8. Laborator

Fire de execuţie şi zone critice de cod

8.1. Scopul laboratorului

Acest laborator are următoarele obiective:

1. Învăţarea practică a conceptelor fundamentale ce definesc firele de execuţie;
2. Zone critice de cod – concepte fundamentale şi utilizarea practică;
3. Modalităţi de depanare (*debug*) în cazul existenţei mai multor fire de execuţie.

Figura 8.1. Interfaţa grafică a programului

8.2. Cerinţe

Dezvoltaţi un program a cărui interfaţă grafică să fie similară cu cea din **Figura 8.1** şi care să îndeplinească următoarele cerinţe:

1. Derularea elementul de tip *progress control* să poată fi controlată de către două zone de cod plasate în două fire de execuţie diferite. Aceste 2 fire sunt independente de firul de execuţie principal al programului.
2. La un anumit moment de timp un singur fir de execuţie să aibă controlul exclusiv asupra elementului de tip *progress control* (atâta timp cât derularea, ce este executată în unul din cele două fire de execuţie, nu este finalizată cel de al doilea fir de execuţie nu va avea acces la elementul de tip *progress control*). Această cerinţă se va realiza printr-un **mecanism de sincronizare de tip *zone critice de cod***.

3. Diferenţierea dintre execuţia celor două fire se va realiza prin derularea cu o viteză superioară a unuia (în 3 secunde) față de cel de al doilea (într-o singură secundă);
4. Butonul ***Creare*** va: crea cele două fire de execuție, inițializa zona critică de cod şi va iniţializa elementul *progress control*.
5. Butonul ***Start*** va porni procesul de derulare pe elementul *progress control*.
6. Prin intermediul primului *check box*-ului se va opri sau nu execuția firului lent.
7. Fiecare fir de execuţie va accesa (derulând complet), consecutiv elementul *progress control* de 5 ori;
8. Butonul *Eliberează resurse* va elibera resursele alocate/utilizate în cadrul programului;
9. În primul *edit control* se va afișa cuanta (*quantum*) de execuție a firului lent pe care programatorul SO a asociat-o cu acest fir de execuție, atunci când se apasă butonul asociat – afișarea se va realiza în milisecunde;
10. În cel de al doilea *edit control* se va afișa prioritatea firului lent atunci când butonul asociat va fi apăsat.
11. Cu ajutorul celui de al doilea *check box* se va crește prioritatea firului lent (atunci când va fi bifat) sau se va reveni la prioritatea normală a lui (atunci când va fi debifat).

8.3. *Fire de execuţie*

Firele de execuție (*threads*) sunt entități fundamentale în cadrul unui sistem de operare. Acestea execută diferite funcții specifice în mod independent sau corelat cu alte fire de execuție. SO alocă firelor de execuție, ce rulează pe același procesor, o anumită cuantă de timp (*quantum*). Două sau mai multe fire de execuție ce au aceeași prioritate se execută în mod consecutiv de către procesor, vezi firele 2 și 3 din **Figura 8.2**. În momentul apariției unui fir de execuție de o prioritate superioară, acesta preia controlul (întrerupând execuția tuturor firelor de priorități inferioare) și se execută până la finalizare. Dacă apar mai multe fire de execuție de o prioritate superioară, acestea întrerup firele de prioritate inferioare și se vor executa în mod *round-roubin* până la finalizare.

