Multithreaded Application Tutorial/ru

From Lazarus wiki

Jump to navigationJump to search

| Deutsch (de) | English (en) | español (es) | français (fr) | 日本語 (ja) | polski (pl) | português (pt) | **русский (ru)** | slovenčina (sk) | 中文（中国大陆） (zh_CN) |

Обзор

Эта страница объясняет как писать и отлаживать многопоточные(multi-threaded) приложения средствами Free Pascal и Lazarus. Многопоточное приложение — это приложение, которое создаёт два или более потока исполнения, работающих одновременно. Если Вы новичок в многопоточности, пожалуйста, прочитайте раздел "Нужна ли на самом деле многопоточность?" чтобы определить, что это действительно необходимо.

Один из потоков называется главным, основным (Main Thread). Главный поток создаётся операционной системой после запуска приложения. Главный поток **должен быть** единственным потоком, который обновляет компоненты, взаимодействующие с пользователем: иначе приложение может зависнуть.

Основная идея состоит в том, что приложение может производить некоторую обработку в фоновом режиме во втором потоке, а пользователь может продолжать работу с помощью основного потока.

Другое использование потоков состоит в лучшей отзывчивости (responding) приложения. Если вы создали приложение, где пользователь нажимает на кнопку, обрабатывающую большой объём работы, во время обработки экран перестаёт отвечать на запросы и пользователь думает, что приложение не отвечает. Создаётся плохое и вводящее в заблуждение впечатление. Если это задание выполняется во втором потоке, приложение сохраняет работоспособную форму (почти), как будто оно находится в состоянии простоя. В этом смысле это хорошая идея перед запуском потока выключить кнопки на форме, позволяющие пользователю запустить более чем один поток для работы.

Другим способом использования многопоточности может быть серверное приложение, которое одновременно отвечает многим клиентам.

Нужна ли на самом деле многопоточность?

Если Вы новичок в многопоточности и хотите только чтобы Ваше приложение отвечало на действия пользователя, пока оно выполняет задачи умеренной длительности, тогда мультипоточность, возможно, это избыточное средство.

Многопоточные приложения всегда более сложные в отладке и зачастую они имеют гораздо более сложную структуру; во многих случаях Вам не нужна многопоточность. Одного потока вполне достаточно. Если Вы можете разделить трудоёмкую задачу на несколько небольших частей, тогда вместо многопоточности Вы должны использовать **Application.ProcessMessages**. Этот метод позволяет LCL обрабатывать все ожидающие сообщений и возвращения (messages and returns). Центральной идеей является вызов **Application.ProcessMessages** в одинаковые промежутки в ходе выполнения длительной задачи чтобы определить, нажал ли пользователь куда-нибудь или индикатор прогресса должен быть перерисован, и так далее.

К примеру, чтение большого файла и его обработки См. *examples/multithreading/singlethreadingexample1.lpi*.

Многопоточность необходима только для:

- блокировки handles, таких как сетевое соединение
- использования нескольких процессоров одновременно (multiple processors simultaneously, SMP)
- вызовы алгоритмов и библиотек, которые должны вызываться через API, т.к. они не могут быть разделены на мелкие части.

Модули, необходимые для многопоточного приложения

Вам не нужен какой-либо специальный модуль для работы с Windows. Однако с Linux, macOS и FreeBSD вам нужен модуль `cthreads`, и он *должен* быть первым используемым модулем проекта (в исходнике программы, обычно файл .lpr)! В случаях, когда потребуется разместить одновременно несколько "важных" модулей, таких как `cthreads`, `cmem` и `cwstrings`, ввиду особенностей их работы оптимальным порядком размещения будет `cmem`, `cthreads`, и лишь затем `cwstrings`.

Итак, код вашего приложения Lazarus должен выглядеть так:

```
program MyMultiThreadedProgram;
{$mode objfpc}{$H+}
uses
{$ifdef unix}
  cthreads,
  cmem, // диспетчер памяти C в некоторых системах намного быстрее для
многопоточности
{$endif}
  Interfaces, // это включает в себя набор виджетов LCL
  Forms
  { Вы можете добавить другие модули здесь },
```

Если вы забудете об этом и используете TThread (не добавив для nix-ов cthreads), вы получите эту ошибку при запуске:

```
 This binary has no thread support compiled in.
 Recompile the application with a thread-driver in the program uses clause
before other units using thread.
 { В этом двоичном файле нет поддержки потоков.}
 {Перекомпилируйте приложение с драйвером потока, указанном в разделе uses
программы перед другими модулями, использующими поток.}
```

Примечание: Если вы получаете ошибку компоновщика о том, что "mcount" не найден, значит, вы используете какой-то модуль, содержащий многопоточный код, и вам нужно добавить модуль `cthreads` или использовать интеллектуальное связывание.

