Приемы параллельного программирования в Borland Delphi

Гомозов А.В., ДонНТУ, каф. ЭВМ
Руководитель: Святный В.А., проф. каф. ЭВМ
okidoky@ukr.net

Abstract

Gomozov A. The methods of parallel programming in Borland Delphi.

The parallel programming is programming in terms of parallel processes and their cooperation and adds to traditional sequential programming a new dimension and gives to programmer the instrument of decomposition of complicated tasks, and many of them describes naturally and comfortably as a group of processes which cooperating with each other. Generally marking out two classes of systems needing parallelization, there are synchronous and anisochronous. The demand in synchronous systems arises in matric, computing or searching algorithms for speed increasing of getting results. The realization of synchronous systems concerned with multiprocessing. The demand in anisochronous parallelization appears in interactive tasks and real time tasks. The feature of these tasks is following – the separate components of task is in event waiting state the grater part of time, this fact allows to bigger number of processes to separate the processing time of single processor.

Введение

Параллельное программирование - программирование в терминах параллельных процессов и их взаимодействий - добавляет к традиционному последовательному программированию новое измерение и дает программисту инструмент декомпозиции сложных задач, многие из которых естественно и удобно описываются как группа взаимодействующих друг с другом параллельных процессов. Обычно выделяют два больших класса систем, требующих распараллеливания - синхронные и асинхронные. Потребность в синхронных системах (их называют еще векторными, матричными или системами массового параллелизма) возникает наиболее часто в матричных вычислительных или поисковых алгоритмах для увеличения скорости получения решения. Адекватная реализация синхронных систем напрямую связана с многопроцессорностью. Потребность в асинхронном распараллеливании наиболее ярко проявляется в интерактивных задачах и задачах управления в реальном времени. Особенность этих задач, которая позволяет эффективно применять параллельное программирование даже на однопроцессорной машине, состоит в том, что отдельные компоненты задачи большую часть времени проводят в состоянии ожидания каких-либо событий - именно это позволяет большому числу процессов эффективно разделять процессорное время единственного процессора.

Существуют языки, которые непосредственно поддерживают конструкции параллельного программирования, например, Ада и Оккам, но первый - чересчур громоздкий и сложный, для него нет доступного и качественного компилятора, работающего в среде Windows, а второй - Оккам - достаточно специфичный язык, разработанный специально для программирования транспьютеров фирмы Inmos. В тоже время есть доступные и качественные Windows-инструменты: объектно-ориентированные языки последовательного программирования C++ и Delphi, которые можно естественно расширить для реализации концепций параллельного программирования, используя понятие класса. Далее описывается библиотека классов Gala для Delphi.

Реализация библиотеки Gala опирается на низкоуровневые возможности параллельного

программирования, предоставляемые платформой Win32 (ОС Windows 95, 98, NT, 2000) - потоки, сигналы, мьютексы, семафоры, критические секции, сообщения. Библиотека позволяет выполнять декомпозицию задачи с использованием асинхронных взаимодействующих параллельных процессов и разделяемых ресурсов, существующих в рамках одной Windows-программы. Delphi предоставляет класс TThread, но его функциональность ограничивается, в основном, взаимодействием с VCL-компонентами, которые не являются потокобезопасными. TThread не дает никаких методов взаимодействия с другими параллельными процессами (потоками). Этот класс удобно использовать, когда в программе малое количество статически создаваемых процессов (потоков), взаимодействующих с основным VCL-потоком и не взаимодействующих между собой. Если же задача раскладывается на достаточно большое число динамически создаваемых, уничтожаемых и взаимодействующих между собой процессов, то использование TThread становится очень сложным и ненадежным. Непосредственное же использование разнообразных низкоуровневых средств синхронизации Win32 чрезвычайно трудоемко, чревато ошибками и громоздко (хотя и более эффективно).

Описание библиотеки классов Gala для Delphi

ГалаТеатр - это то место, где происходит рождение, жизнь и смерть ГалаПроцессов. В программе может быть только один объект GalaTheater (так же как объект Application). ГалаТеатр позволяет объединять процессы в группы-кластеры. Как правило, процессы взаимодействуют только внутри группы. В программе может быть одна или несколько групп. ГалаТеатр предоставляет методы для создания и завершения группы процессов, отладочной трассировки и протоколирования работы процессов.

ГалаПроцесс - это основное действующее лицо в ГалаТеатре. Все процессы в ГалаТеатре порождаются от базового класса TGalaProcess, который имеет виртуальный абстрактный метод Execute - именно в нем описывается алгоритм функционирования процесса. Базовый ГалаПроцесс инкапсулирует поток (thread) Win32 без использования Delphi-класса TThread и имеет методы для:
завершения процесса, реализации действий при нормальном и принудительном завершении,
получения и изменения его приоритета,
приостановки, запуска и приостановки на заданный промежуток времени,
взаимодействия с основным VCL-потоком, с другими параллельными процессами и с разделяемыми ресурсами
отладочной трассировки
Объект ГалаПроцесса явно создается конструктором, но не уничтожается явно при завершении процесса, а только освобождает основные системные ресурсы. Уничтожение объектов ГалаПроцессов выполняется ГалаТеатром и только для всей группы взаимодействующих процессов - это позволяет корректно обрабатывать ситуации взаимодействия с уже завершенным процессом. Операция принудительного завершения заканчивает процесс даже в том случае, если он приостановлен или ожидает какого-либо события. Принудительное завершение процесса выглядит как директива-предписание, но завершающие действия выполняются процессом самостоятельно. ГалаПроцесс может владеть каналами для взаимодействия с другими ГалаПроцессами и посылать сообщения основному VCL-потоку. Для взаимодействия с другими параллельными процессами и разделяемыми ресурсами используется модель асимметричного рандеву. По сравнению с симметричным рандеву, асимметричное рандеву имеет три важных преимущества. Во-первых, это позволяет явно разделить процессы на два больших класса - клиенты и серверы, во-вторых, позволяет делать независимую (библиотечную) реализацию серверов, так как при асимметричном рандеву серверам не требуется знание своих клиентов и, в-третьих, унифицировать взаимодействия Процесс-Процесс и Процесс-РазделяемыйРесурс.

