8. Sincronizarea şi comunicarea între fire de execuție

Firele de execuție se execută în acelaşi spațiu de memorie, ceea ce înseamnă că o zonă de memorie poate fi accesată simultan de mai multe fire de execuție. În aceste situații intervine problema consistenței datelor, ce trebuie protejată prin sincronizarea accesului. Pe lângă aceasta, de regulă în cazul aplicațiilor bazate pe mai multe fire de execuție se pune şi problema comunicării între fire. O asemenea comunicare trebuie să asigure transmisia şi recepționarea mesajelor între fire, o comunicare fără pierderi care să asigure o bună funcționare a sistemului.

Pornind de la aceste probleme, în cadrul acestui capitol vom prezenta câteva aspecte legate de asigurarea consistenței datelor partajate şi comunicarea între mai multe fire de edxecuție.

8.1 Implementarea excluziunii mutuale

O secvență de program în cadrul căruia datele sunt accesate de mai multe fire poartă denumirea de *secțiune critică*. Dacă două sau mai multe fire se află simultan în cadrul unei secțiuni critice, datele rezultate pot fi inconsistente. Pentru a evita această situație programatorii pot apela la *excluziunea mutuală* prin care se asigură accesul unui singur fir la o secțiune critică.

În continuarea acestui sub-capitol vom prezenta metodele de implementare a excluziunii mutuale pe sisteme Windows şi Unix. Totodată, vom prezenta şi o încapsulare pentru cele două platforme într-o singură clasă.

8.1.1 Implementarea excluziunii mutuale utilizând API-ul Win32

API-ul Win32 pune la dispoziție mai multe metode de implementare a excluziunii mutuale. Una dintre cele mai răspândite metode întâlnite reprezintă utilizarea obiectelor *secțiune critică*, despre care vom discuta în continuare. Aceste obiecte asigură excluziune mutuală a firelor de execuție din cadrul aceluiaşi proces, fără a putea fi utilizate peste mai multe procese.

Un obiect secțiune critică se creează prin utilizarea structurii `CRITICAL_SECTION`. După declararea unei variabile de acest tip, secțiunea critică trebuie inițializată prin apelul funcției `InitializeCriticalSection()`, cu prototipul:

```
void WINAPI InitializeCriticalSection(
               LPCRITICAL_SECTION lpCriticalSection );
```

Singurul parametru al acestei funcții reprezintă adresa obiectului secțiune critică. Intrarea în secțiunea critică, adică acapararea obiectului de excluziune mutuală se realizează prin apelul funcției `EnterCriticalSection()`, cu prototipul:

```
void WINAPI EnterCriticalSection(
               LPCRITICAL_SECTION lpCriticalSection );
```

Ieşirea din secțiunea critică se realizează prin apelul funcției `LeaveCriticalSection()`, cu prototipul:

```
void WINAPI LeaveCriticalSection(
                LPCRITICAL_SECTION lpCriticalSection );
```

După terminarea utilizării obiectului secțiune critică, trebuie eliberate resursele atașate prin apelul funcției `DeleteCriticalSection()`. În caz contrar, starea firelor care așteaptă intrarea în secțiunea critică este nedefinită. Prototipul acestei funcții este următorul:

```
void WINAPI DeleteCriticalSection(
                LPCRITICAL_SECTION lpCriticalSection );
```

Utilizarea secțiunilor critice este extrem de simplă şi intuitivă. De exemplu, dacă dorim să sincronizăm accesul asupra unei liste, mai întâi creem obiectul secțiune critică, îl inițializăm, după care apelăm funcțiile de intrare şi ieşire din secțiune critică la fiecare utilizare a listei:

```
// Declararea şi inițializarea obiectului secțiune critică
CRITICAL_SECTION mutex;
__try
{
       InitializeCriticalSection( &mutex );
}
__except ( 1 )
{
       assert( 0 );
}

…

// Intrarea în secțiunea critică
__try
{
       EnterCriticalSection( &mutex);
}
__except ( 1 )
{
       assert( 0 );
}


// Utilizarea listei
…

// Ieşirea din secțiunea critică
__try
{
       LeaveCriticalSection( &mutex );
}
__except ( 1 )
{
       assert( 0 );
```

```
}
…

// Distrugerea secțiunii critice
__try
{
	DeleteCriticalSection( &mutex );
}
__except ( 1 )
{
	assert( 0 );
}
```

Implementarea obiectelor secțiune critică este una optimizată prin utilizarea unor instrucțiuni dedicate ale procesorului, rezultând un timp de accesare foarte redus pentru aceste obiecte. Pe lângă secțiunile critice, API-ul Win32 definește şi obiectele *mutex* pentru asigurarea excluziunii mutuale. Acestea se pretează de regulă pentru sincronizarea accesului firelor din procese diferite. Prezentarea acestui set de funcții este în afara scopului acestei lucrări, pentru mai multe detalii cititorul fiind îndrumat să consulte descrierea funcțiilor `CreateMutex()`, `OpenMutex()` şi `ReleaseMutex()`.