Примечание: Если вы получаете ошибку: "Project raised exception class 'RunError(232)'" в процедуре SYSTEM_NOTHREADERROR, то ваш код требует многопоточности и вам необходимо добавить модуль cthreads.

Простейший пример FPC

Приведенный ниже код дает очень простой пример. Проверено с помощью FPC 3.0.4 на Win7.

```
Program ThreadTest;
{тестирование возможности многопоточности}
{
      OUTPUT
thread 1 started
thread 1 thri 0 Len(S)= 1
thread 1 thri 1 Len(S)= 2
thread 1 thri 2 Len(S)= 3
thread 1 thri 3 Len(S)= 4
thread 1 thri 4 Len(S)= 5
thread 1 thri 5 Len(S)= 6
thread 1 thri 6 Len(S)= 7
thread 1 thri 7 Len(S)= 8
thread 1 thri 8 Len(S)= 9
thread 1 thri 9 Len(S)= 10
thread 1 thri 10 Len(S)= 11
thread 1 thri 11 Len(S)= 12
thread 1 thri 12 Len(S)= 13
thread 1 thri 13 Len(S)= 14
thread 1 thri 14 Len(S)= 15
thread 2 started
thread 3 started
thread 1 thri 15 Len(S)= 16
```

```
thread 2 thri 0 Len(S)= 1
thread 3 thri 0 Len(S)= 1
thread 1 thri 16 Len(S)= 17
...
...
thread 5 thri 997 Len(S)= 998
thread 5 thri 998 Len(S)= 999
thread 5 thri 999 Len(S)= 1000
thread 5 finished
thread 10 thri 828 Len(S)= 829
thread 9 thri 675 Len(S)= 676
thread 4 thri 656 Len(S)= 657
thread 10 thri 829 Len(S)= 830
thread 9 thri 676 Len(S)= 677
thread 9 thri 677 Len(S)= 678
thread 10 thri 830 Len(S)= 831
thread 10 thri 831 Len(S)= 832
thread 10 thri 832 Len(S)= 833
thread 10 thri 833 Len(S)= 834
thread 10 thri 834 Len(S)= 835
thread 10 thri 835 Len(S)= 836
thread 10 thri 836 Len(S)= 837
thread 10 thri 837 Len(S)= 838
thread 10 thri 838 Len(S)= 839
thread 10 thri 839 Len(S)= 840
thread 9 thri 678 Len(S)= 679
...
...
thread 4 thri 994 Len(S)= 995
thread 4 thri 995 Len(S)= 996
thread 4 thri 996 Len(S)= 997
thread 4 thri 997 Len(S)= 998
thread 4 thri 998 Len(S)= 999
thread 4 thri 999 Len(S)= 1000
thread 4 finished
10

}

uses
  {$ifdef unix}cthreads, {$endif} sysutils;

const
  threadcount = 10;
```

```
  stringlen = 1000;

var
   finished : longint;

threadvar
   thri : ptrint;

function f(p : pointer) : ptrint;
var
  s : ansistring;
begin
  Writeln('thread ',longint(p),' started');
  thri:=0;
  while (thri<stringlen) do begin
    s:=s+'1'; { create a delay }
    writeln('thread ',longint(p),' thri ',thri,' Len(S)= ',length(s));
        inc(thri);
  end;
  Writeln('thread ',longint(p),' finished');
  InterLockedIncrement(finished);
  f:=0;
end;


var
   i : longint;

Begin
   finished:=0;
   for i:=1 to threadcount do
     BeginThread(@f,pointer(i));
   while finished<threadcount do ;
   Writeln(finished);
End.
```

Класс TThread

Самый простой путь для создания многопоточного приложения — использование **класса TThread**. Этот класс позволяет достаточно просто создать дополнительный поток (наряду с основным). Обычно от вас требуется переопределить только 2 метода: **конструктор Create**, и **метод Execute**.