Для выполнения недетерминированного ожидания по своим каналам ГалаПроцесс имеет метод AlternativeAccept. Метод AlternativeAccept ожидает первого поступившего запроса по заданным каналам или возбуждает исключительную ситуацию EGalaTimeout, если взаимодействие не произошло за указанное время.

ГалаКонтейнер - пассивный ресурс, разделяемый несколькими ГалаПроцессами. Объекты этого класса порождаются от базового класса TGalaContainer. ГалаКонтейнер может владеть каналами, посредством которых ГалаПроцессы получают доступ к разделяемому ресурсу. Сам контейнер ничего не делает, он только владеет каналами, которые все и делают. Каналы ГалаКонтейнера позволяют выполнять доступ к совместно используемым данным одновременно многим процессам в режиме взаимного исключения. Если контейнер не может в данный момент времени удовлетворить запрос, то он приостанавливает вызывающий процесс, а после завершения очередной операции в контейнере активизирует все ожидающие процессы так, чтобы они опять вступили в конкуренцию за обладание ресурсами контейнера. ГалаКонтейнеры подобны процессам-серверам, но более эффективны и потребляют меньше системных ресурсов, чем ГалаПроцессы. Контейнеры можно использовать вместо процессов в тех случаях, когда назначение контейнера только в том, чтобы предоставить доступ нескольким процессам к одному ресурсу. В отличие от процессов, контейнеры создаются, как правило, не динамически, а статически.

ГалаСообщение - позволяет взаимодействовать ГалаПроцессу с основным VCL-потоком с помощью посылки окну (форме) синхронного Windows-сообщения методом Send. Для ответа на такое сообщение в VCL-форме определяется соответствующий метод. Данные передаются в сообщении с помощью бестипового указателя, что позволяет выполнить двунаправленный обмен данными любого типа. На время обработки сообщения, вызвавший процесс приостанавливается.

ГалаКанал позволяет ГалаПроцессам взаимодействовать друг с другом и с ГалаКонтейнерами. ГалаКаналами владеют либо ГалаПроцессы, либо ГалаКонтейнеры. Каждый канал может получать сообщения одновременно от нескольких процессов. Объект-ГалаКанал может быть типа TGalaProcessChannel или TGalaContainerChannel в зависимости от того, кто будет создавать и владеть этим каналом. Создаются объекты каналов явно в конструкторах ГалаПроцесса или ГалаКонтейнера с помощью функции CreateChannel, а уничтожаются автоматически при уничтожении владельцев. Действия над данными, реализуемые при передаче сообщения в канал, описываются во входной (серверной) процедуре - она вызывается каналом в режиме взаимоисключения. Охраняющая функция определяет условие, при котором входная процедура может быть вызвана. Входная процедура соответствует входу entry оператора accept, а охраняющая функция - оператору when. Для взаимодействия с ГалаПроцессами в канале определен метод Send, принимающий указатели на посылающий процесс и на посылаемые-принимаемые данные. Использование бестипового указателя позволяет передавать по каналу данные любого типа в обоих направлениях.

ГалаСигнал. Кроме взаимодействий с другими процессами, ГалаПроцессы могут ожидать сигналы. Сигналы несут в себе только факт наступления некоторых событий, как связанных с другими процессами, так и с внешними событиями, происходящими в операционной системе. ГалаСигналы представляются собой набор таких средств, инкапсулирующих примитивы, непосредственно предоставляемые операционной системой - события, мьютексы, семафоры и файловые уведомления. Использование их необязательно, но иногда более удобно, особенно в тех случаях, когда не требуется обмен

данными, а важен сам факт наступления некоторого события. Все ГалаСигналы порождаются от базового класса TGalaSignal и имеют метод Wait. Метод Wait позволяет ограничить время ожидания и отслеживает факт получения ожидающим процессом директивы завершения, что дает возможность выполнить корректное принудительное завершение процесса, находящегося в состоянии ожидания.

Теперь более подробно опишем все сказанное выше. В этом разделе рассмотрим все существенные для использования свойства и методы ГалаТеатра.

ГалаТеатр.

```
TGalaTheater = class
public
  constructor Create;
  destructor  Destroy; override;
  function    GetNewGroup: Integer;
  procedure   ResumeGroup(aGroup: Integer);
  procedure   SuspendGroup(aGroup: Integer);
  function    TerminateAllProcesses: Boolean;
  function    TerminateGroup(aGroup: Integer): Boolean;
  function    TryToDestroyAllProcesses: Boolean;
  function    TryToDestroyGroup(aGroup: Integer): Boolean;
  procedure   DestroyAllProcesses;
  procedure   DestroyGroup (aGroup: Integer);
  property    Log(const S: String);
  property    PriorityClass: Integer read  GalaGetPriorityClass
                                     write GalaSetPriorityClass;
  property    NotificationWindow: HWND
                read  FNotificationWindow
                write FNotificationWindow;
end;

var
  GalaTheater: TGalaTheater;
```

