Quelques exemples d'utilisation des Threads

SOMMAIRE

I. Rappels

II. Objectif

2.1 Objectif final
2.2 Etat actuel du projet

III. Contenu succinct des Projets

IV. Projet 1 : Faire clignoter un contrôle

V. Projet 2 : Prise en compte en temps réel de données par le Thread

7.1 Exemple 1
7.2 Exemple 2
7.3 Création du thread

VI. Projet 3 : Traitement long réel sur analyse de fichier

6.1. L'organisation générale de notre mini projet réel
6.2. La création des fichiers tests
6.3. Le traitement Long : travail sur Fichier

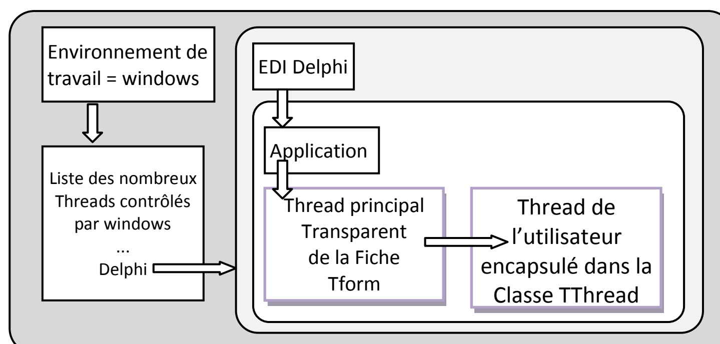
VII. Améliorations et Objectif final à atteindre

I. Rappels

Le thread permet à l'application de se comporter comme un système multitâche, ce qui signifie qu'il peut exécuter plusieurs actions en même temps.

En réalité, la charge du micro-processeur est partagée entre tous les threads y compris ceux qui sont externes à l'application...

L'instance de la TForm qui utilise le code écrit représente en quelque sorte le Thread principal mais qui n'est pas déclaré en tant que Thread. Pour soulager l'exécution du code principal de la Fiche qui s'exécute, le code doit être encapsulé dans un type Tthread.



On utilisera les threads lorsqu'une partie du code peut bloquer l'interface de l'application s'il n'était pas écrit dans une classe qui lui est spécialement réservé : **`TThread`**.

Il est utilisé pour lorsqu'une tâche est lourde, longue ou complexe mais surtout bloquante et qui peut se dérouler en arrière plan laissant ainsi la main à l'utilisateur. Par exemple un travail de sauvegarde ou d'analyse de gros fichiers, une animation visuelle etc.

Son utilisation peut être très simple ou délicate car son fonctionnement ne suit pas une logique linéaire et séquentielle.

Exemple :

```
procedure TPersoThread.Execute;
begin
  repeat
    Synchronize(MaProcedureToDo);
  until Terminated;
end;
```

Le code ci-dessus suggère que la procédure **MaProcedureToDo** qui s'exécute en arrière plan se répète plusieurs fois jusqu'à ce que le Booléen Terminated soit vrai. Il n'en ai rien, **MaProcedureToDo** ne s'exécutera qu'une seule fois.

Lors de mes recherches je me suis aidé de mes prédécesseurs, en particulier les tutoriaux suivants :

http://www.delphifr.com/tutoriaux/DELPHI-THREADS_231.aspx de Grandvisir pour les bases

Et de quelques exemples de codes :
http://www.delphifr.com/codes/THREADS-EXEMPLE-AVEC-CHRONOMETRE_11948.aspx de Delphiprog
http://www.delphifr.com/codes/COPIE-FICHIER-BYTE-BYTE-VIA-THREAD-AVEC-AVANCEMENT_12589.aspx de cyrille2
http://www.delphifr.com/codes/COPIE-FICHIER-BYTE-BYTE-VIA-THREAD-AVEC-AVANCEMENT_12589.aspx de Barbichette
et bien sûr de nombreux autres appuis sur le forum en particulier les Dissertations de Cirec et Foxi comme dab !...

II. Objectif

Les exemples trouvés jusqu'alors sur la toile ne répondaient pas à mes besoins. J'ai du donc m'approprier des concepts subtiles sur les Threads par moi même. Les projets présentés ci-dessous devraient permettre à d'autres d'en faire de même et leur faciliter la tâche.

Mon besoin et l'objectif final :

L'application principale a besoin de lancer plusieurs opérations longues sur de nombreux fichiers volumineux ou pas. Ces opérations longues sont un enchainement de tâches dont certaines pourraient être exécutées parallèlement et seront classées en une dizaine voir une vingtaine de catégories.
L'utilisateur a besoin d'accéder en détail au Log des différentes tâches si besoin est et doit suivre l'évolution de chacune des catégories d'opérations (contrôles, vérifications, analyses, fichiers de résultats...) ainsi que d'être informé d'une évaluation finale lorsque les tâches d'une catégorie est terminée.

État actuel du projet :

Dans cette version 1 écrite sous D7 perso, trois exemples (Project1, Project2, Project3) permettent une appropriation graduelle des Threads. L'objectif final n'est pas encore atteint mais s'en approche même si le code peut probablement trouver ci et là des optimisations...