В **конструкторе** Вы подготавливаете поток для запуска. Вы устанавливаете начальные значения переменных и свойств, которые Вам нужны. Подлинный конструктор TThread требует параметр, называющийся **Suspended** (приостановить). Как и следует ожидать, установка Suspended = True

будет препятствовать автоматическому запуску потоков после создания. Если Suspended = False, поток запускается сразу после создания. Если поток создан приостановленным (Suspended = True), тогда он запускается только после вызова **метода Resume**.

В FPC version 2.0.1 и более поздних, TThread.Create также имеет неявный параметр, отвечающий за размер стека. Теперь, если это необходимо, Вы можете изменить стандартный размер стека каждого потока, который Вы создали. Глубокая рекурсия в процедурах является хорошим примером. Если Вы не изменяете размер стека, используется стандартный размер стека операционной системы.

В переопределённом **методе Execute** Вы будете писать код, который запускается в потоке.

Класс TThread имеет одно важное свойство: **Terminated** : Boolean;

Если поток имеет цикл (loop) — и это нормально, — выход из цикла должен осуществиться, когда **Terminated = True** (по умолчанию – False). В каждом проходе значение свойство Terminated должно проверяться, и если оно True, цикл должен завершиться так быстро, как это возможно, после необходимой очистки. Имейте ввиду, что метод **Terminate** по умолчанию ничего не делает: в метод **Execute** нужно явно организовать завершение задания.

Как мы объяснили ранее, поток не должен взаимодействовать с видимыми компонентами. Обновления для видимой компоненты должны быть сделаны в контексте главного, основного потока. Чтобы это сделать, у TThread существует метод, называющийся **Synchronize**. Synchronize требует метод в качестве аргумента. Когда вы вызываете метод через **Synchronize(@MyMethod)**, запущенный поток приостанавливается, код метода **MyMethod** запускается в главном потоке, а затем выполнение потока будет продолжено. Точная обработка Synchronize зависит от платформы, но упрощённо он делает следующее: он посылает сообщение в главную очередь сообщений и приостанавливает своё выполнение(goes to sleep). В конечном итоге главный поток обрабатывает сообщение и вызывает **MyMethod**. Таким образом, **MyMethod** вызывается без контекста, то есть не во время события **OnMouseDown** или в течение события перерисовки (paint event), но после этого. После того, как главный поток запускает **MyMethod**, он будит спящий поток и обрабатывает следующее сообщение из главной очереди сообщений, а поток продолжает работать.

Существует другое важное свойство TThread: **FreeOnTerminate**. Если оно True, тогда объект потока автоматически освобождается, когда выполнение потока (метод **Execute**) останавливается. В противном случае, приложению будет необходимо освободить его вручную.

Следующий пример может быть найден в каталоге *examples/multithreading/*:

```
Type
  TMyThread = class(TThread)
  private
    fStatusText : string;
    procedure ShowStatus;
  protected
    procedure Execute; override;
```

```
    public
      Constructor Create(CreateSuspended : boolean);
    end;

  constructor TMyThread.Create(CreateSuspended : boolean);
  begin
    FreeOnTerminate := True;
    inherited Create(CreateSuspended);
  end;

  procedure TMyThread.ShowStatus;
  // этот метод запущен главным потоком и поэтому может получить доступ ко
всем элементам графического интерфейса.
  begin
    Form1.Caption := fStatusText;
  end;

  procedure TMyThread.Execute;
  var
    newStatus : string;
  begin
    fStatusText := 'TMyThread Starting...';
    Synchronize(@Showstatus);
    fStatusText := 'TMyThread Running...';
    while (not Terminated) and ([любое необходимое условие]) do
      begin
        ...
        {здесь расположен код цикла главного (основного) потока}
        ...
        if NewStatus <> fStatusText then
          begin
            fStatusText := newStatus;
            Synchronize(@Showstatus);
          end;
      end;
  end;
```

В приложении

```
  var
    MyThread : TMyThread;
  begin
    MyThread := TMyThread.Create(True); // Таким способом он не запустится
автоматически
    ...
```

```
    {Здесь расположен код, который инициализирует всё возможное перед
запуском потоков}
    ...
    MyThread.Resume;
  end;
```

