)
{
// tratare eroare
        CloseHandle(hFile);
return FALSE;
}

// dacă nu sunt mesaje

if (cbMessage == MAILSLOT_NO_MESSAGE)
{
printf("Coada de mesaje este goala.\n");
        CloseHandle(hFile);
return TRUE;
}

while (cMessage != 0)  // preiau toate mesajele din coadă
{
// alocare memorie pentru mesaj

        lpszBuffer = (LPSTR) HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, cbMessage);
if (lpszBuffer == NULL)
{
           (hFile);CloseHandle
return FALSE;
}

        fResult = ReadFile(hFile,
                lpszBuffer,
                cbMessage,
                &cbRead,
(LPOVERLAPPED) NULL);

if (!fResult)
{
        (GetProcessHeap(), 0, lpszBuffer);HeapFree
           (hFile);CloseHandle
return FALSE;
}

         // afiare mesaj

printf("Mesaj primit : %s\n", lpszBuffer);

        HeapFree(GetProcessHeap(), 0, lpszBuffer);

        fResult = GetMailslotInfo(hFile,     // handle-ul asociat mailslot-ului
(LPDWORD) NULL,               // fără dimensiune maximă
                &cbMessage,   // dimensiunea următorului mesaj
                &cMessage,    // numărul de mesaje
(LPDWORD) NULL);              // fără timp de expirare

if (!fResult)
{
           (hFile);CloseHandle
```

```
return FALSE;
        }
}

CloseHandle(hFile);
return TRUE;
```

Memorie partajata (FileMapping)

File Mapping în cazul general permite accesul mai multor procese la un fisier ca i cand fisierul ar fi o zonă de memorie. Astfel se pot folosi toate operaiile aplicabile asupra memoriei, inclusiv pointeri.

O facilitate specială a File Mapping este aceea de "named shared memory", sau memorie partajata identificată de un nume. Această facilitate specială este subiectul laboratorului de faă.

ATENIE! Accesul la o zonă de memorie partajată trebuie reglementat folosind unul din mecanismele de sincronizare descrise în laboratorul precedent!

Crearea unei zone de memorie partajată

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

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

CreateFileMapping creează o resursă (un obiect) de tipul *FileMapping* i are următoarea sintaxa :

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

Dacă există un obiect cu acelai nume dar de alt tip, funcia va eua i va întoarce NULL.

MapViewOfFile întoarce un pointer la zona de memorie partajată i 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 funcii (în ordinea specificată) :

1. **`OpenFileMapping()`** - o funcie 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* i 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 funcia **`MapViewOfFile()`**, se folosete funcia **`UnmapViewOfFile()`** care are următoarea sintaxă :

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

Exemplu de utilizare

Exemplu 03-a. Secvena de program care creează o zonă de memorie partajată

```
HANDLE hMapFile;
LPVOID lpMapAddress;

hMapFile = CreateFileMapping(INVALID_HANDLE_VALUE,      // swap, nu mapăm un fiier anume
NULL,// securitatea standard
                        PAGE_READWRITE,// read/write
0,// dimensiunea (partea superioară)
1024,// dimensiunea (partea inferioară)
"MyFileMappingObject");// numele

if (hMapFile == NULL)
{
// tratare eroare
}

// maparea zonei de memorie

lpMapAddress = MapViewOfFile(hMapFile,     // handle-ul obiectului
                        FILE_MAP_ALL_ACCESS,// read/write
0,// deplasamentul (partea superioară)
0,// deplasamentul (partea inferioară)
0);// dimensiunea (0 = întreaga zonă)

if (lpMapAddress == NULL)
```

```
{
// tratare eroare
}

// acum la adresa lpMapAddress avem zona de memorie partajată

...

// demaparea zonei de memorie
UnmapViewOfFile(lpMapAddress);

CloseHandle(hMapFile);
```

Exemplu 03-b. Secvena de program care accesează zona de memorie partajată

```
HANDLE hMapFile;
LPVOID lpMapAddress;

hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS,       // acces read/write
FALSE,// copii nu motenesc handle-ul
"MyFileMappingObject");// numele obiectului

if (hMapFile == NULL)
{
// tratare eroare
}

lpMapAddress = MapViewOfFile(hMapFile,     // handle-ul obiectului
                        FILE_MAP_ALL_ACCESS,
0,// deplasamentul (partea superioară)
0,// deplasamentul (partea inferioară)
0);// dimensiunea (0 = întreaga zonă)

if (lpMapAddress == NULL)
{
//tratare eroare
}

// acum la adresa lpMapAddress avem zona de memorie partajata

...

