УДК 681.3.06

С.В. Кур'ята, аспір.
Житомирський інженерно-технологічний інститут

ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ БАГАТОПОТОКОВИХ ОБЧИСЛЕНЬ У ПІДСИСТЕМІ РЕДАГУВАННЯ ВИРАЗІВ "EQUATION BUILDER" ПРОГРАМНОГО КОМПЛЕКСУ "DSR Open Lab 1.0"

(Представлено к.т.н., доц. М.М. Колодницьким)

Розглянуто особливості реалізації багатопотокових обчислень у підсистемі редагування виразів "Equation Builder" програмного комплексу "DSR Open Lab 1.0". Наведена загальна схема організації багатопотокових обчислень. Розглянуті окремі моменти програмної реалізації.

В роботі [1] було описано топологію та архітектуру інтерфейсу користувача прикладної програмної системи (ППС) для моделювання певного класу математичних структур, призначеної, головним чином, для використання у навчальному процесі – "DSR Open Lab 1.0" (Dynamical system research open laboratory – відкрита лабораторія дослідження складних систем).

Підсистема "Equation Builder" є складовою частиною інтерфейсу ППС "DSR Open Lab 1.0" [4] і відповідає за представлення опису задачі у формі, яка максимально наближена до предметної області системи, що досліджується, а також за виведення результатів обчислень. Застосування підсистеми "Equation Builder" для опису всіх математичних структур надає програмному засобові певної універсальності, а застосування запропонованої у роботі [2] класифікації (типології) математичних моделей дозволяє виділити підмножини спеціалізованих панелей інструментів для представлення конкретної задачі. Програмно підсистема "Equation Builder" реалізовувалась із застосуванням технології ActiveX. З цієї точки зору, "Equation Builder" є Active Document Server із реалізацією певного набору методів OLE Automation, що є важливим для зручного використання цієї підсистеми в ППС "DSR Open Lab 1.0".

Підсистема "Equation Builder" може бути використана як для вводу даних [3], так і для відображення результатів обчислень. В першому випадку, як правило, користувач вводить невеликі обсяги інформації й паралельна обробка даних не потрібна. У другому випадку, коли необхідно, наприклад, відобразити результати обчислення функції, що задана на певному інтервалі, з великою точністю або будь-які інші дані обчислень, що мають великій об'єм, для обробки таких початкових даних необхідно досить багато часу. Тому стає очевидною необхідність розпаралелювання процесу оброки інформації для підвищення потужності та швидкодії системи. Тобто, необхідно відокремити процес обробки початкових даних від головного потоку та перенести такі обчислення в допоміжні потоки.

Як відомо, будь-яка програма має, в крайньому випадку, один потік, який є первинним, або головним. Крім головного, програма може запускати скільки завгодно допоміжних потоків (ниток або гілок), що виконуються в середині програми. У багатозадачному програмному додатку кожний потік (задача) має свій власний стек та працює незалежно від будь-яких інших потоків, які можуть бути запущені у даній програмі. При роботі з бібліотекою MFC розрізняють:

потоки з користувацьким інтерфейсом, які працюють з повідомленнями та, як правило, вирішують задачі, що пов'язані з організацією інтерфейсу;

робочі потоки, тобто потоки, які не мають обробника повідомлень.

Як правило, допоміжний потік вирішує деяку задачу для головної програми, що передбачає існування деякого каналу зв'язку між програмою та потоком, що був породжений нею. Одночасне використання в програмі декількох потоків може призвести до виникнення серйозних проблем. Наприклад, як заборонити одночасний доступ двох потоків до одних і тих самих даних? Що трапиться, якщо в той момент, коли один потік ще не завершив процедуру поновлення деяких даних, другий потік спробує ці дані зчитати? Можна впевнено сказати, що дані, зчитані другим потоком, будуть некоректними, тому що лише деяка їх частина була на даний момент поновлена. Забезпечення коректної роботи потоків називається синхронізацією потоків. Для організації такої роботи можна застосовувати декілька підходів:

використання глобальних змінних;
використання об'єктів синхронізації;
обмін повідомленнями.