8.1.2 Implementarea excluziunii mutuale utilizând API-ul POSIX

Pe lângă setul de funcții pentru crearea firelor de execuție prezentat în capitolul anterior, API-ul POSIX pune la dispoziție şi un set de funcții pentru asigurarea sincronizării accesului la secțiunile critice. Aceasta se realizează prin intermediul obiectelor mutex. Comportamentul obiectelor mutex se poate configura prin utilizarea structurii `pthread_mutexattr_t` definită în `pthread.h` în care se stochează valoarea corespunzătoare tipului de mutex dorit:

```
typedef struct {
	int __mutexkind;
}pthread_mutexattr_t;
```

Valorile posibile pentru `__mutexkind` sunt următoarele:

- `PTHREAD_MUTEX_DEFAULT`: obiect mutex implicit, pentru care închiderea recursivă a mutexului va duce la un comportament nedefinit;
- `PTHREAD_MUTEX_NORMAL`: obiect mutex ce nu asigură detectarea deadlock-ului, închiderea recursivă ducând la deadlock;
- `PTHREAD_MUTEX_RECURSIVE`: obiect mutex ce asigură o închidere recursivă pentru acelaşi fir de execuție. Eliberarea mutexului va necesita acelaşi număr de paşi;
- `PTHREAD_MUTEX_ERRORCHECK`: obiect mutex ce asigură verificarea erorii de închidere/eliberare. O încercare de închidere a mutexului fără o eliberare în prealabil (pentru acelaşi fir) va genera o eroare. O încercare de eliberare a unui mutex închis de către un alt fir va genera o eroare. O încercare de eliberare a unui mutex eliberat va genera o eroare.

Înainte de setarea valorii, structura `pthread_mutexattr_t` trebuie inițializată prin apelul funcției `pthread_mutexattr_init()`, cu prototipul:

```
int pthread_mutexattr_init( pthread_mutexattr_t* pAttr );
```

Singurul parametru al acestei funcții reprezintă un pointer către structura care este inițializată. În caz de succes, această funcție returnează valoarea 0 iar altfel se returnează un cod de eroare.

Setarea valorii pentru `__mutexkind` se realizează prin intermediul funcției `pthread_mutexattr_settype()`, cu prototipul:

```
int pthread_mutexattr_settype( pthread_mutexattr_t* pAttr, int nType );
```

Primul parametru, `pAttr` este un pointer către o structură în care se stochează tipul mutexului, iar al doilea parametru, `nType` reprezintă tipul obiectului mutex. În caz de succes, funcția returnează valoarea 0 iar altfel se returnează un cod de eroare.

După utilizare, structura `pthread_mutexattr_t` trebuie distrusă. Distrugerea nu se referă la eliberarea memoriei alocate structurii ci la eliberarea memoriei alocate membrilor acestuia. Momentan structura conține un singur membru ce nu necesită eliberare, dar în viitor pot fi adăugate o serie de alte câmpuri ce vor necesita eliberare. Distrugerea acestei structuri se realizează prin apelul funcției `pthread_mutexattr_destroy()`, cu prototipul:

```
int pthread_mutexattr_destroy( pthread_mutexattr_t* pAttr );
```

Parametrul transmis, `pAttr` reprezintă un pointer către structura distrusă, funcția returnând valoarea 0 în caz de succes şi o valoarea diferită de 0, reprezentând codul de eroare, în caz de eroare.

În continuare vom prezenta funcțiile pentru utilizarea obiectelor mutex. Un obiect mutex este de tipul `pthread_mutex_t`, definit în `pthread.h`:

```
typedef struct {
        struct _pthread_fastlock lock;
        _pthread_descr owner;
        int kind;
        unsigned int count;
}pthread_mutex_t;
```

Câmpurile acestei structuri asigură implementarea închiderii recursive, asigură specificarea proprietarului şi implementarea excluziunii mutuale. Inițializarea unui obiect mutex se realizează prin apelul funcției `pthread_mutex_init()`, cu prototipul:

```
int pthread_mutex_init( pthread_mutex_t* pMutex,
                        const pthread_mutexattr_t* pAttr );
```

Primul parametru, `pMutex` reprezintă un pointer către obiectul mutex inițializat. Al doilea parametru, `pAttr` reprezintă un pointer către o structură ce conține atributele mutexului creat. Pentru al doilea parametru se poate transmite şi valoarea NULL, caz în

care se creează un mutex de tipul `PTHREAD_MUTEX_DEFAULT`. Valoarea returnată este 0 în caz de succes şi diferită de 0 în caz de eroare.

O dată ce obiectul mutex a fost creat, acesta poate fi închis şi eliberat prin intermediul următoarelor funcţii:

```
int pthread_mutex_lock( pthread_mutex_t* pMutex );
int pthread_mutex_unlock( pthread_mutex_t* pMutex );
```

Pentru ambele funcţii singurul parametru, `pMutex` reprezintă mutexul asupra căruia se aplică operaţiile de închidere respectiv eliberare. În ambele cazuri, valoarea returnată este 0 în caz de success şi diferită de 0 în caz de eroare.

