7. Crearea şi încapsularea firelor de execuție pentru sisteme Windows şi Unix

Aplicațiile prezentate până în momentul de față au fost construite pe un singur fir: firul principal al procesului. Crearea şi administrarea timpului de viață au fost ascunse de arhitectura MFC, făcând ca munca programatorului să se concentreze asupra proiectării componentelor de comunicare şi asupra celor legate de capabilitățile aplicațiilor. Cu toate că arhitectura MFC a făcut posibilă dezvoltarea rapidă a aplicațiilor, aplicațiile rezultate nu sunt portabile pe sisteme Unix. Totodată, prin ascunderea mecanismelor aplicației, programatorul nu va putea adapta aplicația la cerințe noi cărora nu le corespunde arhitectura MFC.

Din aceste motive, în cadrul acestui capitol vom prezenta o serie de metode prin care se crează şi se utilizează firele de execuție. Aceste metode vor ilustra specificul soluțiilor atât pentru sisteme Windows cât şi pentru sisteme Unix. În aplicațiile uzuale, crearea firelor nu este suficientă, trebuie să se asigure o interfață unitară de utilizare a acestora care să ascundă detaliile de implementare. Astfel, în cadrul acestui capitol vom prezenta şi încapsularea operațiilor de utilizare a firelor de execuție, încapsulări care nu necesită modificări pe platformele Windows şi Unix pentru a fi compilate şi rulate.

7.1 Crearea firelor de execuție

Există foarte multe posibilități pentru crearea firelor de execuție, posibilități date de API-urile disponibile pe platformele Windows şi Unix. În cadrul acestei secțiuni vom face referire la administrarea firelor de execuție prin intermediul a trei asemenea API-uri:

- API-ul Windows MFC pentru platforme Windows;
- API-ul Win32 pentru platforme Windows;
- API-ul Posix pentru platforme Unix.

7.1.1 Crearea firelor de execuție cu API-ul Windows MFC

În cadrul API-ului MFC se menționează două tipuri de fire de execuție ce pot fi create:

- Fire de execuție de lucru;
- Fire de execuție pentru interfețe grafice.

Prima categorie asigură execuția unei funcții stabilite de programator, fără a avea o coadă de mesaje sau posibilitatea accesării controalelor interfață grafică. Pe de altă parte, firele de execuție pentru interfețe grafice asigură tot execuția unei funcții însă au asigurată şi o coadă de mesaje şi permit accesarea directă a controalelor interfață grafică.

Pentru crearea acestor fire se utilizează funcția `AfxBeginThread()`, pentru care s-au definit două prototipuri fiecare fiind utilizat pentru crearea unui anumit tip de fir de execuție. În urma apelului acestei funcții se creează un nou obiect de tipul `CWinThread` şi

se apelează funcția Win32 `CreateThread()` despre care vom discuta în secțiunile următoare. În caz de eşuare, funcția returnează valoarea `NULL`.

Pentru fire de execuție de lucru se utilizează următoarea formă a funcției `AfxBeginThread()`:

```
CWinThread* AfxBeginThread( AFX_THREADPROC pfnThreadProc,
                            LPVOID pParam,
                            int nPriority = THREAD_PRIORITY_NORMAL,
                            UINT nStackSize = 0,
                            DWORD dwCreateFlags = 0,
                            LPSECURITY_ATTRIBUTES lpSecAttrs = NULL );
```

Primul parametru, `pfnThreadProc` reprezintă adresa funcției care va fi executată de nou fir creat. Următorul parametru, `pParam` reprezintă un pointer către adresa parametrului transmis funcției de execuție. Prin intermediul parametrului `nPriority` se va specifica prioritatea de execuție a noului fir, iar prin intermediul parametrului `nStackSize` se specifică dimensiunea stivei noului fir. Următorul parametru, `dwCreateFlags` permite specificarea unor opțiuni de creare a firului, iar ultimul parametru, `lpSecAttrs` permite specificarea atributelor de securitate pentru noul fir.

Funcția care se execută trebuie să aibă următorul prototip:

```
UINT __cdecl FunctiaFiruluiDeExecutie( LPVOID pParam );
```

unde parametrul `pParam` este acelaşi cu parametrul `pParam` transmis funcției `AfxBeginThread()`.

Pentru parametrul de prioritate, `nPriority` o valoare egală cu 0 va asigura crearea unui fir ce va rula cu aceeaşi prioritate cu firul părinte, aceasta fiind de fapt valoarea implicită `THREAD_PRIORITY_NORMAL`. Prioritățile ce pot fi specificate sunt date în tabelul 7.1.

Parametrul de dimensiune a stivei `nStackSize` permite programatorului să stabilească dimensiunea stiovei alocate firului de execuție. O valoare 0 va asigura o stivă de dimensiuni egale cu cea a stivei firului părinte.

Opțiunile de creare a firului sunt date prin intermediul parametrului `dwCreateFlags`. Valorile posibile pentru acest parametru sunt:

- `CREATE_SUSPENDED`: crearea unui fir oprit, ce poate fi pornit prin apelul metodei `CWinThread::ResumeThread()`. O asemenea opțiune este utilă în cazul în care înainte de execuția firului sunt necesare inițializări;
- `0`: firul este pornit imediat după apelul funcției de creare.

Ultimul parametru, `lpSecAttrs` permite specificare unor atribute de securitate. Aceste atribute fac referire la drepturile de acces ale firului şi asupra drepturilor de accesare a firului de către alte fire. Atributele sunt specificate sub forma unei structuri `SECURITY_ATTRIBUTES` ce trebue completată de programator. Întrucât aspectele de securitate se plasează în afara obiectivelor acestei lucrări, vom considera valoarea `NULL` pentru `lpSecAttrs` prin care se asigură moştenirea atributelor de securitate de la firul părinte.

Tabelul 7.1 Prioritatea firelor de execuție Win32

Clasa de prioritate	Valoarea	Explicații
THREAD_MODE_BACKGROUND_BEGIN	0x00010000	Începerea procesării de fundal prin scăderea ritmului de planificare a resurselor alocate firului pentru a nu afecta activitatea celorlalte fire.
THREAD_MODE_BACKGROUND_END	0x00020000	Terminarea procesării de fundal şi restabilirea priorității la valoarea dinaintea intrării în starea de procesare de fundal.
THREAD_PRIORITY_ABOVE_NORMAL	1	Prioritate cu valoarea 1 peste prioritatea normală.
THREAD_PRIORITY_BELOW_NORMAL	-1	Prioritate cu valoarea 1 sub prioritatea normală.
THREAD_PRIORITY_HIGHEST	2	Prioritate cu valoarea 2 peste prioritatea normală.
THREAD_PRIORITY_LOWEST	-2	Prioritate cu valoarea 2 sub prioritatea normală.
THREAD_PRIORITY_IDLE	-15	Prioritatea cea mai redusă.
THREAD_PRIORITY_NORMAL	0	Prioritate normală cu care sunt create firele implicit.
THREAD_PRIORITY_TIME_CRITICAL	15	Prioritatea cea mai mare.