До об'єктів синхронізації відносяться: об'єкти подій, критичні секції, м'ютекси, семафори.

У підсистемі редагування виразів "Equation Builder" багатопотокова архітектура використовується, коли програма працює в режимі сервера відображення результатів обчислень. Це пов'язано з тим, що саме в цьому режимі програмі доводиться виконувати найбільшу кількість розрахунків та перетворень. Розглянемо загальну схему роботи підсистеми "Equation Builder" у режимі сервера звітів (рис. 1).

При побудові архітектури програми ми виходили з таких міркувань: усі функції інтерфейсу повинен виконувати головний потік програми, а всі додаткові обчислення виконуються у додаткових потоках. Виходячи з цього, на головний потік програми були покладені такі функції:

отримання вхідних даних із зовнішнього середовища (за допомогою методів OLE Automation);
створення допоміжного потоку та передача йому початкових даних;
контроль та синхронізація роботи потоків;
відображення результатів роботи допоміжних потоків.

Рис. 1. Загальна схема організації багатопотокових обчислень

Отже, коли головний потік через метод OLE Automation, який називається AddToBuilderBuffer (рис. 2), отримує дані для відображення, він виконує такі дії:

створює об'єкт класу CAddBufferThread, що представляє допоміжний потік;
передає об'єкту потоку масив команд, покажчик на вікно відображення даних, покажчик на об'єкт документа програми;
викликає функцію CreateThread класу CAddBufferThread для безпосереднього створення потоку та його запуску.

```
void CDSREquBDoc::AddToBuilderBuffer(LPCTSTR buff)
{
      CAddBufferThread* pThread;
      pThread = new CAddBufferThread(new string(buff), GetView(), this);
      if(!pThread->CreateThread()){
            AfxMessageBox(IDS_THREAD_ERROR_CANNOT_CREATE);
            delete pThread;
            return;
      }//end if
}
```

Рис. 2. Реалізація методу OLE Automation AddToBuilderBuffer

Тепер розглянемо організацію та реалізацію архітектури допоміжного потоку. Як зазначалося вище, вторинний потік в програмі представлений класом CAddBufferThread. Протокол опису даного класу наведено на рис. 3. Як бачимо, клас CAddBufferThread успадковується від стандартного класу CWinThread бібліотеки Microsoft Foundation Class (MFC) [5].

```
class CAddBufferThread : public CwinThread
{
public:
      DECLARE_DYNAMIC(CAddBufferThread)
      CAddBufferThread(string* pBuff, CView* pView, CDSREquBDoc* pDoc);
      virtual ~CAddBufferThread();
public:
      public:
      virtual BOOL InitInstance();
      virtual int ExitInstance();
protected:
      void UpdateObList(CObList& list, CDSREquMisc* pMisc);
      void AddToBuilderBuffer();

      string*                 m_pBuff;
      CView*                  m_pView;
      CDSREquBDoc*      m_pDoc;

      DECLARE_MESSAGE_MAP()
};
```

Рис. 3. Протокол опису класу CAddBufferThread

Коли потік буде запущено на виконання, система автоматично викличе функцію InitInstance() класу потоку. Тому зручно розмістити виклик процедури, що буде виконувати обробку отриманих даних, саме у цій функції (рис. 4), а потім повернути значення FALSE для завершення роботи потоку.

```
BOOL CAddBufferThread::InitInstance()
{
      m_pDoc->AddThread(this);
      AddToBuilderBuffer();
      m_pDoc->DeleteThread(this);

      return FALSE;
}
```

Рис. 4. Функція InitInstance( ) класу CAddBufferThread

Для обробки даних функція InitInstance() викликає іншу функцію класу AddToBuilderBuffer(). Функція AddToBuilderBuffer() виконує такі дії (рис. 5):

створює об'єкт класу CSingleLock бібліотеки MFC та викликає функцію-член класу Lock() для блокування доступу до даних з інших потоків;

створює об'єкт класу CTextCompiler, який аналізує список переданих команд, та будує за цим списком дерево об'єктів;

викликає функцію AddFromObList для передачі побудованого дерева об'єктів в головний потік програми;