După utilizare, obiectul mutex trebuie distrus prin apelul funcţiei `pthread_mutex_destroy()`, cu prototipul:

```
int pthread_mutex_destroy( pthread_mutex_t* pMutex );
```

Parametrul `pMutex` reprezintă adreasa obiectului mutex distrus, valoarea returnată fiind 0 în caz de succes şi diferit de 0 în caz de eroare.

În continuare, vom prezenta un exemplu complet de utilizare a atributelor şi a obiectelor mutex. Vom creea un mutex recursiv şi vom crea o secţiune critică în care accesăm o listă:

```
// Configurarea tipului obiectului mutex
pthread_mutexattr_t mutexAttr;
int nRes = pthread_mutexattr_init( &mutexAttr );
assert( 0 == nRes );
nRes = pthread_mutexattr_settype( &mutexAttr,
                                  PTHREAD_MUTEX_RECURSIVE );
assert( 0 == nRes );

// Configurarea obiectului mutex
pthread_mutex_t mutex;
nRes = pthread_mutex_init( &mutex, &mutexAttr );
assert( 0 == nRes );

// Distrugerea obiectului atribute
nRes = pthread_mutexattr_destroy( &mutexAttr );
assert( 0 == nRes );

…

// Închiderea mutexului
nRes = pthread_mutex_lock( &mutex );
assert( 0 == nRes );

// Utilizarea listei
…

// Eliberarea mutexului
nRes = pthread_mutex_unlock( &mutex );
assert( 0 == nRes );

…
```

```
// Distrugerea mutexului
nRes = pthread_mutex_destroy( &mutex );
assert( 0 == nRes );
```

8.2 Încapsularea obiectelor excluziune mutuală

Utilizarea directă a funcțiilor prezentate în sub-capitolul anterior nu este practică, mai ales în aplicațiile ce necesită o utilizare pe mai multe platforme. Din acest motiv, utilizând modelul din capitolul anterior vo realiza încapsularea obiectelor excluziune mutuală şi a operațiilor ce se aplică asupra lor prin intermediul programării orientate pe obiecte.

Denumirea clasei pe care o vom utiliza în continuare este `MyMutex`. Aceasta definește o variabilă membru de tipul unui obiect mutex. Întrucât clasa trebuie utilizată atât pe platforme Windows cât şi pe cele Unix, vom defini un tip unitar pentru ambele platforme `CRITICAL_SECTION`. Acest tip este deja disponibil pe platformele Windows, motiv pentru care pentru platforme Unix folosim următoarea definiție:

```
#if !defined(_WIN32)
      #include <pthread.h>
      typedef pthread_mutex_t CRITICAL_SECTION;
#endif
```

Pentru fiecare operație efectuată asupra mutexului asigurăm o metodă publică, rezultând următoarea declarație a clasei `MyMutex`:

```
class MyMutex
{
public:
     MyMutex( void );
     virtual ~MyMutex( void );

     void lock( void );
     void unlock( void );

private:
     CRITICAL_SECTION m_mutex;
};
```

Constructorul clasei trebuie să asigure inițializarea obiectului mutex. Codul rezultat pentru ambele platforme este următorul:

```
MyMutex::MyMutex( void )
{
#if defined(_WIN32)
     __try
     {
            InitializeCriticalSection( &m_mutex );
     }
     __except ( 1 )
     {
            assert( 0 );
```

```
    }
#else
    pthread_mutexattr_t mutexAttr;
    int nRes = pthread_mutexattr_init( &mutexAttr );
    assert( 0 == nRes );
    nRes = pthread_mutexattr_settype( &mutexAttr,
                                      PTHREAD_MUTEX_RECURSIVE );
    assert( 0 == nRes );
    nRes = pthread_mutex_init( &m_mutex, &mutexAttr );
    assert( 0 == nRes );
    nRes = pthread_mutexattr_destroy( &mutexAttr );
    assert( 0 == nRes );
#endif
}
```

Destructorul clasei asigură distrugerea obiectului mutex:

```
MyMutex::~MyMutex( void )
{
#if defined(_WIN32)
      __try
      {
            DeleteCriticalSection( &m_mutex );
      }
      __except( 1 )
      {
            assert( 0 );
      }
#else
     int nRes = pthread_mutex_destroy( &m_mutex );
     assert( 0 == nRes );
#endif
}
```

Închiderea şi eliberarea obiectului mutex se realizează prin metodele pereche, respectiv, `lock()` şi `unlock()`:

```
void MyMutex::lock( void )
{
#if defined(_WIN32)
     __try
     {
            EnterCriticalSection( &m_mutex );
     }
     __except( 1 )
     {
            assert( 0 );
     }
#else
     int nRes = pthread_mutex_lock( &m_mutex );
     assert( 0 == nRes );
#endif
}
```

```
void MyMutex::unlock( void )
{
#if defined(_WIN32)
     __try
     {
            LeaveCriticalSection( &m_mutex );
     }
     __except( 1 )
     {
            assert( 0 );
     }
#else
     int nRes = pthread_mutex_unlock( &m_mutex );
     assert( 0 == nRes );
#endif
}
```

8.3 Încapsularea obiectelor excluziune mutuală citire/scriere

Obiectele excluziune mutuală reprezintă o componentă extrem de utilă programatorilor în sincronizarea accesului la resurse comune mai multor fire de execuție. Cu toate că implementarea acestora a fost optimizată foarte mult pe parcursul anilor, în cazul accesului simultan al unui număr foarte mare de fire (e.g. 10000 fire de excuție), performanța sistemului poate fi afectată foarte mult, secțiunile critice fiind elemente de gâtuire a sistemului.