Pentru exemplificarea utilizării acestei funcții, vom considera generarea unei secvențe de numere aleatoare în noul fir. Secvența de numere aleatoare va fi stocată într-o structură transmisă ca parametru funcției de execuție. Codul rezultat este următorul:

```
typedef struct {
      unsigned int* pV;
      unsigned int nCount;
}DATE_TH;

UINT __cdecl runStub( LPVOID pParam )
{
      DATE_TH* pD = reinterpret_cast< DATE_TH* >( pParam );
      if ( ( NULL == pD ) || ( 0 == pD->nCount ) ) {
            return 1;
      }

      srand( time( NULL ) );
      pD->pV = new unsigned int[ pD->nCount ];
      for ( int i = 0 ; i < pD->nCount ; ++i ) {
            pD->pV[ i ] = rand();
      }
      return 0;
}

… // Într-o altă funcţie se iniţializează structura şi se
  // creează firul de execuţie
DATE_TH* pD = new DATE_TH;
pD->nCount  = 2048;
pD->pV      = NULL;
```

```
if ( NULL == AfxBeginThread( runStub, pD ) ) {
      ::MessageBox(
            NULL,
            “An error was encountered while creating a new thread”,
            “Error”,
            MB_OK | MB_ICONERROR );
      return false;
}
```

Firele create prin metoda prezentată sunt distruse prin simpla revenire a lor din funcția executată. O dată cu revenirea din această metodă se distruge automat şi obiectul `CWinThread` creat prin apelul funcției `AfxBeginThread()`. Dacă nu se doreşte distrugerea obiectului `CWinThread`, la sfârşitul funcției `runStub`, înainte de revenire se poate apela funcția `AfxEndThread()`, cu prototipul:

```
void AFXAPI AfxEndThread( UINT nExitCode, BOOL bDelete = TRUE );
```

Primul parametru, `nExitCode` reprezintă codul de revenire, iar al doilea parametru, `bDelete` reprezintă o variabilă booleană ce denotă ştergerea sau menținerea obiectului în memorie după distrugerea firului. Dacă `bDelete = TRUE` (care este de altfel şi valoarea implicită, chiar dacă funcția nu este apelată), atunci obiectul nu este şters, iar programatorul poate interoga codul de ieşire a firului prin apelul funcției `GetExitCodeThread()`, având următorul prototip:

```
BOOL WINAPI GetExitCodeThread( HANDLE hThread, LPDWORD lpExitCode );
```

unde primul parametru, `hThread` reprezintă descriptorul firului, iar al doilea parametru reprezintă adresa variabilei în care se transferă codul. Dacă firul nu şi-a terminat încă execuția, valoarea transferată este `STILL_ACTIVE`. Un exemplu de utilizare a funcțiilor descrise anterior este dat în cele ce urmează:

```
UINT __cdecl runStub( LPVOID pParam )
{
      …
      // Obiectul nu este sters, codul de ieşire este ‘1’
      AfxEndThread( 1, FALSE );
      return 0;
}


…
CWinThread* pth = AfxBeginThread( runStub, pD );
if ( NULL == pth ) {
      …
}
else {
      // Aşteptăm ca firul să-şi termine execuția
      DWORD nCode = 0;
      do
      {     GetExitCodeThread( pth->m_hThread, &nCode );
            Sleep( 100 );
      }while( STILL_ACTIVE == nCode );
      delete pth;
```

```
}
```

Firele de execuție pentru interfețe grafice se creează utilizând aceeaşi funcție `AfxBeginThread()` dar cu un alt prototip:

```
CWinThread* AfxBeginThread( CRuntimeClass* pThreadClass,
                            int nPriority = THREAD_PRIORITY_NORMAL,
                            UINT nStackSize = 0,
                            DWORD dwCreateFlags = 0,
                            LPSECURITY_ATTRIBUTES lpSecAttrs = NULL );
```

Diferența între acest prototip şi cel anterior stă în parametrii. În acest caz primii doi parametrii au fost înlocuiți cu unul singur `pThreadClass` reprezentând clasa de rulare a unui obiect derivat din cadrul `CWinThread`. Crearea unui asemenea pointer se realizează prin utilizarea macroului `RUNTIME_CLASS( DenumireClasa )`.

În acest caz firul execută metoda `Run()` a clasei `CWinThread`. Această metodă extrage mesaje din coada de mesaje ataşată firului şi apelează metodele virtuale ataşate. Metodele virtuale ce pot fi suprascrise sunt următoarele:

- `ExitInstance()`: apelat înainte de distrugerea firului şi permite efectuarea unor operații de curățare, de eliberare a memoriei, de distrugere a unor obiecte, etc.;
- `InitInstance()`: apelat la crearea firului şi permite efectuarea unor operații de inițializare;
- `IsIdleMessage()`: apelat pentru a verifica dacă mesajul transmis ca parametru poate fi procesat ca mesaj *Idle*;
- `OnIdle()`: apelat când pot fi efectuate operații specifice aplicației. Aceasta este de fapt metoda ce trebuie suprascrisă pentru implementarea operațiilor executate de noul fir;
- `PreTranslateMessage()`: apelat pentru a da posibilitatea filtrării anumitor mesaje;
- `ProcessMessageFilter()`: apelat pentru interceptarea anumitor mesaje înainte ca acestea să ajungă la aplicație;
- `ProcessWndProcException()`: apelat la interceptarea excepțiilor netratate;
- `PumpMessage()`: apelat pentru administrarea cozii de mesaje;
- `Run()`: apelat pentru procesarea mesajelor.

Pentru implementarea funcționalităților firului de execuție, programatorul trebuie să suprascrie metoda `OnIdle()`, unde poate efectua operații de procesare. Această metodă nu trebuie să fie cu blocare, ea fiind apelată de câte ori nu există alte mesaje pentru procesare. Pentru ca această metodă să fie apelată în continuare, programatorul trebuie să returneze o valoare diferită de 0. Pentru valoarea 0 returnată această metodă nu mai este apelată.

Pentru exemplificarea creării unui fir de execuție prin această metodă, vom crea o clasă MFC ce moşteneşte clasa `CWinThread`. În cadrul noii clase denumite `CMyMFCThread`, suprascriem metoda virtuală `OnIdle()`, în care considerăm aceeași generare de numere aleatorii din exemplul anterior. De data aceasta însă, nu mai trebuie

să transmitem prin parametrii adresa structurii alocate, ci vom declara o variabilă membră privată `m_d`, a cărei câmpuri sunt inițializate în constructor.