Les trois projets comprennent chacun une unité portant le numéro du projet et un interface général permet d'accéder à chacun des projets.

Je publie en l'état ces exemples afin de soumettre mes codes à vos remarques éventuelles et me permettre d'améliorer et réaliser les productions à suivre. Voir à la Fin !

Description détaillée des projets :

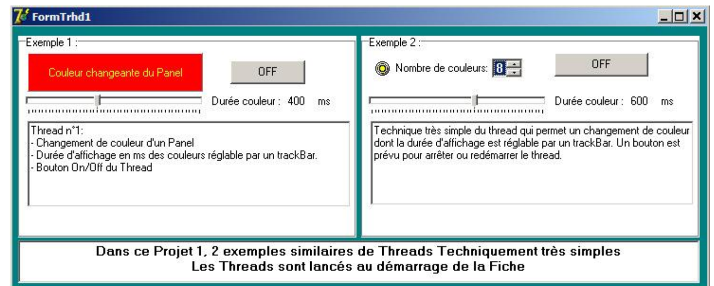
Ce fichier d'aide n'est pas un tutoriel, il n'explique pas d'une manière exhaustive tous les paramètres possibles des threads. Mais il détaillera :

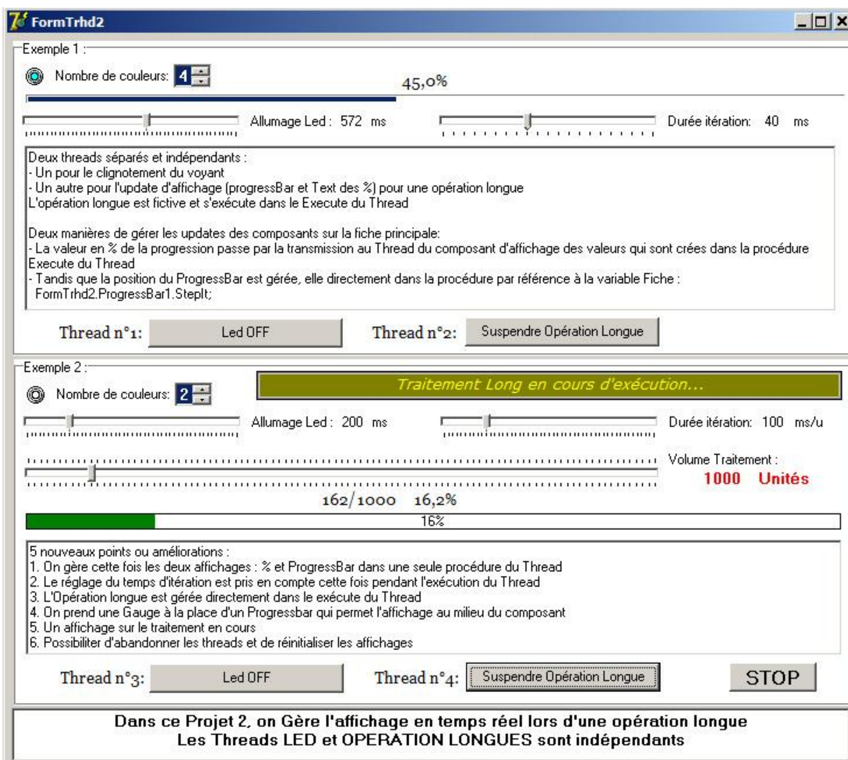
- D'une part ce qu'il est possible de faire dans chaque exemple
- D'autre part, donnera les raisons autant que je m'en souvienne qui m'ont poussé à choisir telle solution plutôt que telle autre...

III. Contenu succinct des Projets

Project1 : Deux exemples de threads intégrant le clignotement d'Objects

Les techniques de base sont employées ici dans une version minimaliste comme on en trouve dans de nombreux exemples sur la toile.





Project2 : Deux exemples de threads indépendants : voyant et progression :

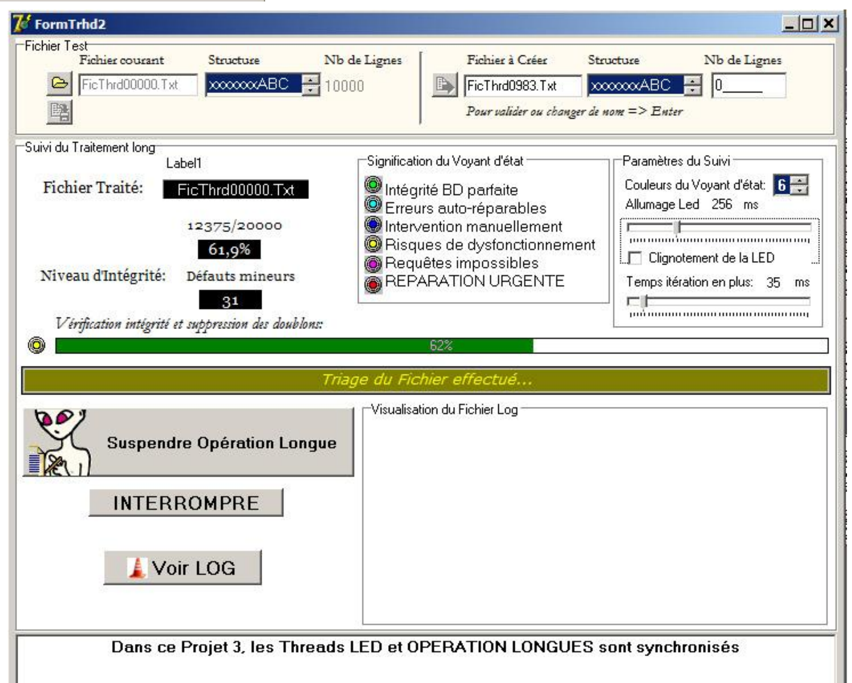
Les possibilités de commandes sont augmentées et des améliorations diverses apportées sur l'affichage des données.