Объект GalaTheater создается конструктором Create и уничтожается деструктором Destroy. Аналогично объекту Application в программе может быть только один объект GalaTheater. Удобное место для создания и уничтожения ГалаТеатра - методы FormCreate и FormDestroy (или FormClose) главной формы приложения. Процессы в ГалаТеатре существуют группами. Группа характеризует связанность процессов, - процессы взаимодействуют между собой только внутри группы. Исключение составляют серверные (системные) процессы, которые объединяются в одну группу и предоставляют сервисы процессам остальных групп. Функция GetNewGroup позволяет получить уникальный номер группы в том случае, если в вашей программе будет несколько групп, создающихся и уничтожающихся динамически. Если в программе одна или несколько статических групп, то идентификаторы (номера) групп можно задать константами (начиная от 1). Перед закрытием ГалаТеатра нужно завершить все (еще работающие) процессы. Закрыть ГалаТеатр можно только из VCL-потока: функция TerminateAllProcesses вызывает функцию завершения (Terminate) для всех еще активных процессов. Функция TryToDestroyAllProcesses сначала вызывает TerminateAllProcesses и только если все процессы уже завершены, уничтожает их. Если в процессе уничтожения обнаруживается новый процесс, то ему посылается уведомление о завершении, уничтожение процессов прекращается и функция возвращает False. Такое явление может случиться в том случае, если ГалаПроцесс создал новый процесс и только после этого получил уведомление о завершении. Функция возвращает True, если все процессы ГалаТеатра успешно уничтожены. Аналогично работают функции TerminateGroup и TryToDestroyGroup, но в отличие от предыдущих функций они закрывают и уничтожают только процессы

указанной группы и могут быть вызваны не только из VCL-потока, но и из процесса другой группы. Процедура SuspendGroup позволяет одновременно приостановить, а ResumeGroup - активизировать все процессы группы. Процедуры DestroyAllProcesses и DestroyGroup уничтожают все процессы ГалаТеатра или процессы заданной группы и дожидаются полного завершения операции уничтожения. Свойство PriorityClass позволяет изменить класс приоритета всех процессов ГалаТеатра. Свойство NotificationWindow задает окно, в которое будут посылаться уведомления о создании, уничтожении и трассировке процесса. Эта информация полезна при отладке. Если окно не задано, то уведомляющие сообщения не посылаются. Свойство Log выводит отладочную строку в файл протокола (обеспечивая взаимное исключение процессов).

ГалаПроцесс.

```
TGalaProcess = class
protected
  FSuspendedOnStart: Boolean;
  FStackSize:        Integer;

  function  CreateChannel(aEntry: TGalaEntry;
            aGuard: TGalaGuard = nil): TGalaProcessChannel;
  procedure Execute; virtual; abstract;
  procedure OnNormalTermination; virtual;
  procedure OnPrematureTermination; virtual;
  procedure OnUnhandledException(E: Exception); virtual;
  procedure Trace(const S: String);
  procedure Log(const S: String);
  procedure Pause(aTime: Cardinal);
  procedure Send(aMessageId: Cardinal; aData: Pointer = nil;
            aForm: TForm = nil;  aTimeout: Cardinal = INFINITE);
  procedure Accept(aChannel: TGalaProcessChannel;
            aTimeout: Cardinal = INFINITE);
  procedure AlternativeAccept(aChannels: array of TGalaProcessChannel;
            aTimeout:  Cardinal = INFINITE);
  procedure Wait(aSignal: TGalaSignal;
            aTimeout: Cardinal = INFINITE);
  function  AlternativeWait(aSignals: array of TGalaSignal;
            aTimeout: Cardinal = INFINITE): Integer;
  procedure WaitCompletion(const aProcesses: array of TGalaProcess;
            aTimeout: Cardinal = INFINITE);
public
  ProcessName:     String;
  FreeOnTerminate: Boolean;
  constructor Create(aGroup: Integer; aParentForm: TForm = nil);
  destructor  Destroy; override;
  procedure   Terminate;
  procedure   Suspend;
  procedure   Resume;

  property    Handle:     THandle read  FHandle;
  property    Suspended:  Boolean read  FSuspended;
  property    Terminated: Boolean read  FTerminated;
  property    Finished:   Boolean read  FFinished;
  property    Priority:   Integer read  GetPriority
                                  write SetPriority;
end;
```

ГалаПроцесс создается явным вызовом его конструктора Create, которому передается номер группы и ссылка на форму, с которой процесс будет наиболее плотно взаимодействовать. Как правило, это форма, из которой вызван конструктор процесса. Если в программе только одна группа процессов, то номер группы можно задавать константой 1. Ошибка при создании процесса будет возбуждать исключительную

ситуацию EGalaObjectCreationFail. Собственно тело процесса программируется в методе Execute, который обязательно должен быть переопределен в наследуемом классе. Если процесс завершается нормально, то он освобождает основные системные ресурсы, но объект процесса продолжает существовать и уничтожается только ГалаТеатром - и только вместе со своей группой. Попытка взаимодействия с завершенным процессом будет возбуждать исключение EGalaProcessWasTerminated. Для принудительного завершения процесса нужно вызвать процедуру Terminate. Действие ее состоит в следующем: она устанавливает флажок Terminated и активизирует внутренний сигнал процесса FTerminationEvent, который выводит процесс из состояния ожидания (если процесс в нем находится). В ходе выполнения метода Execute процесс должен периодически контролировать свойство Terminated и выполнять завершающие действия, например:

```
while not Terminated do begin
  // действия процесса
end;
// завершающие действия
```