După cum s-a precizat anterior, metoda `OnIdle()` este apelată de câte ori coada de mesaje a firului este goală. Parametrul acestei metode, `lCount` este un contor a cărui valoare reprezintă numărul de apeluri ale metodei `OnIdle()` între două mesaje procesate. Parametrul poate fi folosit pentru a determina intervalul de timp cât firul nu a procesat mesaje. Metoda `OnIdle()` rezultată precum şi secvența de creare a firului sunt ilustrate în cele ce urmează:

```
BOOL CMyMFCThread::OnIdle( LONG lCount )
{
      srand( time( NULL ) );
      m_d.pV = new unsigned int[ m_d.nCount ];
      for ( int i = 0 ; i < m_d.nCount ; ++i ) {
            m_d.pV[ i ] = rand();
      }

      CWinThread::OnIdle( lCount );

      // Ne asigurăm că metoda nu mai este apelată
      return FALSE;
}

… // Într-o altă funcţie se creează firul de execuţie
CMyMFCThread* pth = dynamic_cast< CMyMFCThread* >(
                        AfxBeginThread(RUNTIME_CLASS(CMyMFCThread)) );
if ( NULL == pth ) {
      ::MessageBox(
            NULL,
            “An error was encountered while creating a new thread”,
            “Error”,
            MB_OK | MB_ICONERROR );
      return false;
}
```

Terminarea firului se realizează prin apelul funcției `PostQuitMessage()` din cadrul firului, având prototipul:

```
void PostQuitMessage( int nExitCode );
```

unde `nExitCode` reprezintă codul de ieşire. Prin apelul acestei metode, în coada de mesaje a firului se adaugă un mesaj `WM_QUIT`, ce semnalează terminarea execuției. Determinarea stării firului se realizează prin utilizarea funcției `GetExitCodeThread()`, prezentată anterior. În acest caz se distruge şi obiectul automat, fără ca distrugerea acestuia să fie posibilă din exterior (i.e. destructorul este protejat – en. „protected”).

În cazul ambelor metode de creare a firelor de execuție există foarte multe situații în care terminarea firului este semnalată din exterior. În asemenea cazuri se poate recurge la utilizarea unei variabile *volatile* booleane prin care se semnalează continuarea execuției, de exemplu:

```
volatile bool g_bRunStub = true;
UINT __cdecl runStub( LPVOID pParam )
{
      // Execuţia codului firului până când g_bRunStub devine fals
      while ( g_bRunStub ) {
            …
      }
      return 0;
}
```

Pentru cazul al doilea considerăm o variabilă *volatilă* booleană membră a cărei valoare poate fi modificată din exterior (de regulă prin intermediul unor metode publice):

```
BOOL CMyMFCThread::OnIdle( LONG lCount )
{
      if ( !g_bRunStub ) {
            // Punem un mesaj de ieşire în coada de mesaje
            PostQuitMessage( 1 ) ;

            // Ne asigurăm că metoda OnIdle() nu mai este apelată
            return FALSE ;
      }
      return CWinThread::OnIdle( lCount );
}
```

7.1.2 Crearea firelor de execuţie cu API-ul Win32

API-ul MFC prezentat în secţiunea anterioară este construit peste API-ul Win32, asigurând o încapsulare a operaţiilor de creare a firelor de execuţie. API-ul Win32 defineşte funcţia `CreateThread()` pentru crearea unui nou fir de execuţie, având următorul prototip:

```
HANDLE WINAPI CreateThread( LPSECURITY_ATTRIBUTES lpThreadAttributes,
                            SIZE_T dwStackSize,
                            LPTHREAD_START_ROUTINE lpStartAddress,
                            LPVOID lpParameter,
                            DWORD dwCreationFlags,
                            LPDWORD lpThreadId );
```

Primul parametru, `lpThreadAttributes` reprezintă un pointer către o structură ce conţine atributele de securitate cu care se creează firul, o valoare `NULL` asigurând utilizarea setului de atribute implicite preluate din tokenul de securitate al utilizatorului. Parametrul `dwStackSize` permite configurarea dimensiunii stivei utilizate, o valoare egală cu 0 va asigura utilizarea dimensiunii implicite (egală de regulă cu 1MB). Următorul parametru, `lpStartAddress` reprezintă adresa funcţiei executate, prezentată mai târziu în cadrul acestui sub-capitol. Parametrul `lpParameter` reprezintă un pointer către adresa parametrului transmis funcţiei executate.

Parametrul `dwCreationFlags` are aceleaşi valori ca şi parametrul `dwCreateFlags` transmis funcţiei `AfxBeginThread()`. În cazul creării unui fir suspendat, pentru pornirea

ulterioară se foloseşte funcţia `ResumeThread()` căruia i se transmite descriptorul firului ce trebuie pornit. Pentru acest parametru se mai defineşte în plus un flag `STACK_SIZE_PARAM_IS_A_RESERVATION`, cu valoarea `0x00010000`. Acest parametru este extrem de util când sistemul trebuie să aloce un număr foarte mare de fire pentru un singur proces. În cazul utilizării valorilor implicite pentru stivă, sistemul poate aloca un număr maxim de 2048 de fire pentru un proces. Acest parametru este utilizat pentru a aloca o stivă mai mică de 1MB, ce va permite crearea mai multor fire de execuţie. Dacă acest flag nu este setat, prin intermediul parametrului `dwStackSize` se configurează dimensiunea stivei rezervate. Dacă însă flag-ul este setat se configurează dimensiunea stivei utilizate iniţial. Dimensiunea stivei utilizate iniţial se poate mări în timpul execuţiei automat în funcţie de necesarul de memorie, până la maximum dimensiunea stivei rezervate. Astfel, pentru a permite alocarea mult mai multor fire decât este posibil implicit se poate recurge la o stivă utilizată de dimensiuni reduse (e.g. 4096KB) ce poate creşte până la dimensiunea maximă de 1MB. În cadrul Microsoft Visual Studio 2005, cele două valori pot fi configurate prin intermediul proprietăţilor proiectului de link-editare (i.e. *Configuration Properties->Linker->System*). Valorile astfel configurate sunt cele utilizate în mod implicit, fiind suprascrise de valorile transmise funcţiei `CreateThread()`.

Ultimul parametru reprezintă adresa variabilei în care se stochează, la revenirea din funcţia `CreateThread()`, ID-ul noului fir creat. Dacă valoarea acestuia este `NULL`, ID-ul firului nu va fi returnat.