Pentru această problemă trebuie găsită o anumită rezolvare care să asigure reducerea gâtuirii, adică creșterea numărului de fire care să poată accesa o resursă simultan. Accesarea simultană a resurselor se poate realiza numai dacă în urma accesului se păstrează consistența datelor. Cel mai simplu este ca să categorizăm operațiile de accesare a resurselor. Astfel, distingem două clase mari de operații:

- Operații de citire: care nu asigură modificarea resurselor;
- Operații de scriere: care asigură moficarea/adăugarea de noi resurse.

Având aceste clase de operații, observăm că dacă mai multe fire realizează simultan operații de citire, nu este necesară sincronizarea accesului, întrucât resursele nu sunt modificate. O operație de scriere, pe de altă parte, necesită un acces exclusiv la resurse.

Implementarea unui asemenea tip de excluziune se poate realiza prin intermediul obiectelor mutex prezentate în cadrul sub-capitolului anterior. O primă instanță a clasei `MyMutex` este utilizată pentru a asigura sincronizarea accesului asupra unei variabile contor `m_nCount` utilizată la contorizarea numărului de închideri de citire.

O a doua instanță a aceleiaşi clase este utilizată pentru a semnaliza acapararea resursei pentru citire sau scriere. Denumirea clasei care încapsulează un asemenea mutex este `MyRWMutex`. Diagrama de clase prin care s-a ilustrat relația între cele două clase `MyMutex` şi `MyRWMutex` este dată în figura 8.1.

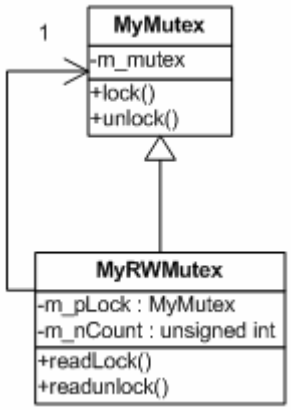

Figura 8.1 Diagrama de clase a încapsulării mutexului de citire/scriere

Declarația clasei `MyRWMutex` este următoarea:

```
#include "MyMutex.h"

class MyRWMutex:
      public MyMutex
{
public:
      MyRWMutex( void );
      virtual ~MyRWMutex( void );

      void readLock( void );
      void readUnlock( void );

private:
      MyMutex* m_pLock;
      volatile unsigned int m_nCount;
};
```

În cadrul constructorului se instațiază clasa `MyMutex`, adresa noii instanțe fiind stocate în variabila membru `m_pLock`, şi se inițializează contorul de citire cu 0:

```
MyRWMutex::MyRWMutex( void ):
      m_pLock( new MyMutex() ),
      m_nCount( 0 )
{
}
```

Destructorul va asigura distrugerea obiectului instanțiat în constructor:

```
MyRWMutex::~MyRWMutex( void )
{
      delete m_pLock;
}
```

Acapararea şi eliberarea resursei protejate pentru scriere se realizează prin apelul metodelor `lock()` şi, respectiv `unlock()`. Pentru a realiza o citire a resursei trebuie ca acest obiect mutex să fie liber. În caz contrar, firul se va bloca până la eliberarea

mutexului. O dată acaparat mutexul clasei de bază pentru o primă citire, accesul asupra acesteia este evitat de celelalte apeluri pentru citire. În cazul apariției unei cereri pentru scriere, firul apelant se va bloca până la ieşirea tuturor firelor din secțiunea de citire.

Metoda `readLock()` rezultată este următoarea:

```
void MyRWMutex::readLock( void )
{
      // Acapararea resursei contor ’m_nCount’
      m_pLock->lock();

      // La primul apel se acaparează şi mutexul din clasa de bază
      // prin care se asigură eliminarea accesului concurent de
      // citire şi scriere
      if ( !m_nCount ) {
            MyMutex::lock();
      }
      m_nCount++;

      // Eliberarea resursei contor
      m_pLock->unlock();
}

void MyRWMutex::readUnlock( void )
{
      // Acapararea resursei contor ’m_nCount’
      m_pLock->lock();

      m_nCount--;

      // Dacă este vorba despre ultimul acces de citire, se
      // eliberează şi mutexul exclusivist de citire/scriere
      if ( !m_nCount ) {
            MyMutex::unlock();
      }

      // Eliberarea resursei contor
      m_pLock->unlock();
}
```

Utilizarea unui asemenea mutex este deosebit de utilă în cazul serverelor ce trebuie să deservească mai multe mii de utilizatori. În asemenea cazuri, de regulă modificările resurselor sunt mult mai reduse la număr decât cele de citire, motiv pentru care utilizarea unui asemenea mutex va asigura creşterea performanțelor serverului.