Если вы хотите, чтобы ваши приложения были более гибкими, Вы можете создавать события для потока; таким образом, Ваши синхронизированные методы не будут тесно связаны с определенной формой или классом: Вы можете прикрепить слушателей событий потока. Вот пример:

```
  Type
    TShowStatusEvent = procedure(Status: String) of Object;

    TMyThread = class(TThread)
    private
      fStatusText : string;
      FOnShowStatus: TShowStatusEvent;
      procedure ShowStatus;
    protected
      procedure Execute; override;
    public
      Constructor Create(CreateSuspended : boolean);
      property OnShowStatus: TShowStatusEvent read FOnShowStatus write
FOnShowStatus;
    end;

  constructor TMyThread.Create(CreateSuspended : boolean);
  begin
    FreeOnTerminate := True;
    inherited Create(CreateSuspended);
  end;

  procedure TMyThread.ShowStatus;
  // этот метод запущен главным потоком и поэтому может получить доступ ко
всем элементам графического интерфейса.
  begin
    if Assigned(FOnShowStatus) then
    begin
      FOnShowStatus(fStatusText);
    end;
  end;

  procedure TMyThread.Execute;
  var
    newStatus : string;
```

```
begin
  fStatusText := 'TMyThread Starting...';
  Synchronize(@Showstatus);
  fStatusText := 'TMyThread Running...';
  while (not Terminated) and ([любое необходимое условие]) do
    begin
      ...
      {здесь расположен код цикла главного (основного) потока}
      ...
      if NewStatus <> fStatusText then
        begin
          fStatusText := newStatus;
          Synchronize(@Showstatus);
        end;
    end;
end;
```

В приложении

```
Type
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { private declarations }
    MyThread: TMyThread;
    procedure ShowStatus(Status: string);
  public
    { public declarations }
  end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  inherited;
  MyThread := TMyThread.Create(true);
  MyThread.OnShowStatus := @ShowStatus;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  MyThread.Terminate;
  MyThread.Free;
  inherited;
```

```
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
 MyThread.Resume;
end;

procedure TForm1.ShowStatus(Status: string);
begin
  Label1.Caption := Status;
end;
```

Важно знать

Проверка стека в Windows

Существует потенциальная проблема с потоками в Windows, если Вы используете переключатель (?) -Ct (stack check). По причинам не совсем понятным проверка стека "переключается" при любом TThread.Create если Вы используете размер стек по-умолчанию.

Единственный работающий способ на данный момент - просто не использовать переключатель -Ct. Заметим, что это не вызывает исключения в основном потоке, но вызывает в недавно созданном. Это "выглядит" как если бы поток никогда не был запущен.

Хороший код для проверки этого и других исключений, которые могут возникнуть в создании потоков:

```
MyThread:=TThread.Create(False);
if Assigned(MyThread.FatalException) then
  raise MyThread.FatalException;
```

Этот код будет гарантировать, что любое исключение, которое произошло во время создания потока будет распространёно на главный поток.

Функция GetDC на Windows

Функция GetDC (и GetDCEx) извлекает хэндл контекста устройства (DC) для клиентской области указанного окна или для всего экрана. Ее можно использовать, например, для создания снимка экрана. Однако обратите внимание, что эта функция не является потокобезопасной: хэндл к DC может использоваться только одним потоком в любой момент времени, как описано здесь.

Многопоточность в пакетах

Пакеты, использующие многопоточность, должны добавлять флаг **-dUseCThreads** в пользовательские параметры использования. Откройте редактор пакета, затем Options > Usage > Custom и добавьте *-dUseCThreads*. Это определит данный флаг для всех проектов и пакетов,

использующих этот пакет, включая IDE. IDE и все новые приложения, созданные IDE, уже имеют следующий код в своем .lpr-файле:

```
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  cmem, // менеджер памяти Си в некоторых системах намного быстрее для
многопоточности
  {$ENDIF}{$ENDIF}
```

Heaptrc

Вы не можете использовать ключ -gh с модулем *cmem*. Ключ -gh использует модуль heaptrc, который расширяет менеджер кучи. Поэтому модуль **heaptrc** должен указываться **после** модуля **cmem**.

```
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  cmem, // менеджер памяти Си в некоторых системах намного быстрее для
многопоточности
  {$ENDIF}{$ENDIF}
  heaptrc,
```

Initialization и Finalization

Чтобы инициализировать сам объект потока, вы можете либо запустить его в приостановленном состоянии и задать его свойства и/или создать новый конструктор и вызвать унаследованный конструктор.

Примечание: Использование AfterConstruction при `CreateSuspended=false` опасно, так как поток уже запущен.