Принудительное завершение - это очень важное событие и оно учитывается всеми остальными объектами ГалаТеатра. Синхронизация процессов всегда связана с ожиданием, при котором процесс переходит в спящее состояние - ожидание некоторого события. И если это событие уже не произойдет (например, завершился другой процесс, генерирующий это событие), то ожидающий процесс никогда не проснется, что приведет к зависанию программы. Поэтому уход процесса в спящее ожидание всегда охраняется таймаутом (необязательным) и реакцией на принудительное завершение (обязательной). Если в то время, когда процесс находится в спящем состоянии, он получает директиву принудительного завершения, то процесс активизируется и в нем возбуждается исключительная ситуация EGalaPrematureTermination. Если процесс явно ее не перехватывает, то нормальная работа процесса завершается и вызывается виртуальное событие OnPrematureTermination, которое может быть переписано в наследуемом классе. При нормальном завершении вызывается виртуальное событие OnNormalTermination. Конечно, процесс может выполнить нормальные завершающие действия и в методе Execute, но выделение их в отдельное событие более логично. Обычно принудительное завершение ГалаПроцесса находится в компетенции ГалаТеатра (и VCL-потока), но этим правом могут воспользоваться также ГалаПроцессы, если у них есть на это веские основания.

При возникновении необработанной явным образом исключительной ситуации, управление передается обработчику события OnUnhandledException, который выводит информационное сообщение и завершает выполнение ГалаПроцесса. Вы его можете переписать в наследуемом классе для реализации собственной индивидуальной реакции на необработанные исключения.

В конструкторе процесса можно задать:
его имя (ProcessName), которое используется при отладочных уведомлениях (по умолчанию - это имя класса)
максимальный размер стека процесса FStackSize (по умолчанию - 1 мб). При создании процесса ему выделяется начальный стек в 8 кб, который растет динамически по мере необходимости признак приостановки при создании процесса FSuspendedOnStart. По умолчанию процесс после создания сразу же активизируется. Часто это умолчание недопустимо, так как процесс сразу же может начать взаимодействовать с еще не созданными процессами. Для того, чтобы корректно обработать эту ситуацию, все взаимодействующие процессы запускаются в 2 этапа: сначала создаются как неактивные (подвешенные), а потом уже активизируются. Для активизации приостановленного процесса используется процедура Resume. Для активизации всех процессов группы можно

использовать метод ГалаТеатра ResumeGroup признак самоуничтожения при завершении FFreeOnTerminate - использовать его нужно очень осторожно и устанавливать только в том случае, если процесс не взаимодействует ни с каким другим Гала-процессом. Этот признак - лазейка в общей стратегии уничтожения процессов стартовый приоритет процесса Priority
По ходу работы сам процесс может менять свой приоритет, а также процессы могут менять приоритеты друг друга.
Значения приоритетов следующие:
-3 - низший приоритет (Idle)
-2 - на 2 пункта ниже нормального
-1 - на 1 пункт ниже нормального
0 - нормальный приоритет
1 - на 1 пункт выше нормального
2 - на 2 пункта выше нормального
3 - высший приоритет (Time Critical)
Процесс может приостанавливать свою работу на заданный промежуток времени вызовом метода Pause, которому передается значение времени в миллисекундах. Если во время паузы процессу будет передана директива принудительного завершения, то пауза будет прервана и возбуждена исключительная ситуация EGalaPrematureTermination.

Для взаимодействия с основным VCL-потоком ГалаПроцесс посылает VCL-форме сообщение с помощью метода Send.
Этот метод имеет аргументы:
идентификатор сообщения (целое число) бестиповый указатель на передаваемые-принимаемые данные (Pointer). По умолчанию данные - это ссылка на сам процесс ссылка на форму, которой направляется сообщение. По умолчанию - это ParentForm время таймаута в миллисекундах (по умолчанию - бесконечность)

ГалаКаналы.

ГалаКаналы - это исполнители, реализующие взаимодействия между процессами. Каналы ГалаПроцессов и ГалаКонтейнеров внешне полностью совпадают, но имеют различную реализацию:

```
TGalaContainerChannel = class
public
  procedure Send(aSender: TGalaProcess; aData: Pointer = nil;
                 aTimeout: Cardinal = INFINITE);
  property  Entry: TGalaEntry read FEntry write FEntry;
  property  Guard: TGalaGuard read FGuard write FGuard;
end;

TGalaProcessChannel = class
public
  procedure Send(aSender: TGalaProcess; aData: Pointer = nil;
                 aTimeout: Cardinal = INFINITE);
  property  Entry: TGalaEntry read FEntry write FEntry;
  property  Guard: TGalaGuard read FGuard write FGuard;
end;
```

Создаются каналы с помощью вызова метода CreateChannel в конструкторах ГалаКонтейнера и ГалаКанала, а уничтожаются автоматически в их деструкторах. Метод CreateChannel у ГалаПроцесса точно такой же, как и у ГалаКонтейнера и имеет аргументы:

- ссылку на процедуру-обработчик (Entry), которая выполняет функциональную обработку данных (aData), передаваемых в процессе рандеву. Обработчик должен быть методом ГалаПроцесса или ГалаКонтейнера и иметь такой тип: TGalaEntry = procedure(aData: Pointer) of object;
- необязательную ссылку на охраняющую функцию (Guard), которая выполняет роль предохранителя, разрешающего или запрещающего вызов обработчика Entry - факт того, что рандеву может состояться. Функция должна быть методом ГалаПроцесса или ГалаКонтейнера и иметь тип: TGalaGuard = function: Boolean of object;

Обработчик вызывается всегда в режиме взаимного исключения и только в том случае, когда оба взаимодействующих процесса готовы к рандеву и охраняющая функция возвращает True. Отметим, что если охраняющая функция не задана, то это эквивалентно функции, которая всегда возвращает True, то есть вход всегда разрешен.