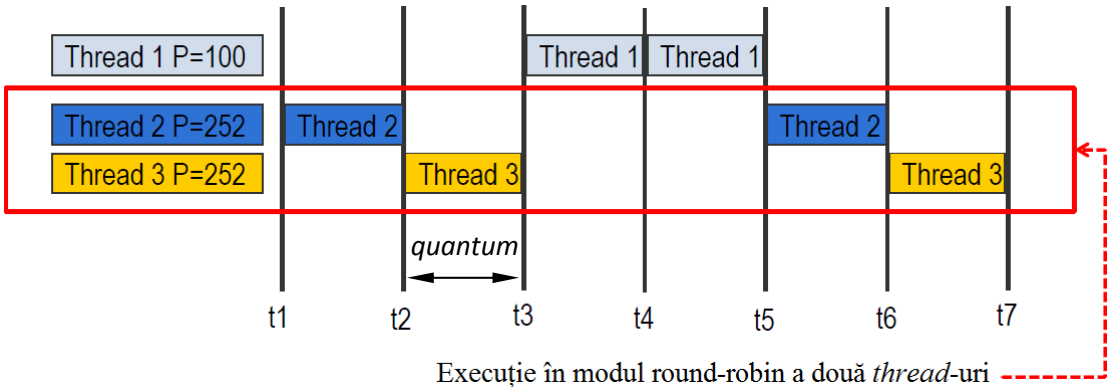

Figura 8.2. Execuție de tipul *round-robin* a două fire de execuție

Durata implicită de execuție a unui fir de execuție atunci când se rulează în mod *round-roubin* este o constantă a SO și poartă numele de *quantum*. Durata acestor felii de timp asociate diferitelor fire de execuție poate fi modificată în mod dinamic sau interogată prin intermediul următoarelor două funcții: CeSetThreadQuantum și CeGetThreadQuantum.

Pentru crearea unui nou fir de execuţie se urmează paşii:

1. Se scrie codul funcţiei care va fi transformat ulterior într-un fir de execuţie distinct. Funcţia ar putea fi de forma: void Nume_funcţie (void *);
2. Se crează firul de excuţie:

   hThread1 = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) Nume_funcție, 0, CREATE_SUSPENDED, &idThread1);

3. Definindu-se anterior *handler*-ul care va identifica în mod unic această funcție:

   HANDLE hThread1

4. Prin crearea *thread*-ului cu ajutorul argumentului CREATE_SUSPENDED acesta este creat dar nu va fi executat.
5. Pentru lansarea în execuție a *thread*-ul se apelează funcția: ResumeThread(hThread1).
6. Întotdeauna anterior finalizării execuţiei programului trebuie să eliberăm în totalitate toate resursele pe care le-am alocat anterior. Din acest motiv, în mod obligatoriu, la sfârșitul programului utilizăm funcția CloseHandle(hThread1) pentru eliberarea resurselor utilizate.

Pentru oprirea execuției unui fir pe o anumită durată, determinată, de timp se poate utiliza funcţia Sleep (timp). Astfel firul de execuție nu-și va realiza funcția pe o durată egală cu argumentul timp – această valoare reprezintă numărul de milisecunde de inactivitate a *thread*-ului.

Pentru oprirea execuției (suspendarea execuției) unui fir de execuție pe o perioadă nedeterminată este necesară utilizarea funcției SuspendThread. Fiecare fir de execuție are un contor al suspendărilor. Dacă acest contor este mai mare decât zero, firul de execuție este oprit – în caz contrar firul nu este suspendat și poate fi executat (dacă nu este blocat prin intermediul unui mecanism de sincronizare). Contorul de suspendare are o valoare maximă ce o poate lua, valoare care este dată de MAXIMUM_SUSPEND_COUNT. Prin lansarea și finalizarea cu succes a funcției SuspendThread se realizează o incrementare a acestui contor, în timp ce funcția ResumeThread decrementează acest contor asociat cu un fir de execuție suspendat.

Exercițiu: Analizați modalitate de comportare a programului când suspendați execuția firului lent atunci când: (a) se execută firul rapid, (b) se execută firul lent.

8.4. Zone critice de cod

O secţiune critică de cod este o secvenţa de cod ce accesează anumite resurse partajate (zone de memorie, dispozitive *hardware* etc.) şi implică blocarea accesului la aceste resurse a oricăror altor fire

de execuţie. Prin intermediul secţiunilor critice de cod se partajează un număr de resurse sistem comune tutror firelor de execuţie astfel încât acestea să nu fie accesate simultan de mai mult de un singur fir de execuţie.

Secţiunea critica de cod este precedata de un anumit protocol de intrare şi este urmata de un protocol specific de ieşire – aceste protocoale limitează accesarea resursei comune de către un singur fir de execuţie. Firul de execuţie care primeşte accesul în regiunea critică de cod este primul care a activat protocolul de intrare sau în cazul în care două sau mai multe fire de execuţie încearcă să intre simultan în propriile secţiuni critice ce partajează aceeaşi resursă comună, SO trebuie să asigure mecanismele necesare astfel încât doar unul dintre ele să intre, cu certitudine, în secţiunea critică de cod. De asemenea, în mod obligatoriu firul care intră în secţiunea critică de cod (*EnterCriticalSection*) va şi ieşi din aceasta (*LeaveCriticalSection*). Ca o cerinţă fundamentală a secţiunilor critice este ca firul de execuţie să nu se termine în interiorul secţiunii critice fără ca astfel protocolul de ieşire să nu fie activat şi resursa să rămână blocată pentru totdeauna.