С другой стороны, деструктор может быть использован для завершения использования ресурсов объекта.

```
type
  TMyThread = class(TThread)
  private
    fRTLEvent: PRTLEvent;
  public
    procedure Create(SomeData: TSomeObject); override;
    destructor Destroy; override;
  end;

procedure TMyThread.Create(SomeData: TSomeObject; CreateSuspended: boolean);
begin
```

```
  // например: настройка событий, критических секций и других ресурсов, таких как файлы или соединения с базой данных
  RTLEventCreate(fRTLEvent);
  inherited Create(CreateSuspended);
end;

destructor TMyThread.Destroy;
begin
  RTLeventDestroy(fRTLEvent);
  inherited Destroy;
end;
```

Не LCL-программы

TThread.Synchronize требует, чтобы главный поток регулярно вызывал CheckSynchronize. LCL делает это в своем цикле. Если вы не используете цикл событий LCL, вы должны вызывать его самостоятельно.

Поддержка SMP

Хорошей новостью является то, что если ваше приложение работает все-таки правильно в многопоточном режиме, то оно уже поддерживает SMP!

Отладка многопоточных приложений с помощью Lazarus

Отладка на Lazarus требует GDB и быстро становится все более полнофункциональной и стабильной. Однако до сих пор существует несколько дистрибутивов Linux с некоторыми проблемами.

Вывод отладочной информации

В однопоточном приложении вы можете просто писать в консоль/терминал/что угодно, и порядок строк будет таким же, как они были записаны. В многопоточном приложении все сложнее. Если пишут два потока, скажем, строка написана потоком A перед строкой потока B, то строки не обязательно будут записаны в таком порядке. В то время как в linux (возможно) вы получите правильный вывод DebugLn(), то в win32 вы можете получить исключения (возможно DiskFull) из-за использования DebugLn() вне основного потока. Поэтому, чтобы избежать головной боли, используйте DebugLnThreadLog(), упомянутый ниже.

Модуль LCLProc содержит несколько функций, позволяющих каждому потоку писать в свой собственный файл журнала:

```
  procedure DbgOutThreadLog(const Msg: string); overload;
  procedure DebuglnThreadLog(const Msg: string); overload;
  procedure DebuglnThreadLog(Args: array of const); overload;
  procedure DebuglnThreadLog; overload;
```

Например: Вместо *writeln('Some text ',123);* задействуйте

```
DebuglnThreadLog(['Some text ',123]);
```

Это добавит строку 'Some text 123' в **Log<PID>.txt**, где <PID> - идентификатор процесса текущего потока.

Хорошей идеей будет удаление файлов журнала перед каждым запуском:

```
rm -f Log* && ./project1
```

Linux

Если вы попытаетесь отладить многопоточное приложение в Linux, вы столкнетесь с одной большой проблемой: менеджер рабочего стола на X-сервере может зависнуть. Это происходит, например, когда приложение захватило мышь/клавиатуру и было приостановлено gdb, а X-сервер ждет вашего приложения. Когда это происходит, вы можете просто войти в систему с другого компьютера и убить gdb или выйти из этой сессии, нажав `CTRL+ALT+F3` и убить gdb. В качестве альтернативы вы можете перезапустить оконный менеджер: введите

```
sudo /etc/init.d/gdm restart.
```

Это перезапустит менеджер рабочего стола и вернет вас на рабочий стол.

Поскольку это зависит от того, где gdb останавливает вашу программу, в некоторых случаях могут помочь некоторые хитрости: для Ubuntu x64 установите в опциях проекта для отладки требуемый файл дополнительной информации...

```
Project Options -> Compiler Options -> Linking -> Debugging: Check Use external gdb debug symbols file (-Xg).
```

Другой вариант - открыть другой рабочий стол X, запустить IDE/gdb на одном и приложение на другом, чтобы зависал только тестовый рабочий стол. Создайте новый экземпляр X с:

```
X :1 &
```

Он откроется, и когда вы переключитесь на другой рабочий стол (тот, с которым вы работаете, нажав `CTRL+ALT+F7`), вы сможете вернуться на новый графический рабочий стол с помощью `CTRL+ALT+F8` (если эта комбинация не работает, попробуйте `CTRL+ALT+F2`... этот вариант работал на Slackware).

Затем вы можете, если хотите, создать сеанс рабочего стола на X, с которого начался сеанс:

```
gnome-session --display=:1 &
```

Затем в Lazarus, в диалоге параметров запуска проекта, отметьте "Use display" и введите :1.

