Facultatea de Electronica, Telecomunicaţii şi Tehnologia Informaţiilor, Iaşi

Disciplina de Sisteme Embedded Inteligente

Laboratorul 10

Scopul laboratorului

1. Înțelegerea teoretică și practică a mecanismului de sincronizare bazat pe elementul de sincronizare de tip **mutex**;
2. Dezvoltarea abilităților de programare.

Cerințe laborator

Dezvoltați două aplicații de tip dialog în mediul Visual Studio cu ajutorul claselor MFC care să îndeplinească următoarele cerințe:

1. Să aibă fiecare dintre ele pe interfața grafică câte un element de tip *progress control* și un număr de butoane capabile să îndeplinească funcțional cerințele următoare:
2. La apăsarea unor butone de pe ambele interfețe grefice (2 butoane, 1 plasat pe o interfață, iar cel de al doilea plasat pe cealată interfață), să se genereze cîte un fir de execuție independent de către fiecare aplicație în parte capabil să permită derularea elementului *progress control*;
3. Între cele două derulări ale elementelor *progress control* va diferi doar viteza – pe o interfață grafică viteza de derulare va fi de 4 ori mai mare decât pe cealaltă interfață grafică;
4. Pe una dintre interfețele grafice va exista un buton ce va porni procese de derulare a elementelor *progress control* de pe cele două interfețe;
5. Cele două fire de execuție vor fi sincronizate prin intermediul mecanismului de tip **mutex**;
6. Derularea elementelor *progress control* va fi sub forma următoare:
    a. Primul element *progress control* se va derula de la valoarea minimă către valoarea maximă;
    b. Imediat cel de al doilea element element *progress control* se va derula de la valoarea maximă la cea minimă;
    c. Acest ciclu repetându-se de 5 ori;
7. În momentul în care cele două aplicații vor fi închise se vor elibera în totalitate resursele utilizate.

Sincronizarea între fire diferite de execuție

În mod conceptual mecanismul de sincronizare se realizează pe baza anumitor obiecte, denumite obiecte de sincronizare. Aceste entități, obiecte, utilizate în cadrul proceselor de sincronizare se pot găsi în două stări:

a. semnalate (active) sau
b. nesemnalate (sau inactive).

În cadrul mecanismelor de sincronizare, un fir de execuție interoghează starea unui anumit obiect specific utilizat în procesul de sincronizare. De exemplu[1], dacă acest obiect este în starea activă (deci este semnalizat) firul de execuție își poate continua execuția, în caz contrar (pentru un obiect nesincronizat) firul de execuție se blochează așteptând schimbarea stării elementului de sincronizare pentru a-și continua execuția.

Există în principal două funcții ce sunt utilizate, în cadrul mai multor mecanisme de sincronizare, pentru blocarea firelor de execuție. Aceste două funcții sunt:

WaitForSingleObject
WaitForMultipleObjects

Prima funcție, *WaitForSingleObject*, așteaptă până când un anumit obiect de sincronizare își schimbă starea sau chiar este deja în starea sincronizată pentru a permite continuarea execuției programului. În caz contrar firul respectiv este blocat (deci, pentru un obiect în starea inactivă). Această funcție oferă și facilitatea definirii unui interval de timp în care așteaptă (*thread*-ul fiind blocat) ca obiectul de sincronizare să devină activ, dacă acest interval temporar este depășit funcția permite continuarea execuției deblocând *thread*-ul. Funcția, *WaitForMultipleObjects*, blochează execuția unui thread până în momentul în care cel puțin unul din obiectele prin intermediul căruia se realizează sincronizarea a trecut în starea activă. Și această ultimă funcție oferă de asemenea și o facilitate temporală similară funcției *WaitForSingleObject* pentru deblocarea stării unui thread în momentul trecerii unui anumit interval de timp.

