Fire de execuție în C++

Programarea concurentă reprezintă exploatarea capacitatăţii unui sistem de a executa mai multe activităţi în paralel.

O modalitate de utilizare a concurenței în programare este împărțirea activităților in procese separate. Procesele își pot trimite mesaje prin utilizarea de mecanisme de comunicare inter-proces(semnale, sockets, fişiere, etc) . Un proces reprezintă un program aflat în execuție.

Comunicarea între procese este deseori mai lenta sau mai complicată pentru ca sistemul de operare utilizează mecanisme care împiedică ca un proces să modifice datele ce aparţin altui proces.

O altă modalitate de a executa activități in paralel este utilizarea de fire de execuție în cadrul unui singur proces. Fiecare fir de execuție se execută independent unul de celălalt și poate rula o secvență diferită de instrucțiuni .

Firele de execuție din cadrul unui proces împart acelaşi spațiu de memorie cu procesul. Din acest motiv este necesar ca accesul la spațiul de date comun al procesului să fie sincronizat. Sincronizarea înseamnă ca firele de execuție să nu modifice sau să acceseze simultan zona comună de date.

1. Crearea şi utilizarea unui fir de execuție

În cadrul unui program există cel puțin un fir de execuție şi anume funcția main() :

```
#include <iostream>
using namespace std;
int main()
{
cout<<"Prima aplicatie cu fire de executie.\n";
return 0;
}
```

Crearea de fire de execuție se realizează prin utilizarea funcției CreateThread() din windows.h. Detalii: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682453(v=vs.85).aspx

Aplicația 1

```
#include <iostream>
#include <windows.h>

using namespace std;

DWORD WINAPI firDeExecutie( LPVOID lpParam )
{
	cout<<"Al doilea fir de executie!"<<endl;
	return 0;
}
```

```
int main()
{
	cout<<"Firul de executie main\n";
	DWORD threadId = 0;
	//Crearea unui thread prin utilizarea functiei CreateThread
	//Aceasta functie returneaza valoare de tip HANDLE si reprezinta un identificator al firului de executie
	HANDLE hThread = CreateThread(
		nullptr,
		0,		// utilizeaza dimensiunea implicita pentru stiva firului de executie
		firDeExecutie,	// numele functie fir de executie
		0,		// argument pentru functia firului de executie(lpParam)
		0,
		&threadId);

	/*se asteapta ca al doilea fir sa isi termine executia
	In caz contrar este posibil ca programul sa isi termine executia(firul main),
	iar al doilea fir sa nu se execute niciodata. */

	//1) Adorminea firului main prin utilizarea functiei sleep
	//Sleep(1000); //1 secunda

	/*2) Utilizarea functiei WaitForSingleObject( sau WaitForMultipleObjects) care asteapta ca handle-ul
	firului de executie sa fie semnalat in momentul in care firul isi termina executia.*/
	WaitForSingleObject(hThread, INFINITE);

	CloseHandle(hThread);

	return 0;
}
```

Exercițiu. Modificați programul de mai sus astfel încât să fie creat și un al treilea fir de execuție. Utilizați WaitForMultipleObjects pentru a aștepta execuția tuturor firelor înainte de terminarea programului.

Aplicația 2

Explicaţi funcţionalitatea şi modul de operare a exemplului de mai jos.

```
#include <windows.h>
#include <iostream>

DWORD WINAPI myThread(LPVOID lpParameter)
{
	unsigned int& myCounter = *((unsigned int*)lpParameter);
	while(myCounter < 0xFFFFFFFF) ++myCounter;
	return 0;
}
```

```
int main(void)
{
	using namespace std;

	unsigned int myCounter = 0;
	DWORD myThreadID;
	HANDLE myHandle = CreateThread(0, 0, myThread, &myCounter, 0, &myThreadID);
	char myChar = ' ';
	while(myChar != 'q') {
		cout << myCounter << endl;
		myChar = getchar();
	}

	CloseHandle(myHandle);
	return 0;
}
```

2. Sincronizarea firelor de execuție

Se consideră următoarea problemă: un producător şi un consumator îşi desfăşoară in acelaşi timp activitatea, folosind in comun un buffer de tip stivă de dimensiune fixă. Producătorul produce câte un obiect şi il adaugă în stivă. Consumatorul extrage câte un obiect din stivă .