8.4 Implementarea cozilor de mesaje

De multe ori se pune problema comunicării între fire de execuție prin intermediul mesajelor. Fiecare mesaj trebuie să conțină o descriere a scopului pentru care a fost creat şi transmis, scop care poate fi descris de regulă prin tipuri, argumente şi componente binare. În continuarea acestui capitol vom prezenta într-o primă fază încapsularea mesajelor, urmată de încapsularea cozilor de mesaje şi a cozilor de mesaje protejate prin obiecte mutex.

8.4.1 Încapsularea mesajelor

Pentru a evita limitările comunicării prin funcţii şi mesaje specifice unei anumite platforme, în continuare vom prezenta o modalitate de descriere şi încapsulare ce poate fi utilizată atât pe platformele Windows cât şi pe cele Unix.

Într-o comunicare bidirecţională de regulă se utilizează protocoale bazate pe cerere-răspuns. În asemenea protocoale fiecare mesaj trebuie etichetat corespunzător pentru a asigura o prelucrare corectă. Pornind de la aceste observaţii, mesajele pot fi clasificate în funcţie de *tipul* lor:

- Mesaje *Cerere*: mesaje care după prelucrare necesită transmiterea unui mesaj răspuns sau al unui mesaj eroare;
- Mesaje *Răspuns*: mesaje care reprezintă răspunsul la o cerere;
- Mesaje *Eroare*: mesaje care conţin codul erorii corespunzătoare mesajului cerere primit.

Comunicarea între fire este una asincronă, un fir poate emite mai multe mesaje cerere, fără a aştepta pentru fiecare în parte răspunsul, procedeu ilustrat în figura 8.2. Pentru a putea identifica mesajul cerere la prmirea răspunsului, firele trebuie să păstreze o copie locală a mesajului cerere şi în plus, pentru fiecare mesaj trebuie definit un identificator (i.e. ID) unic. În cazul ilustrat în figura 8.2, ID-urile mesajelor reprezintă textele *Message1*, *Message2* şi *Message3* iar tipurile corespunzătoare sunt ilustrate prin litere: *Q* pentru mesaje *Cerere*, *R* pentru mesaje *Răspuns* şi *E* pentru mesaje de eroare.

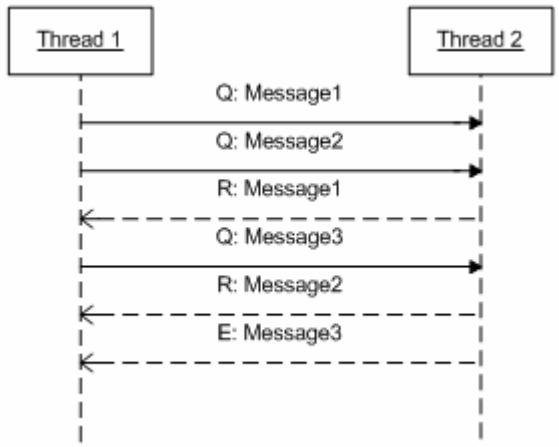

Figura 8.2 Mecanismul de comunicare asincronă între două fire de execuţie

Pe lângă aceste elemente, mesajele mai pot include o colecţie de argumente. Argumentele reprezintă componente de mesaj, pentru protocoale de nivel aplicaţie componentele având de regulă un format textual, pe lângă care pot apărea şi componente binare (e.g. protocoale pentru transfer de imagini video, pachete audio).

Toate aceste aspecte legate de mesaje vor fi în continuare încapsulate într-o clasă cu denumirea `MyMessage`, pentru a păstra consistenţa denumirilor claselor atribuite până în momentul de faţă. Declaraţia clasei trebuie să includă definirea tipurilor de mesaje, un set de metode pentru adăugarea argumentelor, un set de metode pentru interogarea

argumentelor şi un ultim set pentru ştergerea argumentelor. Declarația rezultată este următoarea:

```
class MyMessage
{
public:
	enum MSG_TYPES
	{
		QUERY_TYPE = 0,
		RESPONSE_TYPE,
		ERROR_TYPE,
		NOT_SET
	};

	enum MSGQUERY_TYPES
	{
		// Definirea mesajelor cerere
		…
	};

	enum MSGRESP_TYPES
	{
		// Definirea mesajelor răspuns
		…
	};

	enum MSGERROR_TYPES
	{
		// Definirea mesajelor de eroare
		…
	};

public:
	MyMessage( void );
	MyMessage( const unsigned int nID,
		   const unsigned int nType,
		   const unsigned int nMsg );
	virtual ~MyMessage( void );

	void addArg( const string& rParam );
	void addArg( const int nArg );
	void setRawData( const char* pData, const unsigned int nSize );
	const unsigned int getArgs( void ) const;
	string getArg( const unsigned int nArg ) const;

	// Metode de scriere
	inline void setID( const unsigned int nID ) {
		m_nID = nID;
	}
	inline void setType( const unsigned int nType ) {
		m_nType = nType;
	}
	inline void setMessage( const unsigned int nMessage ) {
		m_nMsg = nMessage;
	}

	// Metode de citire
```

```
      inline const unsigned char getID( void ) const {
            return m_nID;
      }
      inline const unsigned char getType( void ) const {
            return m_nType;
      }
      inline const unsigned char getMessage( void ) const {
            return m_nMsg;
      }
      inline const unsigned int getRawDataSize( void ) const {
            return m_nRawDataSize;
      }
      inline const char* getRawData( void ) const {
            return m_pRawData;
      }

private:
      vector< string >  m_vArgs;
      unsigned int      m_nID;
      unsigned int      m_nType;
      unsigned int      m_nMsg;
      char*             m_pRawData;
      unsigned int      m_nRawDataSize;
};
```