Pentru o execuţie cu succes, `CreateThread()` returnează descriptorul noului fir creat. Altfel, se returnează valoarea `NULL`. Funcţia executată de către noul fir creat trebuie să corespundă următorului prototip:

```
DWORD WINAPI FunctiaFiruluiDeExecutie( LPVOID lpParameter );
```

Pentru exemplificarea utilizării acestei funcţii vom considera aceeaşi problemă de generare a numerelor aleatoare ca şi în sub-capitolul anterior:

```
…
DWORD WINAPI runStub( LPVOID lpParameter )
{
	DATE_TH* pD = reinterpret_cast< DATE_TH* >( lpParameter );
	if ( ( NULL == pD ) || ( 0 == pD->nCount ) ) {
		return 1;
	}

	…
	return 0;
}

…
DATE_TH* pD = new DATE_TH;
…
// Vom creea un fir cu dimensiunea stivei utilizate de 4096KB
DWORD dw = 0;
HANDLE hThread = CreateThread( NULL,
                               4096,
                               runStub,
                               pD,
```

```
                                    STACK_SIZE_PARAM_IS_A_RESERVATION,
                                    &dw );
if ( NULL == hThread ) {
      ::MessageBox(
            NULL,
            “An error was encountered while creating a new thread”,
            “Error”,
            MB_OK | MB_ICONERROR );
      return false;
}
```

Terminarea firului se realizează la revenirea din funcția executată. Semnalizarea terminării execuției din afară se poate realiza printr-o variabilă booleană după modelul prezentat în sub-capitolul anterior. Pe lângă aceasta, după revenire trebuie dealocat descriptorul alocat prin apelul funcției `CloseHandle()`:

```
if( NULL != hThread ) {
      CloseHandle( hThread );
}
```

7.1.3 Crearea firelor de execuție cu API-ul POSIX

POSIX (en. Portable Operating System Interface for Unix) reprezintă o suită de standarde ce asigură definirea unui API unitar pentru familia de sisteme Unix. În cadrul POSIX s-a definit şi un API pentru fire de execuție cunoscute şi sub denumirea de *pthreads*.

Aplicațiile ce utilizează *pthreads* trebuie să includă `pthread.h` ce asigură definirea structurilor şi prototipurilor. La link-editare trebuie adăugată şi opțiunea `-lpthread` pentru legarea apelurilor cu biblioteca *pthreads*. Înainte de prezentarea funcției pentru crearea unui fir de execuție trebuie să spunem câteva cuvinte despre atributele ce pot fi setate prin intermediul structurii `pthread_attr_t`, definită în `pthread.h` astfel:

```
struct pthread_attr
{
  void*    stackaddr;
  size_t   stacksize;
  int      detachstate;
  struct sched_param param;
  int      inheritsched;
  int      contentionscope;
};
typedef struct pthread_attr pthread_attr_t;
```

Această structură conține o serie de câmpuri printre care se numără şi cele pentru pentru configurarea adresei stivei (i.e. `stackaddr`) şi a dimensiunii acesteia (i.e. `stacksize`). În cazul în care se dorește crearea unui fir a cărui terminare poate fi sincronizată cu un alt fir se poate utiliza câmpul `detachstate`. Acest câmp poate lua două valori: `PTHREAD_CREATE_JOINABLE` şi `PTHREAD_CREATE_DETACHED`. În primul caz valoarea returnată a firului poate fi interogată dintr-un alt fir întrucât structurile interne nu sunt distruse o dată cu terminarea execuției. În cazul al doilea, o dată cu terminarea execuției sunt distruse şi structurile interne, iar valoarea returnată nu mai poate fi

interogată. Valoarea implicită pentru care se configurează acest câmp este `PTHREAD_CREATE_JOINABLE`.

Câmpul `param` permite configurarea priorității de rulare a firului. Acesta este de fapt o structură definită în `sched.h`, ce conține un sângur câmp de valoare întreagă:

```
struct sched_param
{
  int sched_priority;
};
```

O asemenea definire sub forma unei structuri permite adăugarea ulterioară a altor câmpuri fără modificarea structurii `pthread_attr_t`. Valorile posibile pentru câmpul `param` sunt următoarele:

- `SCHED_OTHER`: prioritate uzuală (implicită), fără execuție în timp real;
- `SCHED_RR`: prioritate pentru execuție în timp real *round-robin*;
- `SCHED_FIFO`: prioritate pentru execuție în timp real *first-in first-out*.

Câmpul `inheritsched` indică sursa utilizării opțiunilor de prioritate. Valorile posibile sunt următoarele:

- `PTHREAD_EXPLICIT_SCHED`: utilizarea valorilor configurate prin câmpul `param` (implicit);
- `PTHREAD_INHERIT_SCHED`: moştenirea valorilor de prioritate de la firul părinte.

Ultimul câmp, `contentionscope` permite configurarea spațiului de concurență a proceselor. Valorile posibile sunt:

- `PTHREAD_SCOPE_SYSTEM`: spațiu de concurență sistem, ceea ce înseamnă că prioritatea de planificare se aplică pentru firele din cadrul tuturor proceselor ce rulează pe sistem (implicit);
- `PTHREAD_SCOPE_PROCESS`: spațiu de concurență pe process, însemnând că prioritatea de planificare se aplică doar asupra firelor unui singur process.

De regulă, câmpurile structurii `pthread_attr_t` nu se accesează „manual” ci utilizând un set de funcții cu denumiri intuitive:

```
int pthread_attr_init( pthread_attr_t* pAttr );

int pthread_attr_destroy( pthread_attr_t* pAttr );

int pthread_attr_setdetachstate( pthread_attr_t* pAttr,
                                 int nDetachstate );

int pthread_attr_getdetachstate( const pthread_attr_t* pAttr,
                                 int* pDetachstate );

int pthread_attr_setschedpolicy( pthread_attr_t* pAttr, int nPolicy );
```

```
int pthread_attr_getschedpolicy( const pthread_attr_t* pAttr,
                                 int* pPolicy );

int pthread_attr_setschedparam( pthread_attr_t* pAttr,
                                const struct sched_param* pParam );

int pthread_attr_getschedparam( const pthread_attr_t* pAttr,
                                struct sched_param* pParam );

int pthread_attr_setinheritsched( pthread_attr_t* pAttr,
                                  int nInherit );

int pthread_attr_getinheritsched( const pthread_attr_t* pAttr,
                                  int* pInherit );

int pthread_attr_setscope( pthread_attr_t* pAttr, int nScope );

int pthread_attr_getscope( const pthread_attr_t* pAttr, int* pAcope );
```

Având definite câmpurile structurii de atribute şi funcţiile de accesare a acestor câmpuri, putem trece la prezentarea modalităţii de creare a firelor POSIX. Crearea firelor de execuţie POSIX se realizează prin apelul funcţiei `pthread_create()`, cu prototipul:

```
int pthread_create( pthread_t* pThread,
                    const pthread_attr_t* pAttr,
                    void* (*runStub)( void* ),
                    void* pArg );
```

Primul parametru, `pThread` reprezintă un pointer către o variabilă de tipul `pthread_t` în care se va stoca descriptorul firului nou creat. Al doilea parametru, `pAttr` reprezintă un pointer către o structură de atribute prin care se specifică opţiunile de creare a firului. Următorul parametru, `runStub` reprezintă adresa funcţiei care este executată de fir, iar ultimul parametru, `pArg` reprezintă pointer către argumentul transmis ca parametru funcţiei executate. În caz de succes funcţia returnează valoarea 0, în caz contrar funcţia returnează un cod de eroare diferit de 0.