Dificultatea problemei constă în faptul că producătorul şi consumatorul pun/iau obiecte pe/din stivă in ritmuri imprevizibile, ceea ce conduce la următoarele situaţii limită:

-producatorul încearcă să pună un obiect in stiva plină;

-consumatorul încearcă să ia un obiect din stiva goală;

Aplicația 3

//CBuffer.h

```
#include <windows.h>

class CBuffer
{
private:
	static const int BUFFER_SIZE = 10;
	int m_indexCurent;
	int m_buffer[BUFFER_SIZE];
	int val;
	CRITICAL_SECTION   m_sectiuneCritica;

public:
	CBuffer(void);
	~CBuffer(void);
	void consumator();
	void producator();
};
```

//CBuffer.cpp

```
#include "CBuffer.h"
#include <iostream>
using namespace std;

#define PRODUCATOR_SLEEP_TIME_MS 500
#define CONSUMATOR_SLEEP_TIME_MS 100


CBuffer::CBuffer(void):m_indexCurent(-1), m_buffer(), val(0), m_sectiuneCritica()
{
        InitializeCriticalSection (&m_sectiuneCritica);
}

CBuffer::~CBuffer(void)
{
       DeleteCriticalSection(&m_sectiuneCritica);
}

void CBuffer::producator()
{

       while (true)
       {
              Sleep (rand() % PRODUCATOR_SLEEP_TIME_MS);
              //EnterCriticalSection (&m_sectiuneCritica);
              if(m_indexCurent < BUFFER_SIZE - 1)
              {
                     m_indexCurent++;
                     Sleep(3);
                     m_buffer[m_indexCurent] = val;
                     val++;
                     cout<<"Producatorul: valoare "<<val<<", dimensiune buffer "<<m_indexCurent+1
<< endl;
              }
              //LeaveCriticalSection(&m_sectiuneCritica);
       }

}
void CBuffer::consumator()
{
       while (true)
       {
              //EnterCriticalSection (&m_sectiuneCritica);
              if(m_indexCurent != -1)
              {
                     int valoareCitita = m_buffer[m_indexCurent];
                     m_indexCurent--;

                     cout<<"Consumatorul:   valoare   "<<valoareCitita<<",   dimensiune   buffer
"<<m_indexCurent+1 << endl;
              }
              //LeaveCriticalSection(&m_sectiuneCritica);
              Sleep (rand() % CONSUMATOR_SLEEP_TIME_MS);

       }
}
```

//main.cpp

```
#include <windows.h>
#include <stdio.h>
#include <iostream>
#include "CBuffer.h"
using namespace std;

//Firul de executie consumator
DWORD WINAPI Consumator( LPVOID lpParam )
{
	CBuffer* buffer = static_cast<CBuffer*> (lpParam);
	buffer->consumator();
	return 0;
}

//Firul de executie Producator
DWORD WINAPI Producator( LPVOID lpParam )
{
	CBuffer* buffer = static_cast<CBuffer*> (lpParam);
	buffer->producator();
	return 0;
}

int main()
{
	static const int MAX_THREADS = 2;
	HANDLE hThread[MAX_THREADS];
	DWORD threadIdCons;
	DWORD threadIdProd;

	CBuffer* b = new CBuffer(); // bufferul comun care va fi accesat de cele doua fire de
executie

	hThread[0] = CreateThread(
		NULL,
		0,
		Producator,
		b,
		0,
		&threadIdProd);

	hThread[1] = CreateThread(
		NULL,
		0,
		Consumator,
		b,
		0,
		&threadIdCons);
	//WaitForMultipleObjects asteapta ca cele doua fire de executie sa isi termine executia
	WaitForMultipleObjects(MAX_THREADS, hThread, TRUE, INFINITE);
	delete b;

	CloseHandle(hThread[0]);
	CloseHandle(hThread[1]);

	return 0;
}
```

Observați evoluția numărului total de elemente din buffer. Explicați comportamentul programului.

Soluție: utilizarea secțiunilor critice pentru a asigura accesul mutual la membrii clasei Buffer. În acest scop eliminați comentariile unde sunt apelate funcțiile EnterCriticalSection( ) și LeaveCriticalSection( ).

Exercițiu. Comentați unul din apelurile funcției LeaveCriticalSection(). Explicați comportamentul programului.