Теперь приложение будет запущено на втором X-сервере, и вы сможете отлаживать его на первом.

Это было проверено с Free Pascal 2.0 и Lazarus 0.9.10 на Windows и Linux.

Вместо создания новой X-сессии можно использовать Xnest. Xnest - это X-сессия в окне. Используя его, X-сервер не блокируется при отладке потоков, и отлаживать гораздо проще, не меняя терминалы.

Командная строка для запуска Xnest выглядит следующим образом

```
Xnest :1 -ac
```

для создания X-сессии на :1 и отключения контроля доступа.

Интерфейсы виджетов Lazarus

Интерфейсы win32, gtk и carbon поддерживают многопоточность. Это означает, что TThread, критические секции и Synchronize работают. Но они не являются потокобезопасными. Это означает, что только один поток в одно время может получить доступ к LCL. А поскольку главный поток никогда не должен ждать другого потока, это означает, что только главный поток имеет доступ к LCL, то есть ко всему, что связано с TControl, Application и дескрипторами виджетов LCL. В LCL есть несколько потокобезопасных функций. Например, большинство функций в блоке FileUtil являются потокобезопасными.

Использование SendMessage/PostMessage для взаимодействия между потоками

Только один поток в приложении должен вызывать API LCL, обычно это главный поток. Другие потоки могут использовать LCL с помощью ряда косвенных методов, одним из хороших вариантов является использование SendMessage или PostMessage. LCLIntf.SendMessage и LCLIntf.PostMessage отправляют сообщение в окно пула сообщений приложения.

См. также документацию по этим процедурам:

- SendMessage
- PostMessage

Разница между SendMessage и PostMessage заключается в том, как они возвращают управление вызывающему потоку. Как и Synchronize, SendMessage блокирует и управление не возвращается, пока окно, в которое было отправлено сообщение, не завершит его обработку; однако при определенных обстоятельствах SendMessage может попытаться оптимизировать обработку, оставаясь в контексте вызвавшего его потока. При PostMessage управление возвращается немедленно до некоторого определенного системой максимального количества поставленных в очередь сообщений и до тех пор, пока в куче остается место для вложенных данных.

В обоих случаях процедура, обрабатывающая сообщение (см. ниже), должна избегать вызова application.ProcessMessages, поскольку это может привести к отправке второго сообщения, которое будет обрабатываться реентерабельно. Если это неизбежно, то предпочтительнее использовать другой механизм для передачи сериализованных событий между потоками.

Вот пример того, как вторичный поток может посылать текст для отображения в элементе управления LCL главному потоку:

```
const
  WM_GOT_ERROR           = LM_USER + 2004;
  WM_VERBOSE             = LM_USER + 2005;

procedure VerboseLog(Msg: string);
var
  PError: PChar;
begin
  if MessageHandler = 0 then Exit;
  PError := StrAlloc(Length(Msg)+1);
  StrCopy(PError, PChar(Msg));
  PostMessage(formConsole.Handle, WM_VERBOSE, Integer(PError), 0);
end;
```

И пример того, как обработать это сообщение из окна:

```
const
  WM_GOT_ERROR           = LM_USER + 2004;
  WM_VERBOSE             = LM_USER + 2005;

type
  { TformConsole }

  TformConsole = class(TForm)
    DebugList: TListView;
    // ...
  private
    procedure HandleDebug(var Msg: TLMessage); message WM_VERBOSE;
  end;

var
  formConsole: TformConsole;

implementation

....

{ TformConsole }

procedure TformConsole.HandleDebug(var Msg: TLMessage);
var
  Item: TListItem;
```

```
  MsgStr: PChar;
  MsgPasStr: string;
begin
  MsgStr := PChar(Msg.wparam);
  MsgPasStr := StrPas(MsgStr);
  Item := DebugList.Items.Add;
  Item.Caption := TimeToStr(SysUtils.Now);
  Item.SubItems.Add(MsgPasStr);
  Item.MakeVisible(False);

// Далее следует что-то вроде

  TrayControl.SetError(MsgPasStr);
  StrDispose(MsgStr)
end;

end.
```

Когда на машине Linux x64 возникает ошибка: "Project XY raised exception class 'External: SIGSEGV'", необходимо изменить "Integer(PError)" на "PtrInt(PError)" в процедуре "VerboseLog". Измененная строка должна выглядеть следующим образом:

```
 PostMessage(formConsole.Handle, WM_VERBOSE, PtrInt(PError), 0);
```

Критические секции

Критическая секция представляет собой объект, который используется, чтобы удостовериться, что некоторые части кода выполняются только одним [конкурирующим] потоком одновременно. Критическая секция должна быть создана/инициализирована прежде, чем она может быть использована и будет освобождена, когда она не нужна.