//demaparea zonei de memorie
UnmapViewOfFile(lpMapAddress);

CloseHandle(hMapFile);
```

Linux

Linux pune la dispozitie 2 seturi de API-uri mecanisme de comunicare inter-proces, ce tin de standarde diferite:

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

Ambele standarde specifica 3 mecanisme:

- mesaje (messages) - realizeaza schimbul de mesaje cu orice proces sau server
- semafoare (semaphores) - realizeaza sincronizarea executiilor unor procese
- memorie partajata (shared memory) - realizeaza partajarea memoriei intre procese

API-ul studiat in acesta laborator este cel POSIX.

Obiectele de tip IPC pe care se concentreaza laboratorul de fata sunt gestionate global de sistem si raman in viata chiar daca procesul creator moare. Faptul ca aceste resurse sunt globale in sistem are implicatii contradictorii. Pe de o parte, daca un proces se termina, datele plasate in obiecte IPC pot fi accesate ulterior de alte procese; pe de alta parte, procesul proprietar trebuie sa se ocupe si de dealocarea resurselor, altfel ele raman in sistem pana la stergerea lor manuala sau pana la un reboot. Faptul ca obiectele IPC sunt globale in sistem poate duce la aparitia unor probleme: cum numarul de mesaje care se afla in cozile de mesaje din sistem e limitat global, un proces care trimite multe asemenea mesaje poate bloca toate celelalte procese.

ATENTIE!!! `Pentru folosirea API-ului trebuie sa includeti la linking biblioteca 'rt' (-lrt).`

Semafoare

Semafoarele sunt resurse IPC folosite pentru sincronizarea intre procese (e.g. pentru controlul accesului la resurse). Operatiile asupra unui semafor pot fi de *setare* sau *verificare* a valorii (care poate fi mai mare sau egala cu 0) sau de *test and set*. Un semafor poate fi privit ca un contor ce poate fi incrementat si decrementat, dar a carui valoare nu poate scadea sub 0.

Semafoarele POSIX sunt de 2 tipuri:

- cu nume, folosite in general pentru sincronizare intre procese distincte;
- bazate pe memorie (fara nume), ce pot fi folosite doar pentru sincronizarea intre firele de executie ale unui proces.

In contiunare vor fi luate in discutie **semafoarele cu nume**. Diferentele fata de cele bazata pe memorie consta in functiile de creare si distrugere, celelalte functii fiind identice.

- ambele tipuri de semafoare sunt reprezentate in cod prin tipul **sem_t**.
- semafoarele cu nume sunt indenficate la nivel de sistem printr-un sir de forma "/nume".
- header-ele necesare sunt **<fcntl.h>**, **<sys/types.h>** si **<semaphore.h>**.

Crearea si deschiderea

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

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

Comportamentul este similar cu cel de la deschiderea fisierelor. Daca flag-ul O_CREAT este prezent, trebuie folosita cea de-a doua forma a functiei, specificand permisiunile si valoarea initiala.

Decrementare, incrementare si aflarea valorii

Un semafor este decrementat cu functia **sem_wait**:

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

Daca semaforul are valoarea zero, functia blocheaza pana cand un alt proces "deblocheaza" (incrementeaza) semaforul.

Pentru a incerca decrementarea unui semafor fara riscul de a ramane blocat la acesta, un proces poate apela **sem_trywait**:

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

In cazul in care semaforul are deja valoarea zero, functia va intoarce -1 iar **errno** va fi setat la EAGAIN.

Un semafor este incrementat cu functia **sem_post**:

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

In cazul in care semaforul are valoarea zero, un proces blocat in 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);
```

In cazul in care exista procese blocate la semafor, implementare apelului pe Linux va returna **zero** in valoarea referita de pvalue.

Toate aceste functii intorc zero in caz de succes.