L'opération longue est encore fictive

Project3 : Exemple basé sur une opération longue réelle. Un thread principal contrôlant le Thread du clignotement

Le Thread principal (opération longue sur Fichier) contrôle le Thread du voyant clignotant.

A la fin, le Voyant prend une couleur fixe selon l'évaluation de la Fin de l'opération. Un fichier de Log a été généré pendant l'opération et peut-être visualisé séparément.



IV. Projet 1 : Faire clignoter un contrôle

Imaginons un bouton qui lance une simple boucle d'itération. Dans cette boucle nous insérons un inverseur logique qui détermine la couleur d'un Panel :

```
procedure TForm1.LancerBoucleClick(Sender: TObject);
 var ii : Integer;
   inCl : Boolean;
Begin
  inCl := False;
  For ii := 1 to 100000 do
  begin
    If inCl then Panel1.Color := ClRed else Panel1.Color := clLime;
    inCl := Not inCl;
  end
end;
```

La boucle est trop rapide et empêche le changement de couleur du panel qui reste vert. Si l'on ajoute un forçage du rafraichissement du Panel avec **panel1.Update**, on observera des couleurs très rapides entre le Vert et le Rouge donnant une impression d'un mauvais fonctionnement. La solution paraît simple, il suffit de temporiser avec un « sleep(300) » de 300 ms par exemple. De cette façon, les couleurs apparaitront nettement. Mais l'inconvénient majeur est que cette temporisation sera totalement inadaptée à l'exécution rapide des 100000 itérations qui mettront plus de 8h pour se terminer...

Cependant l'Update est approprié pour l'affichage de la valeur d'un index par exemple :
StaticText.Caption := IntToStr(ii); StaticText.Update;
Enfin, la troisième arme du programmeur débutant sera d'utiliser la propriété **Panel1.doubleBuffered** := True. Cela améliore un peu le scintillement mais c'est encore trop rapide et désagréable.

Il faut donc un clignotement qui soit indépendant de la boucle. C'est ce que nous faisons dans le code suivant en utilisant un Timer réglé sur 500 ms. Le bouton lance le Timer par l'instruction :

```
Timer1.Enabled := True;
```

Puis dans l'évènement OnTimer, nous mettons notre code

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Coul := Not Coul;
  If coul then Panel1.Color := ClRed else Panel1.Color := clLime;
  Panel1.Update
end;
```

Ce code ne fonctionne pas car le Timer est complètement bloqué par le temps consacré à l'exécution de la boucle beaucoup trop rapide pour le timer et quelque soit le temps mis dans sa propriété interval. Par ailleurs, durant le déroulement de la boucle, il est impossible de stopper la boucle par un bouton de la fiche sans un traitement spécial.

Solution : Utilisation d'un Thread

Déclarons notre Thread :
Pour un simple thread, cette déclaration peut se faire dans l'unité de la Fiche qui l'utilise :

```
Type

TmyThread1 = class(TThread)
protected
  procedure Execute; override; // c'est le corps du Thread
end;

Var

MyThread1 : TMyThread1;
```

Rappel objectif : Nous souhaitons obtenir une temporisation afin de basculer la couleur d'un contrôle ou d'un voyant sans être bloqué par une boucle rapide comme précédemment. Notre premier réflexe serait d'utiliser un timer classique avec la propriété **Timer1.enabled := True** qui déclenche le timer et serait piloté par le Thread. Mais cela ne marche pas car les méthodes de la VCL doivent être incluses dans synchronize pour éviter les conflits. Il n'est pas aisé de le faire pour la méthode onTimer mais pas impossible en récrivant une fonction dans le Thread.

Puisqu'il nous faut une temporisation, étudions le code suivant :