Prototipul funcţiei care este executată de către noul fir creat trebuie să corespundă următorului format:

```
void* runStub( void* pArg );
```

Pentru exemplificarea creării unui fir prin intermediul API-ului POSIX, vom considera acelaşi exemplu ca şi în cazul firelor anterioare. Vom crea un fir cu o stivă de 4096 de octeţi, prioritate uzuală, spaţiu de concurenţă sistem şi fără posibilitatea determinării valorii returnate (i.e. „detached”):

```
…
void* runStub( void* pArg )
{
      DATE_TH* pD = reinterpret_cast< DATE_TH* >( pArg );
      if ( ( NULL == pD ) || ( 0 == pD->nCount ) ) {
            return NULL;
      }
```

```
      …
      return NULL;
}

…
DATE_TH* pD = new DATE_TH;
…
// Mai întâi se completează structura de atribute
pthread_attr_t attr;
pthread_attr_init( &attr );
pthread_attr_setstacksize( &attr, 4096 );
pthread_attr_setschedpolicy( &attr, SCHED_OTHER );
pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM );
pthread_attr_setdetachstate( &attr, PTHREAD_CREATE_DETACHED );

// După care se creează firul
pthread_t th;
if ( 0 != pthread_create( &th, &attr, runStub, pD ) ) {
      printf("An error was encountered while creating a new thread");
      return false;
}
```

Terminarea execuţiei firelor POSIX se realizează la revenirea din funcţia executată. Semnalizarea terminării execuţiei din afară se poate realiza printr-o variabilă booleană după modelul prezentat în sub-capitolele anterioare. În cazul în care firul a fost creat cu posibilitate de sincronizare la terminare, după revenirea din funcţia executată structurile ataşate trebuie distruse prin apelul funcţiei `pthread_detach()`. În caz contrar, structurile sunt distruse automat la revenirea din funcţia executată.

7.2 Încapsularea firelor de execuţie

Încapsularea unei entităţi software presupune stabilirea unei interfeţe de utilizare, separarea şi ascunderea detaliilor de implementare. Încapsularea trebuie să asigure ascunderea în totalitate a implementării şi asigurarea unei interfeţe *minimale* şi *complete* de utilizare.

În domeniul programării orientate pe obiecte, încapsularea se realizează prin intermediul claselor. În continuare vom prezenta o modalitate de încapsulare a firelor de execuţie care încorporează toate cele 3 metode de creare prezentate în sub-capitolele anterioare. Din domeniul firelor MFC vom realiza doar încapsularea firelor de lucru, pentru categoria firelor destinate interfeţelor utilizator existând deja încapsularea prezentată pe parcursul acestui capitol.

În continuarea acestui capitol vom prezenta două încapsulări: încapsularea unui singur fir şi încapsularea a două fire. Încapsularea unui singur fir va asigura utilizatorului un singur fir de execuţie, iar a doua încapsulare va pune la dispoziţie două fire de execuţie în cadrul aceleiaşi clase.

7.2.1 Încapsularea unui singur fir de execuţie

Datorită diferenţelor existente între cele 3 modalităţi, codul compilat trebuie să fie diferenţiat. În acest scop vom utiliza construcţii `#if defined(simbol) – #else – #endif`. Pentru fire MFC vom utilize simbolul `_MFC`, pentru Win32 simbolul `_WIN32`, iar

pentru POSIX nu vom defini un simbol, acesta fiind ultimul caz posibil. Mediul Microsoft Visual Studio defineşte automat simbolul `_WIN32` (*Project->Properties->Configuration Properties->C/C++->Preprocessor->Preprocessor Definitions*), însă nu acesta este şi cazul simbolului `_MFC`, care trebuie definit manual dacă se optează pentru o asemenea creare.

Denumirea clasei generice pe care o creem pentru încapsulare este `MyThread`. Interfața acestei clase (i.e. declarația acesteia) trebuie să pună la dispoziția utilizatorului un set de metode publice prin care să se asigure pornirea şi oprirea firului şi un set de metode publice pentru interogarea stării firului (e.g. *pornit*, *oprit*). Implementarea funcționalității firului se va realiza în sub-clase care vor suprascrie metoda virtuală şi protejată `run()`.

La crearea firului pentru fiecare caz în parte descriptorul returnat este de un alt tip. Din acest motiv trebuie să definim o denumire comună `THANDLE` pe care să o utilizăm în cadrul implementării. Această definire se va realiza în cadrul fişierului antet al clasei pentru a asigura declararea descriptorului ca o variabilă membră a clasei:

```
#if defined(_MFC)
      #include <afxwin.h>
      typedef CWinThread* THANDLE;
#elif defined(_WIN32)
      #include <windows.h>
      typedef HANDLE THANDLE;
#else
      #include <pthread.h>
      typedef pthread_t THANDLE;
#endif
```

Pornind de la această definire, putem declara o variabilă membru de tipul `THANDLE` reprezentând descriptorul firului creat. În secțiunea de variabile membru declarăm două variabile booleane volatile pentru identificarea stării firului. Astfel, vom utiliza variabila `m_bShouldRun` pentru identificarea stării următoare (i.e. dorite) a firului (e.g. rulează, terminat) şi variabila `m_bExited` pentru identificarea stării curente a firului (e.g. rulează, terminat).

Starea următoare sau dorită a firului reprezintă starea în care utilizatorul firului doreşte să plaseze firul. Stările definite sunt: *pornit* şi *oprit*. Plasarea firului în starea de *pornit* se realizează la crearea firului. După creare, utilizatorul poate trece firul în starea *oprit*.

Trecerea firului dintr-o stare în alta nu se realizează instantaneu. Din acest motiv, starea următoare (cu precizarea că starea următoare poate fi şi starea curentă) nu este întotdeauna egală cu starea curentă, iar trecerea dintr-o stare în cealalta depinde de tipul aplicației. Pentru starea curentă definim aceleaşi stări: *pornit* şi *oprit*.

Pornind de la acestea, maşina de stare a procesului de creare, execuție şi distrugere a unui fir este ilustrată în figura 7.1. în cadrul acestei figuri *start* şi *stop* reprezintă comenzi date de utilizator pentru a porni, respectiv a opri firul de execuție, iar *stared* şi *stopped* reprezintă operații executate de fir pentru a trece în starea următoare.