Din puct de vedere al programului este necesar să se declare un element de tip secţiune critică (o variabilă de tip *CRITICAL_SECTION*) care trebuie, în primul pas, anterior intrării într-o secţiune critică, să fie iniţializată prin intermediul funcţiei *InitializeCriticalSection*. Ulterior atât funcţiile *EnterCriticalSection*, *LeaveCriticalSection* şi *DeleteCriticalSection* (eliberează toate resursele structurii *CRITICAL_SECTION*) vor primi adresa acestei variabile drept argument.

Prin intermediul funcţiei *TryEnterCriticalSection* se verifică dacă există un fir de execuţie într-o secţiune critică fără a se bloca accesul până la eliberarea resursei dacă existe vreun *thread* într-o astfel de zonă a codului. Dacă, nu există nici un thread într-o zonă critică, atunci *thread*-ul curent intră într-o astfel de regiune.

Prin intermediul acestor funcţii nu se creează nici un obiect în nucleul sistemului de operare. Din acest motiv acest mecanism de sincronizare este unul foarte rapid şi eficient, dar prezintă dezavantajul imposibilităţii sincronizării firelor de execuţie ce aparţin la două procese diferite.

Pentru lucru cu secţiuni critice de cod avem la dispoziţie următoarele funcţii:

1. InitializeCriticalSection (&hLock) – permite iniţializarea unei zone critice de cod şi necesită drept argument un pointer către un obiect de tip secţiune critică definit prin:
2. Definire handler: CRITICAL_SECTION hLock.
3. EnterCriticalSection(&hLock) – permite intrarea în zona critică de cod în situaţia în care nici un alt fir de execuţie nu se găseşte într-o astfel de regiune critică.
4. LeaveCriticalSection(&hLock) – Se părăseşte regiunea critică de cod, permiţând ulterior altor fire de execuţie potenţiale să intre în regiuni critice de cod controlate de acelaşi obiect.

8.5. Comentare cod

Codurile pentru funcţia asociată cu butonul ***Creare*** şi codul unuia dintre cele 2 firele de execuţie (firul lent, ce realizează derularea elementului *progress control* în 3 secunde) sunt prezentate mai jos.

```
void CFireZoneDlg::OnBnClickedButtonCreare()
{
	// TODO: Add your control notification handler code here
	hThreadL = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) Lenta,
```

```
                                    0, CREATE_SUSPENDED, &idThreadL);

        hThreadR = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) Rapida,
                                    0, CREATE_SUSPENDED, &idThreadR);

        InitializeCriticalSection (&hLock);

        Progress_adr = &m_Progress;

        m_Progress.SetRange(0, 100);
        m_Progress.SetStep(1);
}


void __stdcall Lenta(void *)
{
        lenta = 0;

        do{
                EnterCriticalSection(&hLock);

                for (int i=0; i<100; i++)
                        {
                        Progress_adr->StepIt();
                        Sleep(30);
                        }

                LeaveCriticalSection(&hLock);
                lenta++;
        } while (lenta < 5);
}
```

Deoarece atât firul de execuție primar cât și cele două fire de execuție create în cadrul firului primar nu au setate alte priorități, toate acestea au aceeași prioritate și vor fi executate în mod *round-roubin*.

Firul de execuție primar așteaptă doar apăsarea unui buton de pe interfața grafică, pentru ca ulterior să execute codul intern. Celelalte două fire de execuție vor primi aceeași cuantă de timp, dar primul care intră în zona critică de cod (execută funcția EnterCriticalSection) va avea acces total la elementul *progress control* și va blocarea accesul celui de al doilea fir de execuție la această resursă. Când cel de al doilea fir de execuție va fi executat (de către programatorul SO) și va dori să acceseze elementul *progress control*, prin intrarea lui în zona critică de cod (execuția funcției EnterCriticalSection), va determina că resursa este deja blocată de primul fir și își va termina execuția chiar dacă mai are timp din cuanta temporară alocată – el își va continua execuția doar atunci când primul fir iese din zona critică de cod (execută funcția LeaveCriticalSection) și, în acel moment, va bloca accesul primului fir la resursa partajată.

Deoarece elementul *progress control* este intern clasei dialog, el nu va putea fi accesat de zonele de cod care nu aparțin acestei clase (cele două fire de execuție create). Pentru a facilita accesul la un element intern unei clase se stochează adresa clasei asociate acestuia într-o variabilă globală care poate fi accesată atât de firul principal cât și de cele două fire de execuție create (de aici apare necesitatea zonei de cod: `Progress_adr = &m_Progress`).