проводить перерахунок координат об'єктів з врахуванням параметрів оточуючого середовища;

створює об'єкт класу CEvent (об'єкт події);

посилає головному потокові програми повідомлення про необхідність відображення нових даних;

переходить в стан очікування завершення відображення даних головним потоком;

викликає функцію Unlock() класу CSingleLock для розблокування доступу до даних з інших потоків.

```
void CAddBufferThread::AddToBuilderBuffer()
{
	CEquMetric m;
	CDSREquSNode* pSNode;
	CSingleLock sLock(&m_pDoc->m_mutex);

	sLock.Lock();

	BeginWaitCursor();

	((CDSREquBView*)m_pView)->m_bDrawLoadingDlg = TRUE;
	((CDSREquBView*)m_pView)->DrawLoadingDlg();

	CTextCompiler *pComp = new CTextCompiler(m_pDoc, *m_pBuff);
	pSNode = (CDSREquSNode*)m_pDoc->m_pEquControl->GetRoot()->m_listChild.GetHead();

	UpdateObList(*pComp->GetObjectList(), pSNode->m_pMisc);
	m_pDoc->UpdateAddToBuffProgress(96);

	pSNode->AddFromObList(pComp->GetObjectList());
	m_pDoc->UpdateAddToBuffProgress(98);

	delete pComp;
	m_pDoc->m_pEquControl->GetRoot()->UpdateCoords(CPoint(CX_MARGIN,
                                                       CY_MARGIN));
	m_pDoc->UpdateAddToBuffProgress(99);

	if(m_pDoc->m_paperWidth != -1)
	{
		CClientDC dc(m_pView);
		CSize size(m_pDoc->m_paperWidth, m_pDoc->m_paperWidth);
		m_pView->OnPrepareDC(&dc);
		dc.HIMETRICtoLP(&size);
		pSNode->Wrap(size.cx);
	}//end if

	m_pDoc->UpdateAddToBuffProgress(100);
	((CDSREquBView*)m_pView)->m_bDrawLoadingDlg = FALSE;

	m_pDoc->m_pUpdatedEvent = new CEvent;
	::PostMessage(m_pView->m_hWnd, MSG_ADD_TO_BUFFER, 0, 0);
	::WaitForSingleObject(m_pDoc->m_pUpdatedEvent->m_hObject, INFINITE);

	delete m_pDoc->m_pUpdatedEvent;
	m_pDoc->m_pUpdatedEvent = NULL;

	EndWaitCursor();

	sLock.Unlock();
}
```

Рис. 5. Функція AddToBuilderBuffer( ) класу CAddBufferThread

Після закінчення роботи допоміжного потоку головний потік проводить поновлення своїх внутрішніх даних, відображення результатів розрахунків та знищення допоміжного потоку. Для синхронізації роботи потоків та забезпечення коректності доступу до спільних даних були використані такі об'єкти синхронізації, як м'ютекси та події.

Приклад роботи підсистеми редагування виразів "Equation Builder" в складі прикладної програмної системи "DSR Open Lab 1.0" під час обробки (завантаження) даних наведено на рис. 6. На ньому зображена робота підсистеми "Equation Builder" в режимі відображення результатів обчислень.