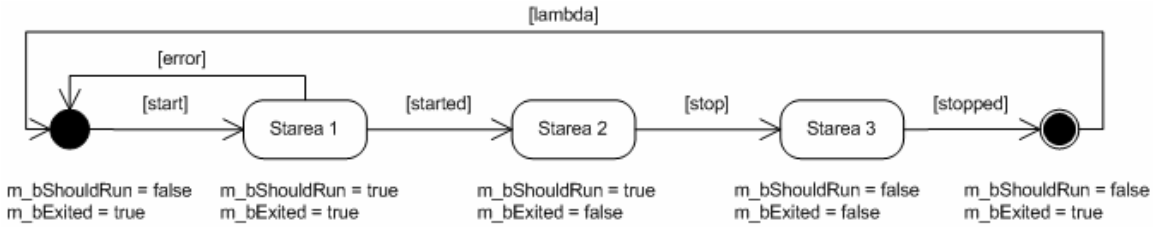

Figura 7.1 Stările unui fir de execuție

În starea inițială firul este *oprit*, iar starea următoare este *oprit*. La primirea unei comenzi de *start*, firul trece în *Starea 1*, de unde se poate întoarce în starea inițială în cazul unei erori de creare. *Starea 1* reprezintă o stare dorită *pornit* cu starea curentă *oprit*.

Se trece în *Starea 2* după lansarea în execuție a firului, starea dorită şi curentă în acest caz fiind *pornit*. Aceasta este starea în care se execută metoda suprascrisă de utilizator. Comanda de oprire a firului *stop* poate veni din partea utilizatorului sau poate fi dată de către firul de execuție. În ambele cazuri, firul trece în *Starea 3*, reprezentând o stare dorită *oprit* şi stare curentă *pornit*.

După terminarea efectivă a execuției şi la revenirea din metoda executată, firul trece în starea finală, cu starea dorită *oprit* şi starea curentă *oprit*. În această stare sunt distruse şi structurile atașate firului, acesta fiind şi motivul pentru care starea finală diferă de starea inițială. Printr-o lambda-tranziție (i.e. tranziție vidă) trecem în starea inițială, de unde firul poate fi lansat din nou în execuție.

Din punctul de vedere al creării firelor încapsulate, acestea trebuie să execute o funcție din spațiul global sau o metodă statică membră a clasei `MyThread`. Ambele soluții se utilizează în aplicații, avantajul celei de-a doua metode reprezintă faptul că se încadrează mult mai bine paradigmei programării orientate pe obiecte, denumirile metodelor executate nu vin în conflict cu denumirile altor funcții din spațiul global şi accesul la aceste metode poate fi protejată. În continuare, vom prezenta ambele metode de încapsulare.

În cazul utilizării unei funcții din spațiul global, această funcție trebuie să fie capabilă să acceseze variabile membre private ale clasei `MyThread`. Nefiind o membră a clasei, există două posibilități de rezolvare a problemei: utilizarea unor variabile publice (ceea ce contravine principiului încapsulării) sau utilizarea unor funcții *prietene*. A doua metodă este cea care va fi utilizată în continuare. Declarația clasei `MyThread` pentru acest caz este următoarea:

```
class MyThread
{
#if defined(_MFC)
      friend UINT __cdecl runStub( LPVOID mthread );
#elif defined(_WIN32)
      friend DWORD WINAPI runStub( LPVOID mthread );
#else
      friend void* runStub( void* mthread );
#endif

public:

      MyThread( void );
      virtual ~MyThread( void );
      bool start( void );
```

```
        bool stop( const unsigned int nTimeout = 0 );
        inline volatile bool& shouldRun( void ) {
              return m_bShouldRun;
        }
        inline volatile bool& isExited( void ) {
              return m_bExited;
        }

protected:
        virtual void run( void );

public:
        static const unsigned int INFINITE_WAIT;

private:
        volatile bool m_bShouldRun;
        volatile bool m_bExited;
        THANDLE m_hThread;
};
```

După cum se poate observa din declarația anterioară, funcția `runStub` executată de firul nou creat este prietenă a clasei `MyThread`. Pornirea şi oprirea firului se realizează cu metodele publice `start()` respectiv `stop()`. Interogarea stării firului se realizează tot prin două metode publice având denumiri sugestive: `shouldRun()` şi `isExited()`. Constanta `INFINITE_WAIT` este transmisă de regulă metodei `stop()` pentru a semnala aşteptarea unui timp infinit până la oprirea execuției firului, în caz contrar, metodei `stop()` i se poate transmite numărul de milisecunde maxime dorite de aşteptare. Constanta `INFINITE_WAIT` este definită în fişierul de implementare a clasei astfel:

```
const unsigned int MyThread::INFINITE_WAIT = UINT_MAX;
```

Utilizarea unor metode statice permite protejarea acecsului la metodele respective prin utilizarea modificatorului de acces `private`:

```
class MyThread
{
public:

        MyThread( void );
        virtual ~MyThread( void );
        bool start( void );
        bool stop( const unsigned int nTimeout = 0 );
        inline volatile bool& shouldRun( void ) {
              return m_bShouldRun;
        }
        inline volatile bool& isExited( void ) {
              return m_bExited;
        }

protected:
        virtual void run( void );

private:
```

```
#if defined(_MFC)
      static UINT __cdecl runStub( LPVOID mthread );
#elif defined(_WIN32)
      static DWORD WINAPI runStub( LPVOID mthread );
#else
      static void* runStub( void* mthread );
#endif

public:
      static const unsigned int INFINITE_WAIT;

private:
      volatile bool m_bShouldRun;
      volatile bool m_bExited;
      THANDLE m_hThread;
};
```

În continuarea acestui sub-capitol vom prezenta implementarea metodelor clasei `MyThread`. În cadrul acestei clase vom utiliza funcția Sleep() din spațiul global pentru deplanificarea firului un anumit număr de milisecunde. Pentru platformele Unix există o altă versiune a acestei funcții, motiv pentru care definim un macro Sleep() care să ne asigure aceeaşi funcționalitate şi pe platformele Unix:

```
#if !defined(_MFC) && !defined(_WIN32)
      #include <unistd.h>
      #define Sleep(x) usleep( 1000*(x) )
#endif
```

În constructorul clasei se inițializează variabilele membru, asigurându-se configurația din starea inițială ilustrată în figura 7.1:

```
MyThread::MyThread( void ):
      m_bShouldRun( false ),
      m_bExited( true ),
      m_hThread( NULL )
{
}
```

Metoda `start()` asigură crearea firului, cu semnalarea succesului sau eşuării operației de creare prin valoarea returnată. Această metodă înglobează toate cele 3 variante de creare şi trecerea în starea inițială în caz de eroare:

```
#define MYTHREAD_STACK_SIZE 4096
bool MyThread::start( void )
{
      m_bShouldRun = true;
      if( m_bExited )
      {
#if defined(_MFC)
            m_hThread = AfxBeginThread( runStub,
                                        this,
                                        0,
                                        MYTHREAD_STACK_SIZE );
            if( NULL == m_hThread )
```

```
#elif defined(_WIN32)
            DWORD dw;
            m_hThread = CreateThread( NULL,
                                      MYTHREAD_STACK_SIZE,
                                      runStub,
                                      this,
                                      STACK_SIZE_PARAM_IS_A_RESERVATION,
                                      &dw );
            if( NULL == m_hThread )
#else
            pthread_attr_t attr;
            pthread_attr_init( &attr );
            pthread_attr_setdetachstate( &attr,
                                         PTHREAD_CREATE_DETACHED );
            pthread_attr_setstacksize( &attr,
                        ( MYTHREAD_STACK_SIZE > PTHREAD_STACK_MIN )?
                          MYTHREAD_STACK_SIZE : PTHREAD_STACK_MIN );
            pthread_attr_setschedpolicy( &attr, SCHED_OTHER );
            pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM );
            if(0 != pthread_create( &m_hThread, &attr, runStub, this ))
#endif
            {
                  m_hThread = NULL;
                  m_bShouldRun = false;
                  m_bExited = true;
                  return false;
            }
      }
      return true;
}
```