8.6. Priorităţi

Un fir de execuţie este activ, deci este rulat, până în momentul în care: (a) execuţia lui se finalizează sau timpul alocat a expirat (programatorul constată terminarea felii de timp asociate), (b) este blocat – prin intermediul unui mecanism de sincronizare sau (c) este întrerupt – de o cerere de întrerupere mai prioritară sau de un alt fir de execuţie mai prioritar.

Există un număr de 256 de priorităţi ce pot fi asociate diferitelor fire de execuţie, vezi **Tabelul 8.1**. Nivelul 0 este nivelul de prioritate maximă în timp ce nivelul 255 este nivelul de prioritate minimă.

Pentru gestionarea priorităţilor firelor de execuţie se pot folosi următoarele funcţii: CeSetThreadPriority, CeGetThreadPriority şi GetThreadPriority.

Deosebirea între ultimele două funcţii este dată de faptul că funcţia CeGetThreadPriority întoarce prioritatea oricărui *thread* din sistem, valoarea întoarsă situându-se în intervalul 0 ... 255, în timp ce funcţia GetThreadPriority întoarce prioritatea firelor de execuţie utilizator ce nu sunt drivere – valoarea întoarsă fiind în intervalul 0 … 8 (corespunzător valorilor 248 ... 255).

Tabelul 8.1. Alocarea intervalelor de priorităţi în cadrul SO Windows Embedded Compact

Priorităţi	Asociată cu:
0 -96	Rezervate pentru *driver*-ele ce trebuie să lucreze în timp real
97-152	Utilizate pentru driverele standard ale sistemului de operare Windows Embedded Compact
153-247	Rezervate pentru driverele ce nu trebuie să lucreze în timp real
248-255	Rezervate pentru diferitele fire de execuţie ce nu lucrează în timp real

Firele de execuţie cu prioritate 0, nu pot fi întrerupte de nici un alt *thread* – un astfel de fir de execuţie rulează întotdeauna până la finalizarea propriilor obiective fiind firul cu prioritate maximă.

Exerciţiu: Analizaţi modalitate de comportare a programului când creşteţi prioritatea de execuţie a firului lent atunci când: (a) se execută firul rapid, (b) se execută firul lent. Explicaţi comportamentul programului.

8.7. Debug-ul programelor ce au fire multiple de execuţie

Pentru atingerea obiectivelor specifice acestui subpunct citiţi şi urmaţi instrucţiunile de la adresa web:

http://www.drdobbs.com/cpp/multithreaded-debugging-techniques/199200938?pgno=4

particularizate pentru aplicaţia dvs. Citiţi şi aplicaţi practic subcapitolele: Threads Window, Tracepoints şi Naming Threads.

Funcţia SetThreadName este utilizata pentru a da un nume specifice fiecărui *thread* în parte. Această funcţie nu este definită implicit. Codul ei îl puteţi lua din help-ul asociat cu mediul Visual Studio sau din paragraful imediat următor.

```
void SetThreadName( DWORD dwThreadID, char* threadName)
{
   Sleep(10);
   THREADNAME_INFO info;
   info.dwType = 0x1000;
   info.szName = threadName;
   info.dwThreadID = dwThreadID;
   info.dwFlags = 0;

   __try
   {
      RaiseException( MS_VC_EXCEPTION, 0, sizeof(info)/sizeof(ULONG_PTR),
(ULONG_PTR*)&info );
   }
   __except(EXCEPTION_EXECUTE_HANDLER)
   {
   }
}
```

8.8. Exerciţii

1. În cazul în care funcțiile de manipulare a *thread*-urilor întorc erori, vizualizați aceste erori în dialogbox-uri dedicate și tratați-le.
2. Realizați condiționarea logică între cele trei butoane astfel încât programul să funcționeze corect (de exemplu să nu puteți porni firele de execuție dacă anterior nu au fost create). Această condiționare se va realiza prin validarea/invalidarea butoanelor.
3. Completați programul de o așa natură astfel încât de fiecare dată când îl închideți să eliberați toate resursele utilizate (actualmente pentru a respecta această cerință trebuie să apăsați butonul *Eliberează resurse* și ulterior să închideți aplicația) – tratați mesajul *WM_DESTROY*.
4. Dacă se setează *quantum*-ul firului de execuție lent la 300 ms ce credeți ca se va întâmpla cu derularea firului de execuție lent ? Implementați în program această cerință.