```
procedure TmyThread1.Execute;
begin
  while not Terminated do
  begin
    Synchronize(Form1.UpdateLED);
    Sleep(Tempo);
  end;
end;

procedure TForm1.UpdateLED;
begin
  Coul := Not Coul;
  if Coul
   then image1.Picture.LoadFromFile('VV1.Jpg')
   else image1.Picture.LoadFromFile('VR1.Jpg');
end;
```

Notre temporisation est donné par Sleep(Tempo). Tempo est en ms. UpdateLED est une procédure de la Fiche qui gère la couleur du voyant avec un inverseur logique Cool qui détermine l'image du voyant VV1 (Vert) ou VR1(Rouge) qui sera chargé ici.

Synchronize permet d'éviter des conflits d'accès mémoire pour le cas ou la fiche (Thread principal) de son côté ait besoin d'accéder aux mêmes données que le Thread.

La méthode execute ci-dessus bien qu'elle semble simple et évidente pose cependant un problème pratique insoupçonné au départ. En effet, les deux lignes de codes : synchronize(Form1.UpdateLED) et Sleep(Tempo) présents dans la pseudo-boucle **While Not Terminated** vont s'exécuter une seule fois. Or comme Sleep endort le Thread durant Tempo ms, il n'y a pas possibilité d'interrompre le thread durant ce laps de temps, ce qui est inacceptable, surtout en cas d'interruption brutale par fermeture de la Fiche par exemple et qui provoquera une erreur.

Solution : Il faut diminuer le temps d'endormissement du thread pour lui permettre de déceler plus souvent un ordre de suspension. Cette solution proposée par Cirec est recommandée par l'aide delphi. La méthode **execute** Teste régulièrement le booléen **Terminated** afin de savoir s'il doit sortir de la boucle.

La méthode adoptée est donc la suivante:

```
procedure TmyThread1.Execute;
var I, V : Integer;
begin
  while not Terminated do
  begin
    V := Tempo div 100;
    Synchronize(Form1.UpdateLED);
    for I := 1 to 100 do
    begin
      if Terminated then Exit;
      Sleep(V);
    end;
  end;
end;
```

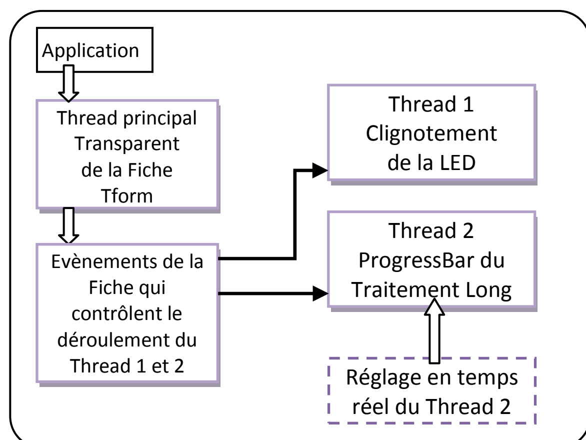
Notre temporisation est donné par Sleep(V) qui sera exécuté 100 fois dans la boucle For. Durant cette période, il est crucial de vérifier que le Thread ne demande pas à s'arrêter : if terminated then Exit.

Pseudo boucle qui scrute l'état du Thread. Le code While...end ne s'éxécute qu'une seule fois !

Etant donné que notre tempo Sleep(V) exécutée elle 100 fois dans une boucle réelle, la consigne de tempo Tempo devra être divisée au début de l'exécution par 100.

Remarque : A noter que la programmation d'un timer dynamiquement dans le Thread est inutile car son évènement OnTimer sera dépendant du thread principal de la Fiche. Le clignotement ne fonctionnera donc qu'en fin d'exécution de la boucle principale.

V. Projet 2 : Prise en compte en temps réel de données par le Thread



Dans le projet 1, les threads réalisent deux clignotements totalement indépendants entre. Ces actions ne sont pas gênés par d'autres actions qui s'exécuteraient parallèlement.

Dans le projet 2, nous allons montrer comment prendre en compte une donnée externe au Thread pendant l'exécution de ce dernier.

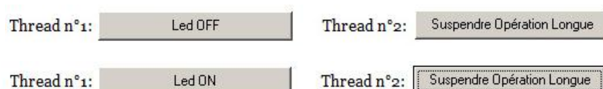
5.1 Exemple 1 :

Dans l'exemple 1, rien de bien nouveau, on reprend l'exemple d'une Led qui clignote indépendamment d'un 2ème Thread qui simule une opération longue (boucle For) . Pour suivre l'évolution de l'opération longue, on utilise une ProgressBar et un affichage en % sa valeur de progression. On notera juste deux améliorations dans le code :

1. Une fermeture brutale de la Fiche entrainerait une erreur de sortie de l'application si les threads sont en cours d'exécution. Pour éviter cela, la prise en compte de l'évènement Fiche est réglé en insérant dans la méthode **OnClose** ou **FormCloseQuery** le code de destruction de chaque thread créé :

```
if Assigned(MyThread1) then
  if not MyThread1.Terminated then myThread1.Terminate;
```

2. Par ailleurs, 2 boutons gèrent l'arrêt des Thread ou leur reprise : « MyThread2.Suspend et MyThread2.resume » en s'aidant comme précédemment d'un inverseur logique.



Cependant une simple commutation (suspend/resume) n'est pas suffisante dans le cas ou Execute a déjà été effectué. En effet, une fois terminé, le Thread est détruit et donc les instructions **suspend** ou **resume** ne peuvent plus s'éxécuter. Il faut donc créer un booléen qui indique que le code a déjà été traité : variable que l'on appellera **BoolThread_2_IsDestroy** et qui permettra donc recréer le Thread.

Le bouton de contrôle du thread devient :