Funcția `runStub()` executată de către noul fir creat asigură trecerea în starea curentă *pornit* şi execuția metodei `run()`. În cazul utilizării spațiului de nume global, funcția `runStub()` este definită astfel:

```
#if defined(_MFC)
      UINT __cdecl runStub( LPVOID mthread )
#elif defined(_WIN32)
      DWORD WINAPI runStub( LPVOID mthread )
#else
      void* runStub( void* mthread)
#endif
{
      MyThread* pThread = reinterpret_cast< MyThread* >( mthread );
      pThread->m_bExited = false;
      pThread->m_bShouldRun = true;
      pThread->run();
      pThread->m_bShouldRun = false;
      pThread->m_bExited = true;
#if defined(_MFC) || defined(_WIN32)
      return 0;
#else
      return NULL;
#endif
}
```

În cazul utilizării unor metode statice definiția `runStub()` devine:

```
#if defined(_MFC)
      UINT __cdecl MyThread::runStub( LPVOID mthread )
#elif defined(_WIN32)
      DWORD WINAPI MyThread::runStub( LPVOID mthread )
#else
      void* MyThread::runStub( void* mthread)
#endif
{
      MyThread* pThread = reinterpret_cast< MyThread* >( mthread );
      pThread->m_bExited = false;
      pThread->m_bShouldRun = true;
      pThread->run();
      pThread->m_bShouldRun = false;
      pThread->m_bExited = true;
#if defined(_MFC) || defined(_WIN32)
      return 0;
#else
      return NULL;
#endif
}
```

Oprirea firului de execuție creat se realizează în metoda `stop()`, cu posibilitatea stabilirii timpului de așteptare pentru terminarea firului. Valoarea returnată reprezintă noua stare curentă a firului:

```
bool MyThread::stop( const unsigned int nTimeout )
{
      m_bShouldRun = false;
      if( !m_bExited )
      {
            for( unsigned int i = 0 ;
                ( i <= nTimeout/100 ) || ( nTimeout == INFINITE_WAIT );
                ++i )
            {
                  m_bShouldRun = false;
                  if( m_bExited ) {
                        break;
                  }
                  Sleep( 10 );
            }
      }
#if defined(_MFC)
      m_hThread = NULL;
#elif defined(_WIN32)
      if( NULL != m_hThread ) {
            CloseHandle( m_hThread );
            m_hThread = NULL;
      }
#else
      m_hThread = NULL;
#endif
      return m_bExited;
}
```

Metoda protejată `run()` nu conține nici o instrucțiune întrucât ea trebuie suprascrisă de utilizator pentru implementarea funcționalității dorite:

```
void MyThread::run( void )
{
}
```

Destructorul clasei nu trebuie să asigure oprirea firului deoarece această operație este efectuată de utilizator prin apelul metodei publice `stop()` înainte de distrugerea obiectului. O asemenea abordare este una firească având în vedere simetria interfeței de utilizare a firului: pentru o metodă `start()` există o metodă pereche `stop()`. Cu toate acestea, în cazul firelor de execuție Win32, trebuie asigurată închiderea descriptorului alocat. În celelalte cazuri structurile şi descriptorii sunt automat dealocați.

```
MyThread::~MyThread( void )
{
#if !defined(_MFC) && defined(_WIN32)
      if( NULL != m_hThread ) {
            CloseHandle( m_hThread );
      }
#endif
}
```

Pentru a exemplifica utilizarea acestei clase vom moşteni clasa `MyThread` în cadrul clasei `MyCommunication`. Declararea acestei clase este următoarea:

```
#include "MyThread.h"

class MyCommunication:
      public MyThread
{
public:
      MyCommunication( void );
      virtual ~MyCommunication( void );

protected:
      // Suprascrierea metodei de execuție
      void run( void );

private:
      // Alte declarații
      …
};
```

În cadrul metodei `run()` suprascrise, execuția trebuie să testeze dacă firul trebuie rulat în continuare sau trebuie oprit:

```
void MyCommunication::run( void )
{
      while( shouldRun() )
      {
            // Execuția operațiilor specifice firului
            …
```

```
                // Deplanificarea firului
                Sleep( 1 );
            }
        }
```

Instanţierea clasei `MyCommunication`, pornirea şi oprirea firului se realizează din exteriorul clasei:

```
MyCommunication* pth = new MyCommunication();
if ( !pth->start() ) {
      ::MessageBox(
            NULL,
            "An error was encountered while creating a new thread",
            "Error",
            MB_OK | MB_ICONERROR );
      delete pth;
}
else {
      …
      pth->stop( MyThread::INFINITE_WAIT );
      delete pth;
}
```

7.2.2 Încapsularea a două fire de execuție

Pornind de la încapsularea prezentată în sub-capitolul anterior, în continuare vom construi o clasă care va asigura o interfață pentru administrarea a două fire de execuție. Diagrama de clase ce ilustrează o asemenea construcție este prezentată în figura 7.2.

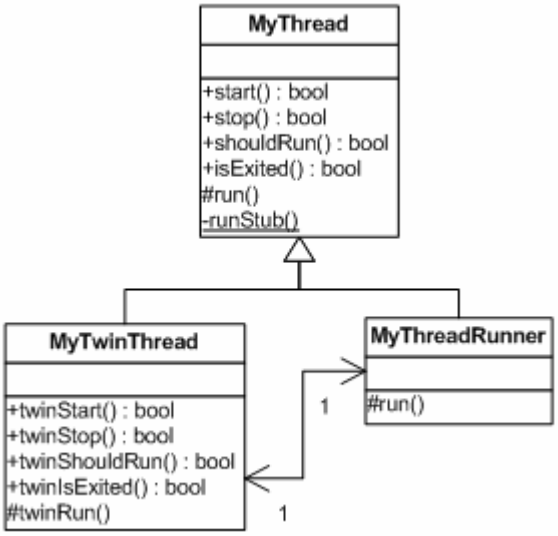

Figura 7.2 Diagrama de clase pentru încapsularea a două fire de execuție

Pentru două fire de execuție trebuie să efectuăm două moşteniri ale clasei `MyThread`. O primă moştenire se realizează în cadrul clasei `MyTwinThread`, iar o a doua moştenire în cadrul clasei `MyThreadRunner`. Clasa `MyThreadRunner` este instanțiată din cadrul clasei `MyTwinThread`. Metoda `run()` este suprascrisă doar în cadrul clasei

MyThreadRunner de unde se apelează metoda twinRun(). Prin acest mecanism, orice clasă ce moşteneşte MyTwinThread poate suprascrie două metode: run() şi twinRun(). Pornirea şi oprirea firului implementat în MyThreadRunner se realizează prin două metode publice: twinStart() şi twinStop(). Pentru interogarea stării acestui fir s-au adăugat alte două metode: twinShouldRun() şi twinIsExited().