Din declarația dată, se poate observa că mesaje pot fi create în două moduri: neinițializate şi inițializate. În varianta neinițializată se foloseşte constructorul implicit, care setează valoarea `NOT_SET` pentru tipul mesajului. În al doilea caz, mai întâi se execută codul constructorului implicit pentru inițializarea variabilelor membru, după care se stochează ID-ul mesajului, tipul şi mesajul transmise ca parametru:

```
MyMessage::MyMessage( void )
{
      m_nID             = 0;
      m_nType           = NOT_SET;
      m_nMsg            = NOT_SET;
      m_pRawData        = NULL;
      m_nRawDataSize    = 0;
}

MyMessage::MyMessage( const unsigned int nID,
                      const unsigned int nType,
                      const unsigned int nMsg )
{
      // Execuția constructorului implicit
      MyMessage();

      // Stocarea ID-ului, tipului şi mesajului
      m_nID       = nID;
      m_nType     = nType;
      m_nMsg      = nMsg;
}
```

Destructorul trebuie să asigure eliberarea memoriei alocate datelor binare:

```
MyMessage::~MyMessage( void )
{
      if ( NULL != m_pRawData ) {
            delete[] m_pRawData;
      }
}
```

Pentru adăugarea argumentelor poate fi utilizată metoda supraîncărcată `addArg()`. Pentru aceasta se utilizează două declarații, una permite adăugarea unui şir de caractere, iar celalata permite adăugarea unui întreg ce va fi transformat în şir de caractere. Argumentele sunt stocate într-un vector STL ce asigură un acces rapid indexat asupra elementelor. Definiția acestor metode este următoarea:

```
void MyMessage::addArg( const string& rParam )
{
      m_vArgs.push_back( rParam );
}

void MyMessage::addArg( const int nArg )
{
      // Transformarea în şir de caractere
      _TCHAR pBuff[ 128 ];
      _sntprintf( pBuff, 127, _T("%d"), nArg );
      pBuff[ 127 ] = _T('\0');

      // Adăugarea unui şir de caractere
      addArg( pBuff );
}
```

Pe lângă argumentele de tip şir de caractere, pentru mesajul declarat se poate defini şi o componentă binară stocată prin apelul metodei `setRawData()`, cu definiția:

```
void MyMessage::setRawData( const char* pData,
                            const unsigned int nSize )
{
      if ( NULL != m_pRawData ) {
            delete[] m_pRawData;
      }
      m_pRawData        = const_cast< char* >( pData );
      m_nRawDataSize    = nSize;
}
```

O dată adăugate argumentele, acestea pot fi citite prin apelul metodelor `getArgs()` pentru determinarea numărului de argumente şi apelul metodei `getArg()` pentru returnarea argumentului de pe o anumită poziție. Definiția celor două metode este dată în continuare:

```
const unsigned int MyMessage::getArgs( void ) const
{
      return ( m_vArgs.size() );
}
string MyMessage::getArg( const unsigned int nArg ) const
{
      if ( nArg >= getArgs() ) {
```

```
            return _T("");
        }

        return ( m_vArgs[ nArg ] );
    }
```

În continuare, în declarația clasei au fost incluse o serie de metode *inline* pentru a asigura o execuție rapidă. Acestea asigură configurarea tipului mesajului prin apelul metodei `setType()`, configurarea mesajului prin apelul metodei `setMessage()`, citirea tipului mesajului `getType()`, citirea mesajului `getMessage()` precum şi o pereche de metode `getRawDataSize()`, `getRawData()` pentru returnarea, respectiv, a dimensiunii bufferului binar şi a adresei de început a acestuia.

Clasa mesaj prezentată este deosebit de utilă nu numai în transferul datelor între fire de execuție dar şi pentru stocarea datelor primite şi a datelor ce trebuie transmise pe rețea. Dacă protocoalele utilizate conțin o parte a componentelor prezentate, atunci clasa `MyMessage` poate fi utilizată fără probleme pentru a uşura procesarea ulterioară a mesajelor.