```
procedure TFormTrhd2.btnOnOffThread2Click(Sender: TObject);
begin
  if BoolThread_2_IsDestroy then
  begin
    MyThread2:=TMyThread2.Create(StPourCent1);
    MyThread2.Resume; // Reprendre l'Opération Longue
    BoolThread_2_IsDestroy := False;
  end else
  if OnOff2 then
  begin
    MyThread2.Suspend; // Suspend l'opération longue
    btnOnOffThread2.Caption := 'Reprise Opération Longue';
    Screen.Cursor := crHourGlass; // Début opération longue
  end else
  begin
    MyThread2.Resume; // Relance l'Opération Longue
    btnOnOffThread2.Caption := 'Suspendre Opération Longue';
    Screen.Cursor := crDefault; // Fin opération longue
  end;
  OnOff2 := Not OnOff2;
end;
```

Re-Création du Thread et son lancement dans le cas ou il a été déjà exécuté

Commutation normale du bouton de commande en cours d'exécution de **execute** = Opération longue

5.2 Exemple 2 :

Deux perfectionnements par rapport à l'exemple 1 :

1. **Une commande arrêt total des threads**. Jusque là, les boutons sont des commandes marche/pause. Maintenant on ajoute une commande arrêt total des threads pour abandon en cours d'exécution. Cela oblige à gérer les variables et les affichages. Comme les threads utilisent ces variables et les affichages, nous ajouteront donc une procédure de **FinTraitement** dans le Thread.

```
procedure TMyThread4.FinTraitement;
begin
  With FormTrhd2 do
  begin
   STPourCent2.Caption := FormatFloat('0.0', 0.0) + '%';
   StTNbunites.Caption := '0 unité de traitement:';
   Gauge1.Progress     := 0;
   TrackBar5.Enabled   := True;
  end
end;
```

Lorsque que le booléen est détecté dans la procédure execute du Thread, il suffit alors de synchroniser la procédure avec le Thread :

```
 if Not ToutStopper then ...
else begin
  FMessage := 'Arrêt brutal des deux Threads...';
  Synchronize(FinTraitement);
  ToutStopper := False;
end;
```

Cette partie de code n'est exécuté qu'une seule fois. Par conséquent, le Booléen **ToutStopper** doit être interrogé également dans la boucle du traitement long si elle est incluse dans le **execute** du Thread. Il suffit alors de l'ajouter dans les conditions de Fin du traitement long :

```
procedure TMyThread4.Execute;
var
  EofFic: Boolean;
begin
  {initialisations locales}
  ...
  FMessage := 'Traitement Long en cours d''exécution...';
  Synchronize(AfficherMessage);

  {Boucle de traitement Long}
  while not EofFic and not Terminated and not ToutStopper do
  begin
    Inc(FCount);
    Synchronize(SetDelay); // Voir plus loin explication
    sleep(FDelay);
    if FCount > FMaxCount then EofFic := True;
    Synchronize(AfficherProgression);
  end;
  if Not ToutStopper
   ...
End;
```

2. **Une acquisition en temps réel d'un changement de donnée**. Dans les exemples précédents, les changements de durée d'allumage des voyants ne peuvent pas être modifiés après le lancement du Thread dans la procédure **execute**. On avait alors le code donné en page 6 suivant :

```
procedure TmyThread1.Execute;
  var I, V : Integer;
 begin
  while not Terminated do
  begin
    V := Tempo div 100;
    Synchronize(Form1.UpdateLED);
    for I := 1 to 100 do
    begin
      if Terminated then Exit;
      Sleep(V);
    end;
  end;
end;
```

Un changement de la valeur de la variable **Tempo** définie par un Trackbar sur la Fiche pendant l'exécution de la procédure ci-contre ne sera pas pris en compte et n'influencera pas **UpdateLed**.

Pour prendre en compte un changement de valeur des valeurs externes au Thread, il faut déclarer des variables dans le Thread et une procédure spécifique SetValeur qui prendra en compte le changement dans la boucle du traitement Long.

Exemple de Déclaration (extrait de l'exemple 2) :

```
TMyThread4 = class(TThread)
private
  FDelay    : Integer;  // Temps additionnel de simulation d'une itération
  ...
protected
  procedure   Execute; override;
public
  procedure   SetDelay;
  ...
end;
```

avec sa procédure simple d'acquisition de la valeur

```
procedure TMyThread4.SetDelay;
begin
  FDelay := Interval4;
end;
```

Et son utilisation en temps réel dans la boucle de la méthode **Execute** :

```
{Boucle de traitement Long}
while not EofFic and not Terminated and not ToutStopper do
begin
  Inc(FCount);
  Synchronize(SetDelay); // On prend en compte la nouvelle valeur de Interval4
  sleep(FDelay);         // Elle est tout de suite utilisée ici
  if FCount > FMaxCount then EofFic := True;
  Synchronize(AfficherProgression);
end;
```

5.3 Création du Thread

Dans notre dernier exemple, la création du Thread qui réalise le traitement long s'est étoffé. Voyons sa déclaration et sa procédure de création :