Критические секции обычно используются следующим образом:

Добавляем модуль SyncObjs.

Объявляем секцию (для всех потоков, которые будут использовать этот раздел):

```
 MyCriticalSection: TRTLCriticalSection;
```

Создаём секцию:

```
 InitializeCriticalSection(MyCriticalSection);
```

Запускаем несколько потоков. Делаем что-нибудь [в монопольном режиме]:

```
  EnterCriticalSection(MyCriticalSection);
  try
    // доступ к переменным, запись файлов, отправка сетевых пакетов, и т.д.
```

```
  finally
    LeaveCriticalSection(MyCriticalSection);
  end;
```

После завершения всех потоков, освобождаем её:

```
 DeleteCriticalSection(MyCriticalSection);
```

В качестве альтернативы вы можете использовать объект TCriticalSection. Создание [объекта] выполняет инициализацию, метод Enter выполняет EnterCriticalSection, метод Leave выполняет LeaveCriticalSection, а удаление объекта - [его] удаление.

Например: 5 потоков инкрементируют счетчик: см. *lazarus/examples/multithreading/criticalsectionexample1.lpi*

Осторожно: Есть два набора из 4 вышеуказанных функций. По одному из RTL и LCL. [Для] LCL [функции] определены в модуле LCLIntf и LCLType. Оба [набора] работают почти одинаково. Вы можете использовать оба [набора функций] в одном приложении одновременно, но не должны использовать функции RTL с критическими секциями LCL и наоборот.

Общие переменные

Если несколько потоков обращаются к переменной, которая предназначена только для чтения, то не о чем беспокоиться. Но если один или несколько потоков изменяют переменную, то необходимо убедиться в том, что в данный момент времени только один поток имеет доступ к ней.

Пример: 5 потоков увеличивают переменную-счётчик. см. lazarus/examples/multithreading/criticalsectionexample1.lpi

Ожидание другого потока

Если потоку A необходим результат работы другого потока B, он должен ждать пока работа потока B будет завершена.

Важно: Главный поток никогда не должен ждать другой поток. Вместо этого используйте Synchronize (см. выше).

Пример: lazarus/examples/multithreading/waitforexample1.lpi

```
{ TThreadA }

procedure TThreadA.Execute;
begin
  Form1.ThreadB:=TThreadB.Create(false);
  // создаем событие
  WaitForB:=RTLEventCreate;
  while not Application.Terminated do begin
    // ждем бесконечно (пока B не разбудит A)
```

```
    RtlEventWaitFor(WaitForB);
    writeln('A: ThreadB.Counter='+IntToStr(Form1.ThreadB.Counter));
  end;
end;

{ TThreadB }

procedure TThreadB.Execute;
var
  i: Integer;
begin
  Counter:=0;
  while not Application.Terminated do begin
    // B: Работает ...
    Sleep(1500);
    inc(Counter);
    // будим A
    RtlEventSetEvent(Form1.ThreadA.WaitForB);
  end;
end;
```

Примечание: RtlEventSetEvent может быть вызвана перед RtlEventWaitFor. Тогда RtlEventWaitFor будет немедленно завершён (return). Используйте RTLeventResetEvent чтобы очистить флаг.

Fork (порождение)

При порождении в многопоточном приложении, следует знать, что любые потоки, созданные и запущенные ПЕРЕД вызовом fork (или fpFork), НЕ будут запущены в дочернем процессе. Как заявлено в справочной странице fork(), невозможно определить состояние любых потоков, которые были запущены перед вызовом fork().

Если есть какие-либо потоки, инициализированные перед вызовом (включая секцию инициализации), они не будут работать.

Параллельные процедуры/циклы

Особый случай работы много поточности во время выполнения кода смотрите здесь.

Распределённые вычисления

Следующим шагом после многопоточности является запуск потоков на различных машинах.

- Вы можете использовать один из TCP компонентов, таких как synapse, Inet или indy для связи. Это даст Вам наибольшую гибкость и, в основном, используется для слабо связанных Клиент / Серверных приложений.