Метод Send является входом канала. Процесс, инициирующий рандеву, вызывает именно этот метод для передачи в канал сообщения. Метод Send принимает 2 аргумента: ссылку на процесс-отправитель и указатель на бестиповые данные. Если данные не заданы, то используется ссылка на процесс-оправитель. Третий, необязательный, аргумент - таймаута, время, в течение которого клиентский процесс готов ждать рандеву.

ГалаСигналы.

Эта группа примитивов инкапсулирует средства синхронизации, предоставляемые операционной системой Windows. Все объекты-сигналы порождаются от базового класса TGalaSignal:

```
TGalaSignal = class
protected
  FHandle: THandle;

  procedure AfterWaiting(p: TGalaProcess); virtual;

public
  constructor Create(aHandle: THandle);
  procedure   Wait(aProcess: TGalaProcess;
              aTimeout: Cardinal = INFINITE);
  property    Handle: THandle read FHandle;
end;
```

Метод Wait - общий для всех ГалаСигналов и позволяет ГалаПроцессам ожидать наступления события, связанного с системным объектом Handle с ограничением по времени. Отметим, что метод отслеживает факт получения ожидающим процессом директивы принудительного завершения и активизирует процесс с возбуждением исключительной ситуации EGalaPrematureTermination. При возникновении таймаута возбуждается исключение EGalaTimeout.

```
TGalaEvent = class(TGalaSignal)
public
  constructor Create(aManualReset: Boolean;
                     aInitialState: Boolean = False;
                     aName: PChar = nil);
  procedure   SetState(aState: Boolean); virtual;
  procedure   Pulse; virtual;
end;
```

Конструктор **ГалаСобытия** имеет 3 аргумента: режим сброса (ManualReset), начальное

состояние и необязательное имя. Если ManualReset = False, то событие автоматически становится несигнализирующим после вызова ожидающим процессом метода Wait. Если ManualReset = True, то событие остается сигнализирующим и его нужно явно сбрасывать вызовом SetState(False). Имя события имеет смысл только при взаимодействии нескольких Windows-программ, это относится также к ГалаМьютексам и ГалаСемафорам. Метод Pulse позволяет установить событие и потом сразу сбросить.

```
TGalaMutex = class(TGalaSignal)
public
  constructor Create(aOwned: Boolean; aName: PChar = nil);
  procedure   Release; virtual;
end;
```

ГалаМьютекс инкапсулируют понятие одиночного ресурса, которым может владеть в один момент времени только один процесс. Слово "Mutex" происходит из соединения двух слов - "Mutual" и "Exclusive", что означает взаимно-исключающее. При создании мьютекса можно сразу завладеть ресурсом, если аргумент aOwned равен True. Для захвата ресурса процесс вызывает метод Wait, для освобождения ресурса - метод Release. Возможен вложенный захват ресурса, то есть, процесс захвативший мьютекс, может захватить его еще раз, но сколько раз захвачен мьютекс, столько раз его нужно освободить.

```
TGalaSemaphore = class(TGalaSignal)
public
  constructor Create(aMaxCount: Integer;
              aInitialCount: Integer = -1;
              aName: PChar = nil);
  procedure   Release(aCount: Integer = 1); virtual;
end;
```

ГалаСемафор похож на ГалаМьютекс, но, в отличие от мьютекса, инкапсулируют понятие множественного, а не одиночного, ресурса. При создании семафора задается количество единиц разделяемого ресурса (aMaxCount). При создании можно также захватить одну или больше единиц ресурса (aInitialCount), если аргумент не задан, то все ресурсы свободны. Для захвата ресурса процесс вызывает метод Wait, для освобождения ресурса - метод Release. За один раз можно освободить одну или несколько единиц ресурса, но захватить можно только одну за один раз. Пример использования семафора - торговый прилавок с тремя продавцами. Одновременно может быть обслужено не более 3х клиентов: остальные клиенты должны ждать освобождения ресурса - продавца.

```
TGalaChangeNotification = class(TGalaSignal)
public
  constructor Create(const aDirectory: String;
                     aSubtree: Boolean;
                     aFilter: Cardinal);
  procedure   NewDir(const aDirectory: String); virtual;
end;
```

ГалаУведомление инкапсулирует Windows-уведомления о различных событиях-изменениях, происходящих в файловой системе. Конструктор имеет параметры: имя каталога, для которого производится уведомление о событиях, флаг, указывающий, нужно ли отслеживать события в подкаталогах и фильтр событий, указывающий, какие конкретно события нужно отслеживать. Для ожидания события процесс просто вызывает метод Wait. В процессе отслеживания можно изменить имя каталога методом NewDir.

```
TGalaDeque = class(TGalaSignal)
```

```
public
  constructor Create(aDequeSize: Integer = 0);
  function    Count: Integer; virtual;
  procedure   PutLast(aObject: TObject); virtual;
  function    GetLast: TObject; virtual;
  procedure   PutFirst(aObject: TObject); virtual;
  function    GetFirst: TObject; virtual;
  function    PeekFirst: TObject; virtual;
  function    PeekLast: TObject; virtual;
end;
```