```
TMyThread4 = class(TThread)
  private
    FDelay    : Integer;  // Temps additionnel de simulation d'une itération
    FCount    : LongInt;  // Compteur
    FMaxCount : LongInt;  // Valeur Max du Compteur
    FMessage  : string;   // Message à écrire
  protected
    procedure   Execute; override; // Là où se passe le Traitement Long
  public
    procedure   SetDelay;
    procedure   AfficherProgression;
    procedure   AfficherMessage;
    constructor Create(aVolume : LongInt; aDelay : Integer);
    procedure   FinTraitement;
  end;
```

Les variables de la partie private qui commencent toute par F par convention d'écriture propre aux composants sont toutes des variables qui sont utilisées par le Thread. Certaines sont fixées pour toute la durée du Thread (FmaxCount) dans le constructeur Create, d'autres évoluent en interne au Thread (FCount, FMessage) et enfin l'une d'elle varie selon des valeurs externes au thread et sont prises en compte durant la durée de la procédure Execute : cas de FDelay :

```
Constructor TMyThread4.create(aVolume : LongInt; aDelay : Integer);
begin
  inherited Create(False);   // Le thread est créé suspendu
  FDelay    := aDelay;       // temps d'une itération fixé au départ par la valeur d'un TrackBar
  FCount    := 0;            // Compteur interne de progression du traitement long
  FMaxCount := aVolume;      // Valeur Max du Compteur pour calcul %
  FMessage  := 'Lancement Opération Longue...'; // Message à écrire en externe
  FreeOnTerminate := True;   // Thread détruit après exécution
end;
```

VI. Projet 3 : Traitement long réel sur analyse de fichier

6.1 L'organisation générale de notre mini projet réel:

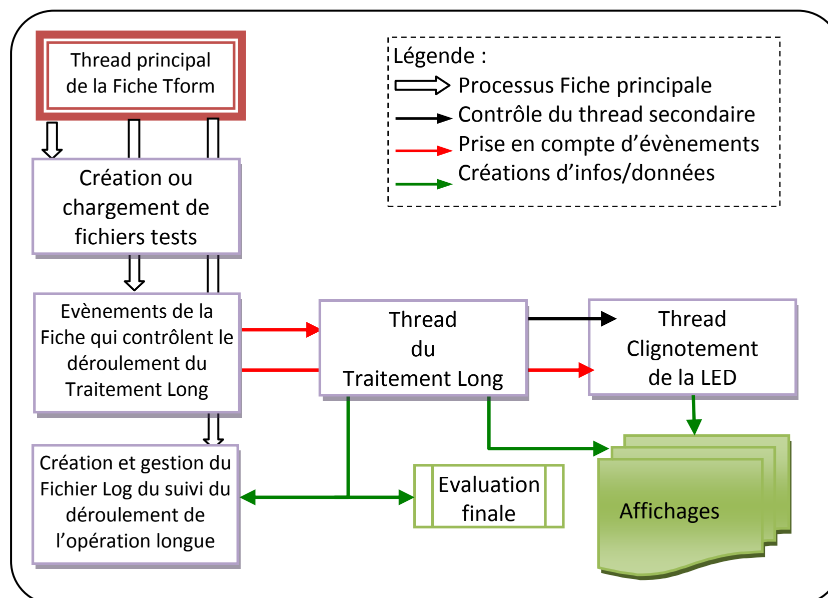
Pour les besoins des tests de l'utilisateur une première partie est consacrée à la génération automatique et aléatoire de fichiers tests. On peut ainsi créer ou ouvrir un fichier test existant. Le fichier test est paramétrable sur deux dimensions :

1. Le volume du Fichier en nombre de lignes d'enregistrement
2. La structure des enregistrements de manière à modifier les probabilités d'avoir des doublons d'enregsitrement.

La deuxième partie est consacrée à l'écriture de la procédure d'analyse du fichier qui consiste en un enchainement séquentiel de traitements sur le fichier :

1. Transfert d'une image du fichier dans un memoryStream et une TList.
2. Tri de la Tlist
3. Vérification de la longueur des lignes d'enregistrement et comptabilité des erreurs dans un fichier Log
4. Identification des doublons et suppression de ces derniers avec tenue de la comptabilité dans le Fichier Log

L'analyse peut être suspendue et reprise à tout moment et peut être arrêtée définitivement à tout moment. Durant cette analyse, les affichages du suivi des phases du traitement sont gérées en temps réel. L'utilisateur peut durant le traitement modifier certains paramètres qui sont pris immédiatement en compte par le process du traitement long. A la fin de l'analyse, une évaluation (simulée) est faite et signalée par un voyant d'état qui prend une couleur particulière en fonction du résultat de l'évaluation. Ce même voyant d'état clignote pendant la durée de l'analyse du fichier. On peut aussi consulter à la fin du traitement, le fichier Log du déroulement des opérations...



Observations :

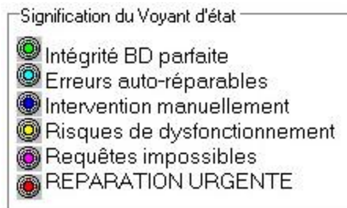
⟶ Le thread du voyant clignotant a sa propre fréquence mais cette fois il est contrôlé par l'état du Thread du traitement long. Pour cela, tous les évènements qui affectent le fonctionnement du Thread *« Traitement Long »* (Suspension, abandon, reprise) affectent simultanément le thread *« Led Clignotante »*.
Par ailleurs, on utilise une variable **FBlink** qui dépend d'un booléen indiquant si le thread du traitement long est terminé et a demandé l'évaluation du traitement (à la fin du execute du traitement long). **FBlink** est interrogé régulièrement dans le execute du clignotant avec **Synchronize(UpdateBlink)**;