Element de sincronizare de tip Mutex

O structură de tip **mutex** (*mutual exclusion locks*) este creată să ofere un acces exclusiv al unui singur fir de execuție asupra unor resurse partajabile. Când structura **mutex** este în posesia unui anumit fir de execuție ea este „inactivă", nesemnalizată și ea devine semnalizată atunci când nici un fir de execuție nu accesează resursa partajată – această semnalizare fiind echivalentă cu un mesaj de genul „resursă disponibilă și neutilizată".

[1] ca în cazul obiectelor de tip mutex

Din descrierea anterioară se observă că un **mutex** este similar ca funcționalitate cu o zonă critică de cod cu deosebirea că un **mutex** este un obiect ce aparține nucleului (*kernel*-ului) SO. De fiecare dată când un fir de execuție dorește accesul către o resursă comună controlată prin intermediul unui mecanism **mutex** se accesează nucleul SO. Din acest motiv utilizarea unui mecanism **mutex** de partajare a resurselor generează întârzieri mult mai mari. Avantajul major este dat de posibilitatea sincronizării firelor de execuție ce aparțin la două procese diferite.

Elementele de tip **mutex** pot avea sau nu un nume, cu observația că doar structurile **mutex** ce au un nume pot sincroniza fire de execuție ce aparțin la două sau mai multe procese diferite.

Pentru crearea sau deschiderea unui **mutex** se utilizează funcția *CreateMutex* (cu sau fără nume), în timp ce eliberarea resurselor structurii **mutex**, din nucleul sistemului de operare, se realizează cu ajutorul funcției *ReleaseMutex*. Sincronizarea firelor de execuție se realizează prin intermediul funcțiilor standard *WaitForSingleObject* sau *WaitForMultipleObjects* prezentate anterior. Pentru o funcționare corectă după eliberarea resurselor trebuie să eliminăm și resursele ocupate de *handler*-ul asociat structurii **mutex** prin intermediul funcției *CloseHandle*.

Pași de urmat

Pentru lucru cu mecanisme de sincronizare de tip **mutex** avem la dispoziție următoarele funcții:

1. Se definește un *handler* ce va fi asociat cu structura de tip **mutex**: HANDLE hMutex;
2. Se crează și inițalizează obiectul **mutex**:

   hMutex = CreateMutex (NULL, FALSE, L"MyMutex");

3. În cadrul fiecărui fir de execuție ce este responsabil de un element de tip *progress control* se realizează sicronizarea prin intermediul funcție:

   WaitForSingleObject(hMutex, INFINITE);

   Atenție! Anterior realizării sicronizării (apelării funcției WaitForSingleObject) unui fir de execuție cu un alt fir de execuție, diferit față de cel în care s-a definit structura **mutex**, trebuie să avem grijă ca cele două fire să se lanseze în execuție de o așa natură astfel încât prima dată elementul de tip **mutex** să fie creat și abia ulterior să se lucreze cu el.
4. Pentru eliberarea resurselor utilizate anterior înhiderii programului apelați funcțiile: ReleaseMutex(hMutex) (are rolul eliberării resuselor din kernel astructurii **mutex**) și CloseHandle(hMutex) (rolul eliberării resurselor alocate handler-ului asociat cu **mutex**-ul).
5. Pentru verificarea funcționării corecte a programului se poate utiliza o opțiune de afișare a mesajelor în fereastra de "Debug" a mediului de dezvoltare (Visual Studio), în timp real – deci în timpul rulării programului –, folosiți macroul RETAILMSG sub forma:

   RETAILMSG(1, (TEXT("Variab. YY ia valoarea %d \n"), valVariabYY));

Primul termen este o expresie bool-eană care dacă este evaluată cu TRUE sau 1 logic atunci textul va fi afișat. Stilul și modalitatea de afișare este similară comezii printf.

Exerciții:

1. Completați programul cu toate condiționările existente între diferitele elemente virtuale (de exemplu butoane) astfel încât un utilizator să nu poată determina un comportament aberent programului (de ex. alocarea de două sau mai multe ori a firelor de execuție). Utilizați opțiunile de validare/invalidare oferite de funcțiile bibliotecă.
2. Adăugați pe una din interfețele grafice un element de tip *edit box* în care să prezentați în formă numerică derularea elementului *progress control*.