ГалаДек - это универсальная очередь, которая может быть использована для передачи асинхронных сообщений между процессами, между контейнерами, между процессом и VCL-потоком, процессом и контейнером. Сообщение - это объект любого типа, порождаемого от TObject. Передача сообщения не связана с синхронизацией и ожиданием. Очередь может быть ограниченной или неограниченной длины. Попытка записи в заполненную очередь будет вызывать исключение EGalaOverflow. Когда очередь непуста, то состояние ГалаДека - сигнализирующее, если в очереди нет ни одного сообщения, то состояние ГалаДека - несигнализирующее. Процессы могут ждать поступления сообщений в очередь, используя обычные для всех ГалаСигналов методы. Сообщения могут посылаться либо в начало, либо в конец очереди и извлекаться также либо из начала, либо из конца очереди. Ответственным за создание объекта-сообщения является процесс, посылающий сообщение, а ответственным за использование и уничтожение объекта-сообщения - принимающий процесс.

```
TGalaMessage = class(TGalaSignal)
public
  constructor Create;
  destructor  Destroy; override;
  procedure   Release; virtual;
  procedure   Send(aWnd: THandle; aMessage: Integer); virtual;
  procedure   Reply; virtual;
end;
```

Гала-сообщение - это Гала-сигнал, который можно использовать для передачи данных и команд и последующего ожидания ответа. Он удобен для взаимодействия Гала-процесса с другими объектами, не являющимися Гала-процессами, например VCL-потоком или очередями. Отличительная особенность Гала-сообщения состоит в том, что этот объект осуществляет подсчет ссылок и уничтожается самостоятельно (подобно COM-объектам). Использование Гала-сообщения позволяет послать ответное сообщение даже в том случае, если ожидающий процесс уже уничтожился. Метод Send посылает сообщение окну и увеличивает счетчик ссылок, метод Reply уведомляет о получении Гала-сообщения установкой его в активное состояние (которое сбрасывается при успешном ожидании) и уменьшает счетчик ссылок. Начальное состояние счетчика ссылок равно 1. Принудительное уменьшение счетчика ссылок выполняет метод Release - его вызов означает, что объект больше не нужен. Если счетчик ссылок становится равным нулю, то объект самоуничтожается.

Кроме указанных примитивов в Gala-библиотеке определен служебный класс ГалаЗамок (TGalaLatch), инкапсулирующий понятие критической секции. Он используется внутри многих объектов библиотеки и может быть использован как самостоятельный в тех случаях, когда требуется очень высокая степень эффективности и за это можно заплатить трудоемкостью и сложностью при программировании. Объект имеет два метода: Lock и Unlock. Процесс может заблокировать (закрыть) замок методом Lock - в этом случае все остальные процессы, попытавшиеся вызвать метод Lock будут приостановлены до того момента, когда захвативший процесс не вызовет метод Unlock. При использовании замка

нужно быть очень осторожным, так как закрытый замок не дает ожидающим процессам никакого шанса на продолжение. Использование замка практически всегда нужно выполнять в блоке try-finally, например:

```
var List: TGalaLatch;
 . . .
 List.Lock;
 try
   // действия со списком
 finally
   List.Unlock;
 end;
```

Некоторые объекты VCL имеют встроенную поддержку разделяемого доступа, например: TThreadList, TCanvas.

Разработка примеров использования ГалаБиблиотеки

Для всех примеров существует одна-единственная программа. Процессы каждого примера образуют группу, причем можно одновременно запускать не только несколько примеров, но также несколько экземпляров одного и того же примера. Главная форма программы имеет 3 элемента - выпадающий список, содержащий имена примеров, кнопку "Старт" и таблицу состояния ГалаПроцессов, в которой отражаются все активные процессы и их отладочные сообщения.

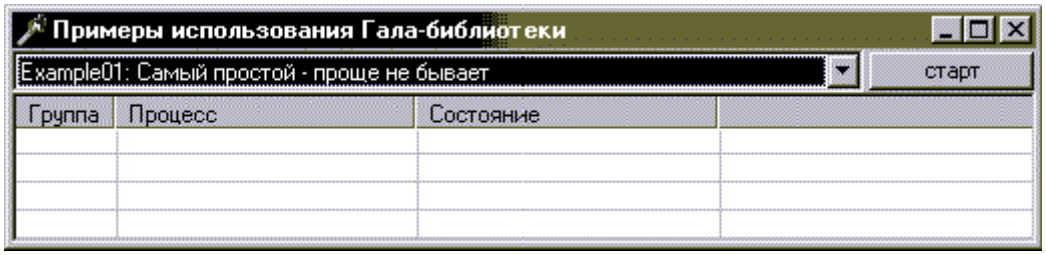

Рисунок 1 – Вид окна программы запуска примеров.

Главная форма регистрируется вызовом GalaTheater.NotificationWindow

и имеет три метода - обработчика трех стандартных уведомляющих ГалаСообщений:

```
procedure OnProcessStart(var Mes: TMessage);
            message GM_PROCESS_ START;
procedure OnProcessTrace(var Mes: TMessage);
            message GM_PROCESS_TRACE;
procedure OnProcessTermination(var Mes: TMessage);
            message GM_PROCESS_TERMINATE;
```

Пример 01.
Название: "Example01: Самый простой - проще не бывает "
Действующие лица:
Циклический процесс "Счетчик"

Декорации:

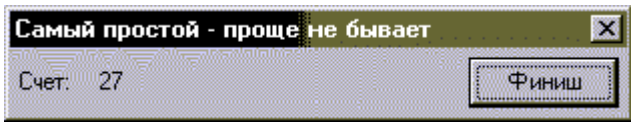

Рисунок 2 – Вид окна примера Example01.

Сценарий: Процесс TCounter каждые 100 миллисекунд передает форме сообщение, содержащее монотонно увеличивающееся число. А форма-декорация его отображает в TLabel. Спектакль начинается с нажатия кнопки "Старт" и заканчивается нажатием одноименной кнопки "Финиш".