```
procedure TMyThread3.UpdateBlink;
begin
  Fblink := Not BoolEvaluationDemandee;
end;
```

Le voyant clignotant est également controlable depuis la Fiche par une case à cocher sur la fiche. Elle autorise ou pas le clignotement durant le traitement long :

```
procedure TFormTrhd2.UpdateLED2;
begin
 if ChkBLed.Checked then
 begin
   ImgLed2.Picture := nil;
   ImageList2.GetBitmap(CptLed2,ImgLed2.Picture.Bitmap);
   inc(CptLed2);
   if CptLed2 = nLed2 then CptLed2 := 0;
 end
end;
```

L'affichage de la LED est donc multiple. Son état est clignotant lors de la durée du traitement long et peut prendre un nombre variable de couleurs. Lors de la fin du traitement La led prend une couleur fixe qui dépendra du résultat de l'évaluation :



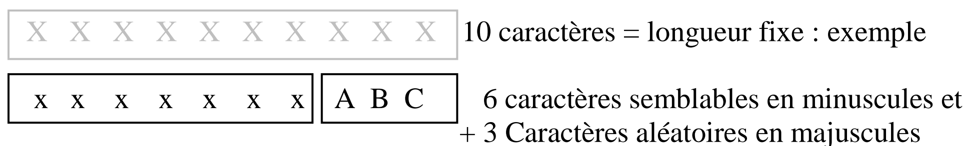
Exemple d'avertissement lumineux à la suite d'une évaluation. Cette évaluation est factice car généralement assez complexe dans la réalité. Dans notre exemple, on se contente de mettre en correspondance le nombre de doublons du fichier examiné et la couleur d'une Led.

6.2 La création des fichiers tests

Pour les besoins de notre projet et en particulier pour le test de divers paramètres agissant sur notre application, nous avons du créer un fichier selon deux axes : Le volume du fichier (nombre d'enregistrement ou de lignes) et probabilité d'apparition de données semblables (nombre de lettres répétées dans un enregistrement).

Nous générons un nom de fichier test unique en intégrant dans le nom l'heure de sa création.

Pour simplification, nous donnons à nos enregistrements une longueur fixe de 10 caractères et définissons 6 types de structure des enregistrements. Une seule structure est possible dans un fichier créé.
C'est cette structure qui permet d'agir sur la probabilité de répétition des enregistrements créés aléatoirement sur l'ensemble du fichier.



Nous aurons ainsi :
xxxxxxxxxA => 9 caractères fixes + 1 caractère aléatoire
xxxxxxxxAB
xxxxxxxABC ...
xxxxxxABCD
xxxxxABCDE
xxxxABCDEF => 4 caractères fixes + 6 caractères aléatoires

Il n'y a pas de limite de taille pour les fichiers. Cependant dans le fichier Ini, nous utilisons un fichier de 10000 enregistrements avec une structure ABC qui génèrent quelques dizaines de doublons.

Le projet 3 est didactique et ne vise pas spécialement la performance à cause de tous les affichages créés dans un but expérimental. Dans le projet final, l'information à l'égard de l'utilisateur est également privilégiée.

6.3 Le traitement long : travail sur fichier

Le fichier test étant créé et chargé, le bouton « traitement long lance une série d'opérations » :

1. **Le fichier test par défaut est FicThrd00000**. Il comprend 10 000 lignes de 10 caractères. Les 10 premiers enregistrements sont :

    nnnnnnnXFF
    sssssssKDR
    rrrrrrrLQP
    cccccccWDY
    pppppppNTS
    uuuuuuuICE
    nnnnnnnHSY
    uuuuuuuUPB
    sssssssABR
    tttttttNOC
    ...

Ce fichier n'est donc pas classé selon un ordre croissant alphabétique. Pour analyser d'éventuels doublons, il faut donc trier le fichier.

2. **Création d'une image mémoire du Fichier** : le fichier original n'étant pas trié, il est bien entendu hors de question de trier directement ce fichier en écrivant directement sur le disque. Cette opération est beaucoup trop risquée (en cas d'abandon ou de problème en cours d'exécution) d'une part et d'autre part le temps d'exécution sera beaucoup trop long.

    Nous utiliserons un transfert de données par bloc du disque vers la mémoire. Le travail en mémoire est toujours plus rapide. Cependant pour ne pas trop compliquer le code, nous nous arrangerons pour que la taille du fichier soit un multiple d'une longueur de l'enregistrement qui est fixe. Nous avons :

```
Const
   LenregDef  = 10 ;          // Longueur d'enregistrement par défaut = 10 caractères
   CR         = #13#10;       // codes Ascii de carriage return = retour de fin de ligne
   TailleBloc = LEnregDef + 2;// 1 ligne = 10 caractère + 2 (#13#10)
```

Et la méthode du parcours simplifiée du fichier dans execute du Thread :