Inchiderea si distrugerea

Un proces *inchide* (notifica faptul ca nu mai foloseste) un semafor printr-un apel **sem_close**:

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

Un proces poate sterge un semafor printr-un apel **sem_unlink**:

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

Distrugerea efectiva a semaforului are loc dupa ce toate procesele care il au deschis apeleaza *sem_close* sau se termina. Totusi, chiar si in acest caz, apelul *sem_unlink* nu va bloca!

Cozi de mesaje

Acestea permit proceselor schimbarea de date intre procese sub forma de mesaje.

- la nivel de sistem sunt indentificabile printr-un string de forma "/nume".

- la nivel codului, o coada de mesage este reprezentata de un descriptor de tipul **mqd_t**.
- header-ele necesare pentru lucrul cu aceste obiecte sunt **<fcntl.h>**, **<sys/types.h>** si **<mqueue.h>**.

Crearea si deschiderea

Functiile de creare si deschidere sunt similare ca forma si semantica celor de la semafoare:

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

In functie de flag-uri (unul din cele de mai jos **trebuie** specificat), coada poate fi deschisa pentru:

- receptionare (O_RDONLY)
- trimitere (O_WRONLY)
- receptionare si trimitere (O_RDWR)

Daca **attr** e NULL, coada va fi creata cu atribute implicite. Structura mq_attr arata astfel:

```
struct mq_attr {
    long mq_flags;       /* 0 sau O_NONBLOCK */
    long mq_maxmsg;      /* numar maxim de mesaje in coada */
    long mq_msgsize;     /* dimensiunea maxima a unui mesaj */
    long mq_curmsgs;     /* numar de mesaje in coada */
};
```

Trimiterea si receptionarea de mesaje

Pentru a trimite un mesaj (de lungime cunoscuta, stocat intr-un buffer) in coada se apeleaza **mq_send**:

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

Mesajele sunt tinute in coada in ordine descrescatoare a prioritatii.

In cazul in care coada este plina, apelul blocheaza. Daca este o coada non-blocanta (O_NONBLOCK), functia va intoarce -1 iar **errno** va fi setat la **EAGAIN**.

Pentru a primi un mesaj dintr-o coada (si anume: cel mai vechi mesaj cu cea mai mare prioritate) se foloseste **mq_receive**:

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

Daca *priority* este non-NULL, zona de memorie catre care face referire va retine prioritatea mesajului extras.

In cazul in care coada este vida, apelul blocheaza. Daca este o coada non-blocanta (O_NONBLOCK), comportamentul este similar cu cel al **mq_send**.

ATENTIE!!! La primirea unui mesaj, lungimea buffer-ului trebuie sa fie **cel putin egala** cu dimensiunea maxima a mesajelor pentru coada respectiva, iar la trimitere **cel mult egala**. Dimensiunea maxima implicita se poate afla pe Linux din /proc/sys/kernel/msgmax.

Inchiderea si stergerea

Inchiderea (eliberarea "referintei") unei cozi este posibila prin apelul **mq_close**:

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

Stergerea se realizeaza cu un apel mq_unlink:

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

Semantica este similara cu cea de la semafoare: coada nu va fi stearsa efectiv decat dupa ce restul proceselor implicate o inchid.

Exemplu

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

/* buffer cel putin egal cu maximul implicit al sistemului
 * /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;

        mqd_t m = mq_open(NAME, (argc>1 ? O_CREAT : 0) | O_RDWR, 0666, NULL);
if (m == (mqd_t)-1) {
perror("queue open");
return 1;
}


if (argc > 1) {
if (mq_send(m, TEXT, strlen(TEXT), prio) == -1)
perror("queue send");
}
else {
if (mq_receive(m, buf, BUF_SIZE, &prio) == -1)
perror("queue receive");
printf("received: %s\n", buf);
mq_unlink(NAME);
}

return 0;
}
```

Memorie partajata

Acest mecanism permite comunicarea intre procese prin accesul direct si partajat la o zona de memorie bine determinata.

- la nivelul sistemului, o zona este identificata printr-un string de forma "/nume";
- la nivelul codului, o zona este reprezentata printr-un file descriptor (int).
- header-ele necesare pentru lucrul cu aceste obiecte sunt **<fcntl.h>**, **<sys/types.h>** si **<sys/mman.h>**.

Crearea si deschiderea

Apelul de creare/deschidere este similar ca semantica apelului open pentru fisiere "obsinuite":