8.4.2 Încapsularea cozilor de mesaje sincronizate

Cozile de mesaje asigură stocarea mesajelor destinate prelucrării. Dacă această coadă este accesată din mai mult de un fir de execuție, resursa coadă de mesaje trebuie protejată prin excluziune mutuală. Astfel, încapsularea realizată în sub-capitolele anterioare pentru obiectele mutex, poate fi utilizată pentru a asigura o sincronizare a accesului la coada de mesaje, de unde va rezulta şi consistența datelor stocate şia cozii în sine.

Pentru încapsularea cozii de mesaje sincronizate vom implementa o nouă clasă, denumită `MySyncQueue`. Datele vor fi stocate într-o listă, pentru implementare vom apela la clasa `std::list` din cadrul STL. De fapt, clasa `MySyncQueue` va putea fi utilizată pentru a stoca orice tipuri de date într-o coadă sincronizată, nu numai mesaje.

Clasa `MySyncQueue` trebuie să pună la dispoziție o listă şi un obiect de sincronizare. Lista va fi folosită pentru implementarea unei cozi, utilizând doar metodele `push_back()` şi `pop_front()`. Obiectul de sincronizare va fi cel încapsulat prin intermediul clasei `MyMutex` construită în sub-capitolele anterioare. Pentru încapsularea listei şi obiectului mutex în cadrul noii clase construite vom apela la moştenirea multiplă *privată*. Prin intermediul moştenirii vom asigura un timp de viață pentru cele două componente egal cu timpul de viață a clasei `MySyncQueue`. Moştenirea *privată* va asigura că metodele moştenite nu pot fi accesate din afara clasei. Prin aceasta ne asigurăm că datele stocate în listă sunt accesate doar prin intermediul unor operații sincronizate care protejează consistența cozii, definite în cadrul noii clase.

Diagrama de clase rezultată pentru noua clasă este ilustrată în figura 8.3. Singurele metode publice din cadrul clasei `MySyncQueue` sunt `getSize()`, `sync_push_back()` şi `sync_pop_front()`. Declarația clasei este dată în continuare:

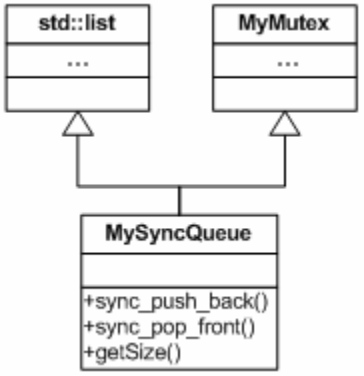

Figura 8.3 Diagrama de clase pentru încapsularea cozii sincronizate

```
#include "MyMutex.h"
#include <list>

template<typename T>
class MySyncQueue:
   private std::list<T>,
   private MyMutex
{
public:

   typedef typename std::list<T>::size_type size_type;

   MySyncQueue( void );
   virtual ~MySyncQueue( void );

   const size_type getSize( void ) const;
   void sync_push_back( const T val );
   const T sync_pop_front( void );
};
```

În declarația anterioară s-a definit `size_type` pentru a asigura un tip de dată propriu clasei pentru returnarea numărului de elemente. Definiția metodelor clasei este următoarea:

```
template<typename T>
MySyncQueue<T>::MySyncQueue( void )
{
}

template<typename T>
MySyncQueue<T>::~MySyncQueue( void )
{
}

template<typename T>
const typename MySyncQueue<T>::size_type
             MySyncQueue<T>::getSize( void ) const
{
 return ( std::list<T>::size() );
}
```

```
template<typename T>
void MySyncQueue<T>::sync_push_back( const T val )
{
 lock();
       std::list<T>::push_back( val );
 unlock();
}

template<typename T>
const T MySyncQueue<T>::sync_pop_front( void )
{
 lock();
       T val = std::list<T>::front();
       std::list<T>::pop_front();
 unlock();
 return ( val );
}
```

În definiția metodelor, se poate observa că metodele `push_back()`, `front()` şi `pop_front()` sunt incluse în secțiuni critice protejate prin obiecte mutex. Utilizarea tipului şablon `T` permite includerea oricărui tip de dată în declararea unei variabile de tipul `MySyncQueue`.

Pornind de la noua clasă definită, putem declara o coadă de mesaje sincronizată în mai multe feluri:

```
// Obiect qM declarat pe stivă (sau heap pentru var. globale)
MySyncQueue< MyMessage* > qM;
MySyncQueue< MyMessage* >* pqM = new MySyncQueue< MyMessage* >;
```

Întrucât la moştenire s-a folosit modificatorul de acces *private*, singurele metode accesibile din afară sunt cele 3 menționate. De exemplu, pentru adăugarea mesajului `QUERY_GETFRIENDS` (presupunând desigur că în `MyMessage` s-a definit în prealabil acest mesaj) de tipul `QUERY_TYPE`, cu un singur argument reprezentând denumirea grupului pentru care se cere lista prietenilor, vom utiliza secvența următoare de cod:

```
MyMessage* pmsg = new MyMessage( 1, QUERY_TYPE, QUERY_GETFRIENDS );
pmsg->addArg( _T(“Work”) );

qM.sync_push_back( pmsg );

// sau

pqM->sync_push_back( pmsg );
```