```
Procedure TMyThread4.Execute;
 Var
  FS       : TFileStream;
  MSb      : TMemoryStream;
  …
Begin
  ...
  FS  := TFileStream.Create(FicTest, fmOpenRead ); // Ouverture fichier d'origine pour lecture
  MSb := TMemoryStream.Create;                     // Création zone mémoire
  Try
    For I:=1 to FS.Size div TailleBloc do // Division du fichier en Nombre multiple de TailleBloc
    begin
      MSb.CopyFrom(FS,TailleBloc); // Copie du disque d'un bloc vers tampon mémoire
      MSb.Position := 0; // Index début de Bloc
      MSb.Read(Ligne,TailleBloc); // Lecture d'un bloc : ici tout le tampon
      ListLignesFic.Add(Copy(ligne,1,LEnregDef)); // Ajout dans la liste des 10 caractères
      Inc(FCount); // Compteur interne du nb d'enregistrement
      Synchronize(AfficherProgression);// Affichage de la progression
      MSb.Clear; // effacement zone mémoire avant prochain transfert
      If Terminated or ToutStopper then Break; // Bouton interrompre ou Fin du Thread
    end;
  Finally
    MSb.Free;  // libération mémoire
    FS.Free;   // Fermeture fichier
  End;
  …
```

L'image du fichier est créée dans la **Tlist** **ListLignesFic**

3. **Choix d’une méthode de tri :**
   Plutôt que d’écrire un algorithme de tri, nous pouvons utiliser la méthode Sort de la Tlist qui est optimale :

   ```
   ListLignesFic.Sort; // Tri de la Liste
   ```

   L’image du fichier créé ainsi dans la Tlist sera trié en quelques secondes tout au plus même pour 100000 enregistrements.

4. **Vérification du Fichier image de la Tlist** :
   On se contente ici de parcourir la liste afin de vérifier que tous les enregistrements ont bien la même longueur. Les éventuelles erreurs sont inscrites dans le Fichier Log.
   La progression du traitement est également faite ici.

5. **Suppression des doublons et création du Fichier sans doublons**:
   La Tlist triée existe. Il suffit de la parcourir et de détecter les doublons. Tous les enregistrements uniques triés par ordre croissants sont ensuite réécrits dans un Fichier disque temporaire qui remplacera le fichier d’origine.
   Le fichier épuré créé prendra l’extension ‘.pur’ pour obliger que le fichier par défaut soit recréé pour les tests.

6. **Evaluation du niveau des erreurs du fichier**:
   Le clignotement de la led (Thread3) s’arrête et Le nombre d’erreurs cumulées sur toutes les phases déterminent la couleur du voyant d’Evaluation. Ce voyant est la même Led qui clignotait mais qui cette fois prend un état fixe.

VII. Améliorations et Objectif final à atteindre

Pour atteindre mon objectif final, il reste en effet au moins trois chantiers à réaliser :

1. Le projet 3 met le Thread principal du lancement des opérations longues dans la methode execute du Thread en question. Cette solution ne me plait qu'à moitié car elle m'oblige à séparer une quantité de code importante de la séquence de la Fiche qui lance les opérations et que j'aime bien faire pour qu'elle soit lisible et synthétique.

   Par exemple : J'aimerai bien que mon code de Fiche soit de la forme suivante :

```
Procedure TformControle.LancementSequences(...);
...
Begin
  ...
  While SequenceNotTerminated do
  Begin
    ...
    While Not EndSeqControleBD
    Begin
      ...   Opérations longues     }  Affichages de suivis
    End;
    ...
    While Not EndSeqVerifDoublons
    Begin
      ...   Opérations longues     }  Affichages de suivis
    End;
   ...
   While Not EndSeqVerifLiens
   Begin
     ...    Opérations longues     }  Affichages de suivis
   End;
   ...
  End;
```


   Dans l'organisation ci-dessus, les boucles de parcours des fichiers ont lieu dans le corps principal de la Fiche et seuls sont délégués aux threads les affichages qui concernent la progression de chaque tâche. Le fait que l'utilisateur n'ait pas la main durant l'opération longue n'est pas gênant. Il doit simplement pouvoir suspendre ou annuler une vérification.

2. Le projet 3 visualise une seule séquence de suivi. Le projet Réel en comprend plusieurs. Le projet devra lancer plusieurs séquences simultanément mais certaines séquences devront se terminer avant que d'autres séquences longues soient lancées.

3. Le projet 3 fait appel à deux threads indépendants même si le thread principal contrôle celui du clignotement du voyant de séquence. Une amélioration consisterait à intégrer le clignotement et la progression d'affichage dans un seul Thread, soit sous la forme d'un sous-thread (je ne sais pas si c'est possible), soit sous la forme d'une simple procédure du thread.

Il me reste à exploiter concrètement en particulier les ressources suivantes :

http://www.delphifr.com/tutoriaux/FIBERS-THREADS-NON-PREEMPTABLES_937.aspx de Caribensila qui parle des sous-threads et qui je pense seraient utiles dans un prochain exemple.
http://edn.embarcadero.com/article/22411 de embarcadero qui parle des sections critiques