Цель примера - показать взаимодействие ГалаПроцессов с VCL-компонентами. Собственно процесс выглядит следующим образом:

```
procedure TProcess01.Execute;
var
  Counter: Integer;

begin
  Counter := 0;
  while not Terminated do begin
    Inc(Counter);
    Send(GM_DRAW_01, @Counter);
    Pause(100);
  end;
end;
```

В цикле (вплоть до завершения процесса по кнопке "Финиш") значение счетчика увеличивается на 1, вызывается метод Send, передающий форме сообщение с номером GM_DRAW_01, а в качестве параметра - указатель на переменную-счетчик. После этого процесс приостанавливается на 100 миллисекунд. Для обработки сообщения в форме имеется метод:

```
procedure TForm01.OnDraw(var Mes: TMessage); message GM_DRAW_01;
begin
  LabelCounter.Caption := IntToStr(PInteger(Mes.LParam)^);
end;
```

Метод OnDraw извлекает значение счетчика из Windows-сообщения, преобразует в строку и отображает.

Для создания процесса сначала запрашивается динамический номер группы (так как можно запускать несколько экземпляров примера параллельно), а потом уже вызывается конструктор процесса. Для завершения процесса просто вызывается процедура уничтожения группы с ожиданием:

```
procedure TForm01.ButtonStartStopClick(Sender: TObject);
begin
  if Group = 0 then begin
    Group := GalaTheater.GetNewGroup;
    TProcess01.Create(Self, Group);
  end
  else begin
    GalaTheater.DestroyGroup(Group);
    Group := 0;
  end;
end;
```

Пример 02.
Название: "Example02: Размножающиеся процессы "

Действующие лица:
"Шарики" - динамически создаваемые и уничтожаемые процессы.

Декорации:

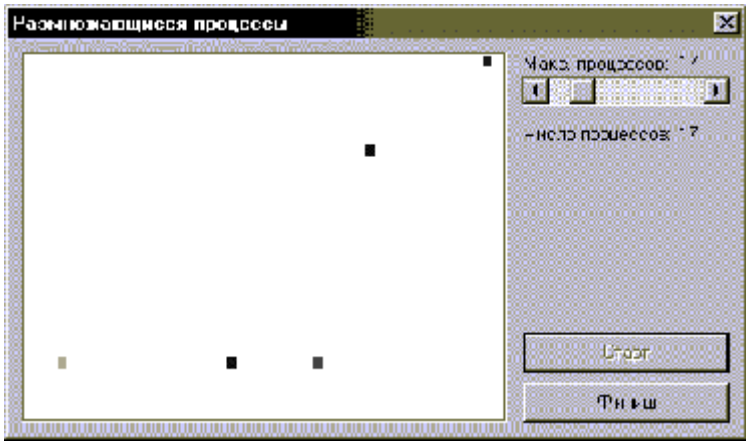

Рисунок 3 – Вид окна примера Example02.

Сценарий: Процессы ведут себя как движущиеся шарики (квадратики), отражающиеся от границ поля. Каждый шарик имеет свой цвет, скорость, время жизни и период размножения (случайные величины). Каждый шарик пытается создать себе подобного, но может размножаться только в том случае, если находится на некотором расстоянии от границы поля и общее число шариков не превышает заданного максимума (который можно динамически изменять).

Цель примера - показать следующие возможности библиотеки:

- динамическое создание и уничтожение процессов
- защита от перегрузки системы при большом количестве процессов
- реакция на нормальное и принудительное завершение

Жизненный цикл каждого процесса таков (в тексте опускается несущественный для параллельного программирования текст, а опущенные параметры и части кода записываются на русском языке):

```
procedure TProcess02.Execute;
begin
  // шарику даем меньший приоритет, чем VCL-процессу
  Priority := -1;
  // Посылка форме сообщения о создании нового процесса, поскольку
  // именно форма отслеживает общее число процессов
  Send(GM_PROCESS_START);
  while (not Terminated) and не_истекло_время_жизни do begin
    расчет_и_проверка_новых_координат;
```

```
    if настало_время_размножения then begin
      расчет_координат_потомка;
      if создание_потомка_возможно then begin
        try
          // Создание потомка
          TProcess02.Create(Group, ParentForm,
            координаты, направление);
        except
          // Недостаточно системных ресурсов
          on EGalaProcessCreationFail do
            Beep;
        end;
      end;
    end;
    // Отображение шарика на форме
    try
      // Время ожидания рандеву с VCL-потоком ограничено. Если
      // форма перегружена запросами и не может нарисовать шарик
      // за 200 мс, то шарик умирает (от тоски, потому что летать
      // не может). В качестве параметров процесс передает форме
      // ссылку на самого себя
      Send(GM_DRAW_02, Self, ParentForm, 200);
      // Перемещение шарика на новую позицию
      PrevPoint := NextPoint;
      // Пауза для того, чтобы шарики бегали не так быстро
      Pause(Speed);
    except
      // Рандеву с VCL-потоком не состоялось,
      // процесс завершает сам себя (увы, в жизни всяко бывает)
      on EGalaTimeout do
        Terminate;
    end;
  end;
end;
```

Действия, совершаемые при нормальном и аварийном завершении в данном случае одинаковы - шарик стирает себя с экрана и уменьшает общее число шариков:

```
procedure TProcess02.OnPrematureTermination;
begin
  OnNormalTermination;
end;

procedure TProcess02.OnNormalTermination;
begin
  Color := clWhite;
  // Стираем себя
  Send(GM_DRAW_02);
  // Посылаем форме уведомление о своем завершении
  Send(GM_PROCESS_TERMINATE);
end;
```