În continuare vom prezenta implementarea celor două clase amintite. Declarația clasei MyTwinThread este următoarea:

```
#include "MyThread.h"

class MyThreadRunner;

class MyTwinThread:
      public MyThread
{
      // MyThreadRunner este declarată o clasă prietenă
      friend class MyThreadRunner;

public:
      MyTwinThread( void );
      virtual ~MyTwinThread( void );

      // Urmează metodele pentru administrarea celui de-al doilea fir
      bool twinStart( void );
      bool twinStop( const unsigned long nTimeout = 0 );
      volatile bool& twinShouldRun( void );
      volatile bool& twinIsExited( void );

protected:
      // Metoda apelată din cadrul MyThreadRunner
      virtual void twinRun( void );

private:
      MyThreadRunner* m_pRunner;
};
```

După cum se poate observa din declarația anterioară, clasa MyThreadRunner este declarată clasă prietenă pentru a permite apelul metodei protejate twinRun(). Fără o asemenea precizare această metodă trebuia declarată publică, ceea ce contravine principiului încapsulării.

Clasa MyThreadRunner este instanțiată în constructorul clasei MyTwinThread şi este distrusă în destructorul acesteia:

```
MyTwinThread::MyTwinThread( void )
{
      m_pRunner = new MyThreadRunner( this );
}

MyTwinThread::~MyTwinThread( void )
{
      delete m_pRunner;
}
```

Următoarele metode publice declarate apelează pur şi simplu metodele clasei de bază a `MyThreadRunner`:

```
bool MyTwinThread::twinStart( void )
{
      return m_pRunner->start();
}

bool MyTwinThread::twinStop( const unsigned long nTimeout )
{
      return m_pRunner->stop( nTimeout );
}

inline volatile bool& MyTwinThread::twinShouldRun( void )
{
      return m_pRunner->shouldRun();
}

inline volatile bool& MyTwinThread::twinIsExited( void )
{
      return m_pRunner->isExited();
}
```

Metoda virtuală şi protejată `twinRun()`, în cadrul clasei `MyThreadRunner` nu include nici o instrucțiune, aceasta fiind destinată suprascrierii în cadrul claselor utilizator:

```
void MyTwinThread::twinRun( void )
{
}
```

Clasa `MyThreadRunner` moşteneşte clasa `MyThread`, ia la instanțierea ei i se transmite adresa obiectului `MyTwinThread` pentru ca acesta să poată apela metoda `twinRun()`. Declarația clasei `MyThreadRunner` este următoarea:

```
#include "MyThread.h"

class MyTwinThread;

class MyThreadRunner:
      public MyThread
{
public:
      MyThreadRunner( MyTwinThread* pTwinThread );
      virtual ~MyThreadRunner( void );

protected:
      virtual void run( void );

private:
      MyTwinThread* m_pTwinThread;
};
```

În cadrul acestei clase, doar metoda `run()` prezintă interes, constructorul fiind utilizat doar pentru stocarea adresei obiectului `MyTwinThread`, constructorul neavând nici o instrucțiune:

```
MyThreadRunner::MyThreadRunner( MyTwinThread* pTwinThread ):
      m_pTwinThread( pTwinThread )
{
}

MyThreadRunner::~MyThreadRunner( void )
{
}

void MyThreadRunner::run( void )
{
      m_pTwinThread->twinRun();
}
```

Pentru exemplificarea utilizării clasei `MyTwinThread` vom considera aceeaşi clasă de comunicare `MyCommunication`. De data aceasta, în cadrul clasei vom suprascrie două metode: `run()` şi `twinRun()`, metode ce sunt executate de două fire diferite. Din acest moment, trebuie acordată o atenție deosebită accesării variabilelor membru din cadrul firelor întrucât intervin probleme de concurență, secțiuni critice, care dacă nu sunt sincronizate pot duce la rezultate nedeterminate sau chiar şi la eşuarea aplicației. Toate aceste probleme vor fi discutate în detaliu în capitolele următoare.

Declarația rezultată a clasei `MyCommunication` este următoarea:

```
#include "MyTwinThread.h"

class MyCommunication:
      public MyTwinThread
{
public:
      MyCommunication( void );
      virtual ~MyCommunication( void );

protected:
      // Suprascrierea metodelor de execuție
      void run( void );
      void twinRun( void );

private:
      // Alte declarații
      …
};
```

În cadrul metodelor `run()` şi `twinRun()` suprascrise, execuția trebuie să testeze dacă firele trebuie rulate în continuare sau trebuie oprite:

```
void MyCommunication::run( void )
{
      while( shouldRun() )
      {
```

```
                // Execuţia operaţiilor specifice firului
                …
                // Deplanificarea firului
                Sleep( 1 );
        }
    }

    void MyCommunication::twinRun( void )
    {
        while( twinShouldRun() )
        {
                // Execuţia operaţiilor specifice firului
                …
                // Deplanificarea firului
                Sleep( 1 );
        }
    }
```

Instanţierea clasei `MyCommunication`, pornirea şi oprirea celor două fire se realizează din exteriorul clasei:

```
MyCommunication* pth = new MyCommunication();
if ( !pth->start() || !pth->twinStart() ) {
    ::MessageBox(
        NULL,
        "An error was encountered while creating a new thread",
        "Error",
        MB_OK | MB_ICONERROR );
    pth->stop( MyThread::INFINITE_WAIT );
    pth->twinStop( MyThread::INFINITE_WAIT );
    delete pth;
}
else {
    …
    pth->stop( MyThread::INFINITE_WAIT );
    pth->twinStop( MyThread::INFINITE_WAIT );
    delete pth;
}
```

Încapsulările prezentate pe parcursul acestui capitol sunt deosebit de utile în cazul aplicațiilor ce trebuie să ruleze pe mai multe platforme. Simplitatea utilizării încapsulărilor le fac ideale pentru o serie de aplicații bazate pe fire de execuție. Încapsularea celor două fire este utilă în cazul aplicațiilor bazate pe socluri cu blocare, unde un fir este utilizat pentru transmitere şi unul pentru recepționare. Asemenea aplicații sunt prezentate în capitolele următoare.

Exercițiu.

Utilizând firele de execuție studiate în cadrul acestui capitol să se implementeze o aplicație pentru sortarea în paralel a mai multor vectori de întregi. Algoritmul de sortare utilizat este la alegere.