- Вы можете использовать библиотеки передач сообщений, таких как MPICH, которые используются для высокопроизводительных вычислений (HPC, High Performance Computing) в кластерах.

Внешние потоки

Чтобы система потоков Free Pascal работала должным образом, каждый вновь созданный поток FPC должен быть инициализирован (точнее, исключение, система ввода-вывода и система threadvar для каждого потока должны быть инициализированы для работы потоков и кучи). Для вас это делается полностью автоматически, если вы используете BeginThread (или косвенно, используя класс TThread). Однако если вы используете потоки, которые были созданы без BeginThread (то есть внешние потоки), может потребоваться дополнительная работа (в настоящее время). Внешние потоки также включают те, которые были созданы во внешних библиотеках Си (.DLL/.so).

Что следует учитывать при использовании внешних потоков (возможно не понадобится во всех или будущих версиях компилятора):

- Не используйте внешние потоки вообще - используйте потоки FPC. Если вы можете получить контроль над тем, как создается поток, создайте его самостоятельно с помощью BeginThread.

Если соглашение о вызовах не подходит (например, если вашей исходной функции потока требуется соглашение о вызовах cdecl, но BeginThread требует соглашения Паскаля), создайте запись, сохраните в ней исходную требуемую функцию потока и вызовите эту функцию в вашей паскалевской функции потока:

```
type
 TCdeclThreadFunc = function (user_data:Pointer):Pointer;cdecl;

 PCdeclThreadFuncData = ^TCdeclThreadFuncData;
 TCdeclThreadFuncData = record
   Func: TCdeclThreadFunc;  //функция cdecl
   Data: Pointer;           //исходные данные
 end;

// Паскалевский поток вызывает функцию cdecl
function C2P_Translator(FuncData: pointer) : ptrint;
var
  ThreadData: TCdeclThreadFuncData;
begin
  ThreadData := PCdeclThreadFuncData(FuncData)^;
  Result := ptrint(ThreadData.Func(ThreadData.Data));
end;

procedure CreatePascalThread;
var
```

```
  ThreadData: PCdeclThreadFuncData;
begin
  New(ThreadData);
  // это желаемая функция потока cdecl
  ThreadData^.Func := func;
  ThreadData^.Data := user_data;
  // это создаст поток Паскаля
  BeginThread(@C2P_Translator, ThreadData );
end;
```

- Инициализируйте систему потоков FPC, создав фиктивный поток. Если вы не создадите поток Pascal в своем приложении, система потоков не будет инициализирована (и, следовательно, потоки не будут работать и, следовательно, куча не будет работать правильно).

```
type
   tc = class(tthread)
     procedure execute;override;
   end;

   procedure tc.execute;
   begin
   end;

{ main program }
begin
  { инициализация системы потоков}
   with tc.create(false) do
   begin
     waitfor;
     free;
   end;
   { ... ваш последующий код }
end.
```

Примечание: После инициализации системы потоков среда выполнения может установить для системной переменной IsMultiThread значение true, которое используется подпрограммами FPC для выполнения блокировок здесь и там. Вы не должны устанавливать значение этой переменной вручную.

- Если по какой-то причине у вас это не работает, попробуйте этот код в функции внешнего потока:

```
function ExternalThread(param: Pointer): LongInt; stdcall;
var
  tm: TThreadManager;
begin
  GetThreadManager(tm);
  tm.AllocateThreadVars;
  InitThread(1000000); // отрегулируйте начальный размер стека здесь

  { делаем что-то в потоке здесь ... }

  Result:=0;
end;
```

Идентификация внешних потоков

Иногда вы даже не знаете, приходится ли вам иметь дело с внешними потоками (например, если какая-то библиотека Си выполняет обратный вызов). Этот алгоритм может помочь понять это:

1. Спрашиваем у ОС идентификатор текущего потока при запуске вашего приложения

```
GetCurrentThreadID() //windows;
GetThreadID() //Darwin/macOS;
TThreadID(pthread_self) //Linux;
```

2. Спрашиваем еще раз идентификатор текущего потока внутри функции потока и сравниваем его с результатом шага 1.

Откажитесь от использования временных функций

```
ThreadSwitch()
```

Примечание: Не используйте виндовый трюк со *Sleep(0)*, так как это будет работать не на всех платформах.

См. также