Форма отвечает на сообщения процессов тремя процедурами - обработчиками сообщений: GM_PROCESS_START, GM_PROCESS_TERMINATE и GM_DRAW_02:

```
procedure TForm02.OnStart(var Mes: TMessage);
begin
```

```
  Inc(Count);
  LabelProcessCount.Caption := IntToStr(Count);
end;

procedure TForm02.OnTermination(var Mes: TMessage);
begin
  Dec(Count);
  LabelProcessCount.Caption := IntToStr(Count);
end;
procedure TForm02.OnDraw(var Mes: TMessage);
begin
  with TProcess02(Mes.LParam) do begin
    отрисовка_шарика;
  end;
end;
```

Если внимательно наблюдать за поведением шариков, особенно при их большом количестве, то можно заметить следующее - иногда количество шариков превышает заданный максимум. Почему это происходит? Если посмотреть на то, как процесс создается, можно заметить, что сначала процесс-родитель запрашивает у формы - можно ли создавать потомка? Затем процесс создает потомка и уже сам потомок дает форме сообщение о создании, которое увеличивает действительное число процессов. Т. е. между проверкой возможности создания и реальным созданием проходит время, в которое операционная система может перейти к выполнению другого процесса-шарика, который также готов к размножению. То есть, подтверждение на создание потомка могут получить несколько процессов-родителей прежде, чем потомки действительно будут созданы. Корректно эту проблему в данном примере можно было бы решить с помощью счетного семафора TGalaSemaphore или с помощью другого параллельного процесса. Такой процесс регистрировал бы запрос на создание и уменьшал бы значение счетчика - но это потребует также обработки ситуации, в которой процесс-родитель получил подтверждение на создание потомка, но не смог его создать.

Есть еще одна проблема, которую можно заметить при внимательном наблюдении за большим количеством процессов. Если нажать на кнопку "Финиш", которая завершает всю группу шариков, то иногда все шарики удаляются, и вдруг, неожиданно, появляется один или несколько новых шариков, которые уничтожаются повторным нажатием на кнопку "Финиш". Причина "ошибки" та же, что и предыдущей - вытесняющая многозадачность. Процесс-родитель может обнаружить, что он должен принудительно завершиться уже после того, как создал потомка, а потомок не успеет получить сообщение о принудительном завершении, так как он только начал создаваться и еще не включен в список активных процессов, которым посылается сигнал принудительного завершения. Решением этой проблемы было бы, например, следующее - при создании процесс запрашивает у формы признак существования данной (своей) группы и, если группа уже не существует (номер группы равен 0), то процесс сразу завершается.

Заключение

Библиотека классов параллельного программирования Gala для Delphi применима для решения асинхронного распараллеливания задач, так как программы использующие эту библиотеку функционируют в пределах однопроцессорной ЭВМ. Такое функционирование реализуется благодаря тому, что большую часть времени компоненты

задачи находятся в состоянии ожидания определенных событий. Дальнейшим усовершенствованием библиотеки Gala является приведение ее к виду для решения проблемы синхронного распараллеливания, то есть чтобы каждый процесс задачи или некоторая их часть функционировала на отдельной ЭВМ. В дальнейшем это может быть реализовано на базе некой компьютерной сети, обьединяющей несколько ЭВМ. Обмен данными будет возможен посредством передачи их по сети используя такой сетевой протокол как TCP/IP. На данный момент IP-сети широко распространены и показали свою большую эффективность и работоспособность как и протокол TCP/IP.

Также важным этапом в усовершенствовании библиотеки Gala для Delphi является приведение ее к стандарту параллельного программирования с передачей сообщений MPI (Message Passing Interface). Стандарт MPI является основным средством программирования таких современных производительных мультикомпьютеров, как Silicon Graphics Origin 2000, Cray T3D, IBM SP2 и многих других. MPI предоставляет программисту единый механизм взаимодействия ветвей внутри параллельного приложения независимо от машинной архитектуры (однопроцессорные / многопроцессорные с общей/раздельной памятью), взаимного расположения ветвей (на одном процессоре / на разных) и API операционной системы.

На данный момент для персональных компьютеров библиотека MPI разработана только для языков программирования C и Фортран. Данный стандарт еще не реализовывался для языка программирования Delphi, поэтому дальнейшие разработки и усовершенствования Gala в сторону стандарта MPI являются перспективными и актуальными в наши дни.

Функционирование программ, использующих библиотеку далее разработанную библиотеку предполагается в рамках операционных система семейства Windows. Программы будут иметь возможность работать как пределах одной ЭВМ и одной операционной системы, так и нескольких ЭВМ, обьединенных между собой при по IP-сетью. Как уже говорилось выше, обмен данными между процессами, функционирующих на различных компьютерах сети будет обеспечено средствами протокола TCP/IP. Современные операционные системы семейства Windows имеют в своем составе все сетевые компоненты, необходимые для реализации этой идеи, поэтому у программиста не займет много времени для настройки функционирования его программы, использующей сетевой обмен данными между процессами программы. Этим будет обеспечен синхронный механизм работы процессов программы, использующей данную MPI для Delphi.

Литература

1. С. Гурин. "Параллельное объектно-ориентированное программирование на C++". Монитор 6, 1995.
2. С. Бобровский. "Delphi 5: учебный курс". Спб: Питер, 2001
3. В. Ш. Кауфман. "Языки программирования. Концепции и принципы". М. Радио и связь. 1993
4. MPI: Стандарт интерфейса передачи сообщений.
5. Ч. Хоар. "Взаимодействующие последовательные процессы". М. Мир 1989
6. С. Янг. "Алгоритмические языки реального времени. Конструирование и разработка". М. Мир 1985