```
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 zona de memorie partajata nou creata are dimensiunea initiala zero. Pentru a o dimensiona se foloseste **ftruncate**:

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

Maparea si eliberarea

Pentru a putea utiliza o zona de memorie partajata dupa deschidere, aceasta trebuie mapata in spatiul de memorie al procesului. Aceasta se realizeaza printr-un apel **mmap**:

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

Valoarea intoarsa reprezinta un pointer catre inceputul zonei de memorie sau MAP_FAILED in caz de esec. Acest apel are o larga aplicabilitate si va fi discutat in cadrul laboratorului de memorie virtuala. Momentan, pentru a mapa intregul continut al unei zone (shm_fd) de dimensiune cunoscuta (shm_len), recomandam folosirea apelului

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

Cand maparea nu mai este necesara, prin apelul **munmap** se realieaza demaparea:

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

Inchiderea si stergerea

Inchiderea unei zone de memorie partajata este identica cu inchidere unui fisier: apelul **close**.

Odata ce o zona de memoria a fost demapata si inchisa in toate procesele implicate, se poate sterge prin **shm_unlink**:

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

Semantica este identica cu cea de la functiile **_unlink* anterioare: stergerea efectiva este amanata pana ce toate procesele implicate inchid zona in cauza.

Depanare POSIX IPC

Memoria partajata

In Linux, zonele pot fi regasite in /dev/shm, ca intrari formate din numele dat la creare + suffixul ".shm".

Semafoare

Cozi de mesaje

Continutul cozilor (continutul mesajelor) nu poate fi vizualizat, insa informatii statistice pot fi obtinute prin montarea unui pseudo-sistem de fisiere:

```
mircea@beast3:/mnt$ sudo mkdir cozi
mircea@beast3:/mnt$ sudo mount -t mqueue none /mnt/cozi/
mircea@beast3:/mnt/cozi$ cat q_name
QSIZE:12         NOTIFY:0     SIGNO:0     NOTIFY_PID:0
```

Exercitii

Prezentare

Pentru a urmari mai uşor noţiunile expuse la începutul laboratorului urmăriţi aceasta prezentare: odp, pdf.

Quiz

Pentru autoevaluare raspundeti la intrebarile din acest quiz.

Exercitii pentru laborator

Exercitiile sunt independente de platforma. Folositi arhiva de sarcini.

1. **(2 puncte)** Intrai în directorul `01_xmit/`.
    - Inspectati continutul fisierelor `util.h` , `client.c` (sender) si `server.c` (receiver)
    - Procesul sender primeste din linia de comanda un numar intreg pozitiv si doreste sa-l trimita procesului receiver.
    - Folosindu-va doar de semafoare ca mijloc de comunicatie realizati transferul numarului astfel:
    - Procesul sender creaza semaforul , procesul receiver deschide semaforul si de asemenea la sfarsit il sterge.
    - Poate procesul receiver sa isi mai primeasca numarul daca procesul sender s-a terminat?
    - **Hints**: Windows: Analizati functia ReleaseSemaphore si observati cum se poate citi valoarea unui semafor.
2. **(2 puncte)** Implementai un protocol simplu de comunicaie între un client i un server folosind cozi de mesaje. Clientul se va conecta la server i va trimite acestuia numărul 1337 pe care server-ul îl va afia. Apoi clientul va trimite server-ului un mesaj de închidere.
    - **Hints**:Fişierul `common.h` conţine structurile necesare protocolului. Pe Windows nu putei selecta din coadă tipul mesajului pe care dorii să-l citii, deci asigurai-vă că mesajele vor putea fi trimise i recepionate în mod asemănător pe Windows i Linux
3. **(2 puncte)**: Folosind memoria partajată, realizai un transfer simplu de informaie între două procese astfel: server-ul va crea o zona de 4k de memorie i va pune numărul 1337 începând cu primul octet, clientul va citi i afia acest număr.
    - **Hints**: Deoarece clientul trebuie ruleze după ce serverul a creat, mapat i scris în zona partajată i înainte ca acesta să o elibereze, folosii o functie de sleep pentru server.

Soluții

Soluții exerciții laborator 5

Resurse utile

- Fast User-level Locking In Linux