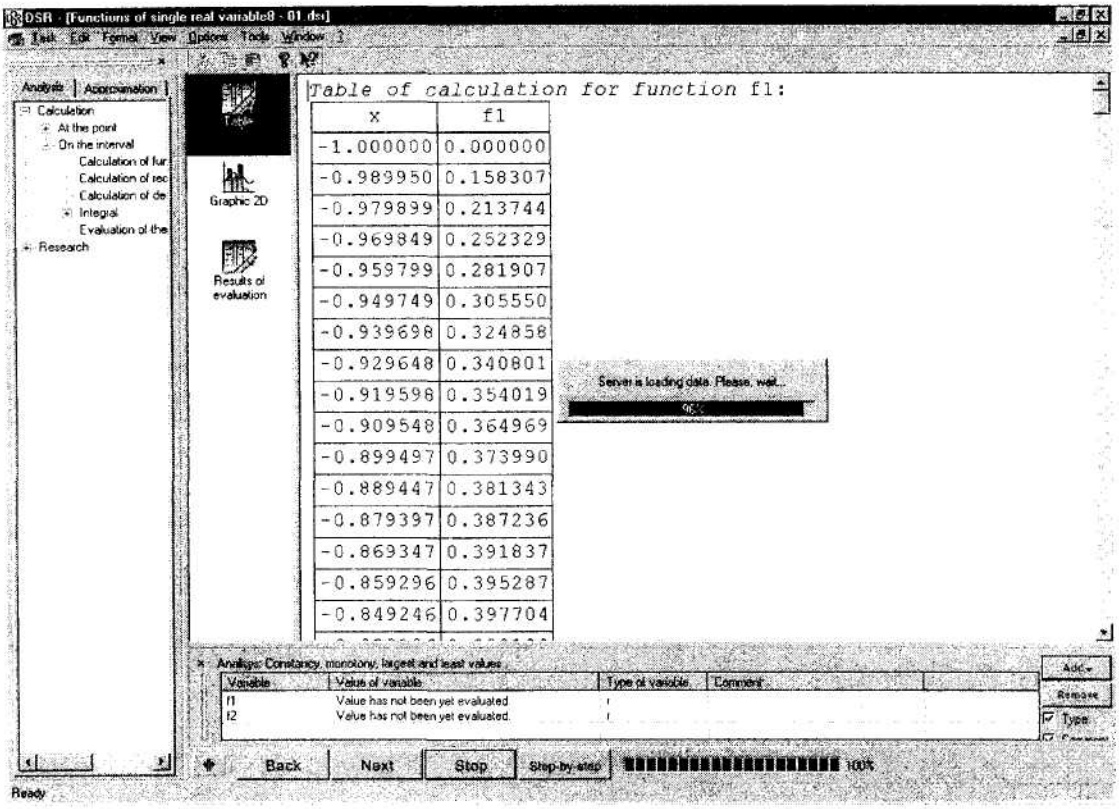

Рис. 6. Приклад роботи підсистеми "Equation Builder" в складі ППС "DSR Open Lab 1.0"

Отже, не дивлячись на те, що використання багатопотокових обчислень підвищує складність розробки програмних продуктів, їх застосування дає суттєвий виграш у роботі програми. Крім того, за рахунок впровадження багатопотокової архітектури ми збільшуємо кількість сервісних функцій програмного продукту, надаємо користувачу можливість вирішувати декілька задач одночасно, що є суттєвою перевагою, в порівнянні зі звичайними однопотоковими програмами.

ЛІТЕРАТУРА:

1. *Колодницький М.М.* Тривимірна компонентна архітектура прикладної програмної системи "Dsr Open Lab 1.0" як втілення концепцій реінженерії // Теорія програмування. – 1998. – № 4. – С. 37–45.
2. *Колодницький М.М.* Типологія математичних моделей технічних систем. Частина 2 // Вісник ЖІТІ. – 1998. – № 7. – С. 208–218.
3. *Кур'ята С., Іваницький І., Рожик О.* Особливості реалізації підсистеми редагування виразів в програмному комплексі "DSR Open Lab 1.0" // Праці 1-ї національної науково-практичної конференції студентів та аспірантів "Системний аналіз та інформаційні технології", 28–29 червня 1999 р. – Київ, 1999. – С. 76–77.
4. *Kolodnytsky M., Ivanitsky I., Kovalchuk A., Kuryata S., Levitsky V.* "DSR Open Lab 1.0" – software system for simulation // Proceedings of the 21st International Conference on Information Technology Interfaces, Pula, Chroatia, 1999. – P. 31.
5. *Грегори К.* Использование Visual C++ 5. – К.: Диалектика, 1997. – 816 с.

КУР'ЯТА Сергій Валерійович – аспірант Житомирського інженерно-технологічного інституту.

Наукові інтереси:

- комп'ютерні інформаційні технології;
- моделювання та розв'язок задач за допомогою обчислювальної техніки;
- математичне моделювання нелінійних ланцюгів.

Подано 16.07.2000