Pentru cel care procesează mesajele utilizarea se realizează în felul următor:

```
while( qM.getSize() > 0 )
{
       MyMessage* pmsg = qM.sync_pop_front();
       if ( NULL != pmsg ) {
             processMessage( pmsg );
             delete pmsg;
       }
```

```
}
// Se defineşte şi metoda de procesare a mesajului
…
void MyCommunication::processMessage( const MyMessage* pmsg )
{
	switch( pmsg->getType() )
	{
	case MyMessage::QUERY_TYPE:
		switch( pmsg->getMessage() )
		{
		case MyMessage::QUERY_GETFRIENDS:
			// Returnare lista prieteni
			…
			break;
		default:
			// Returnare eroare
		}
		break;
	default:
		// Returnare eroare
	}
}
```

Utilizarea unor clase şablon ne permite să creem o coadă sincronizată care să stocheze orice tip de dată dorim. De exemplu, am putea stoca structuri definite de utilizator, sau am putea să stocăm tipuri de date scalare precum în exemplele următoare:

```
typedef strust {
	char pBuf[ 1024 ];
	unsigned short nSize;
}MYDATA;
MySyncQueue< MYDATA >	lstData;
MySyncQueue< MYDATA* >	lstDataPtr;
MySyncQueue< int >	lstDataInt;
MySyncQueue< double >	lstDataDouble;
```

8.4.3 Încapsularea listelor de date sincronizate

În multe situații o coadă nu ne asigură necesarul de funcționalitate, mai ales finidcă suntem limitați la două operații: adăugare sfârşit şi ştergere început. De multe ori, datele sunt persistenete, ele trebuie stocate pentru a putea fi utilizate ulterior, pentru a putea fi căutate, modificate. În aceste situații se poate folosi o listă cu un obiect mutex pentru încapsulate prin moştenire într-o altă clasă `MySyncList`.

Diagrama de clase pentru încapsularea `MySyncList` este ilustrată în figura 8.4. Aceasta nu defineşte nici o metodă nouă. Include doar un constructor, un destructor şi un iterator pentru a asigura parcurgerea listei. Pentru a putea folosi metodele din cele două clase de bază, în acest caz moştenirea va fi de tipul *public*. Declarația rezultată a clasei este dată în continuare:

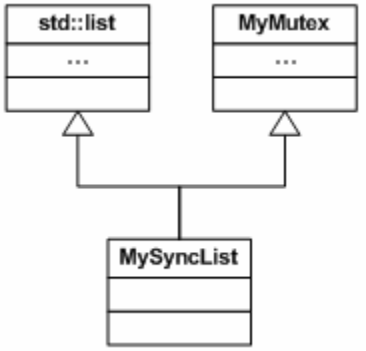

Figura 8.4 Diagrama de clase pentru încapsularea listei sincronizate

```
#include "MyMutex.h"
#include <list>

template<typename T>
class MySyncList:
   public std::list<T>,
   public MyMutex
{
public:

   typedef typename std::list<T>::iterator iterator;

   MySyncList( void );
   virtual ~MySyncList( void );

};
```

Cu definiția constructorului şi destructorului:

```
template<typename T>
MySyncList<T>::MySyncList( void )
{
}

template<typename T>
MySyncList<T>::~MySyncList( void )
{
}
```

În ceea ce priveşte utilizarea acestei clase, pentru fiecare accesare mai întâi se acaparează resursa listă, iar la terminarea utilizării se eliberează. Pentru exemplificare vom considera o listă de nume utilizator pe care se efectuează o adăugare urmată ulterior de o căutare:

```
MySyncList< std::string > lstS;
…
// Adăugarea
lstS.lock();

       lstS.push_back( _T(“User1”) );
```

```
lstS.unlock();

…
// Căutarea
bool bFound = false;
lstS.lock();

        MySyncList< std::string >::iterator pos;
        for ( pos = lstS.begin() ; pos != lstS.end() ; ++pos ) {
              if ( (*pos) == _T(“User2”) ) {
                    bFound = true;
                    break;
              }
        }

lstS.unlock();
```

O asemenea listă care asigură acces exclusiv poate fi modificată prin utilizarea unui mutex de citire/scriere. O asemenea modificare va permite citiri multiple, asigurând totodată o creştere a performanţei sistemului. Codul anterior va deveni:

```
#include "MyRWMutex.h"
#include <list>

template<typename T>
class MySyncList:
   public std::list<T>,
   public MyRWMutex
{
   …
};

MySyncList< std::string > lstS;
…
// Adăugarea
lstS.lock();

         …

lstS.unlock();

…
// Căutarea
bool bFound = false;
lstS.readLock();

         …

lstS.readUnlock();
```

Exerciţiu.

Utilizând firele de execuţie şi cozile de mesaje studiate, să se implementeze problema producător-consumator. Se defineşte un fir producător care generează numere aleatoare şi alte *N* fire de execuţie care sortează numerele după *M* metode de sortare.

Fiecare fir consumator va asigura implementarea unui singur tip de sortare. Să se compare timpul de execuție a operațiilor de sortare.