Application Perl/Tk non figée... les threads

Utilisation des méthodes internes et modules externes

par djibril (Site personnel)

Date de publication : 2011-07-05

Dernière mise à jour : 2011-07-20

Le but de cet article est d'expliquer comment empêcher une application Perl/Tk de se figer pendant une longue tâche.
Pour ce faire, nous parlerons des avantages et inconvénients des différentes méthodes : méthodes Tk, modules externes (threads, Win32...).

1 - Introduction

La création de programmes Perl/Tk a généralement pour but d'éviter à son utilisateur d'avoir à travailler sous une console noire qui peut faire peur ! Le but est de pouvoir interagir avec l'application Perl via une interface graphique. Nous sommes très souvent confrontés à des questions existentielles ! Supposons que notre application Perl/Tk ait pour rôle d'effectuer de longs calculs et que l'on souhaite afficher en même temps l'heure ou faire autre chose. Voici les différentes questions que l'on se pose au cours du développement :

- pourquoi ma fenêtre reste figée quand je clique sur le bouton ?
- pourquoi la fenêtre ne répond plus et qu'il y a une grosse tâche blanche ?
- comment dissocier mon calcul de ma fenêtre ?
- pourquoi l'heure ne bouge plus ?
- ...

Cet article va essayer de répondre à ces questions. Nous supposerons que vous avez déjà les bases de Perl et bien évidemment de Perl/Tk. Si ce n'est pas le cas, les cours de Perl et de Perl/Tk, sans oublier les FAQ, sont à votre disposition dans la **rubrique Perl**.

2 - Problématique

Afin de vous exposer les problèmes rencontrés et expliquer comment les résoudre facilement, nous allons réaliser un script qui aura pour but de :

- rechercher les fichiers dans un répertoire au choix contenant notre motif de recherche ;
- afficher l'heure.

2-A - Tout-en-un

Pour résoudre notre problème, nous allons créer un programme que l'on nomme **recherche_fichier.pl** que voici :

recherche_fichier.pl

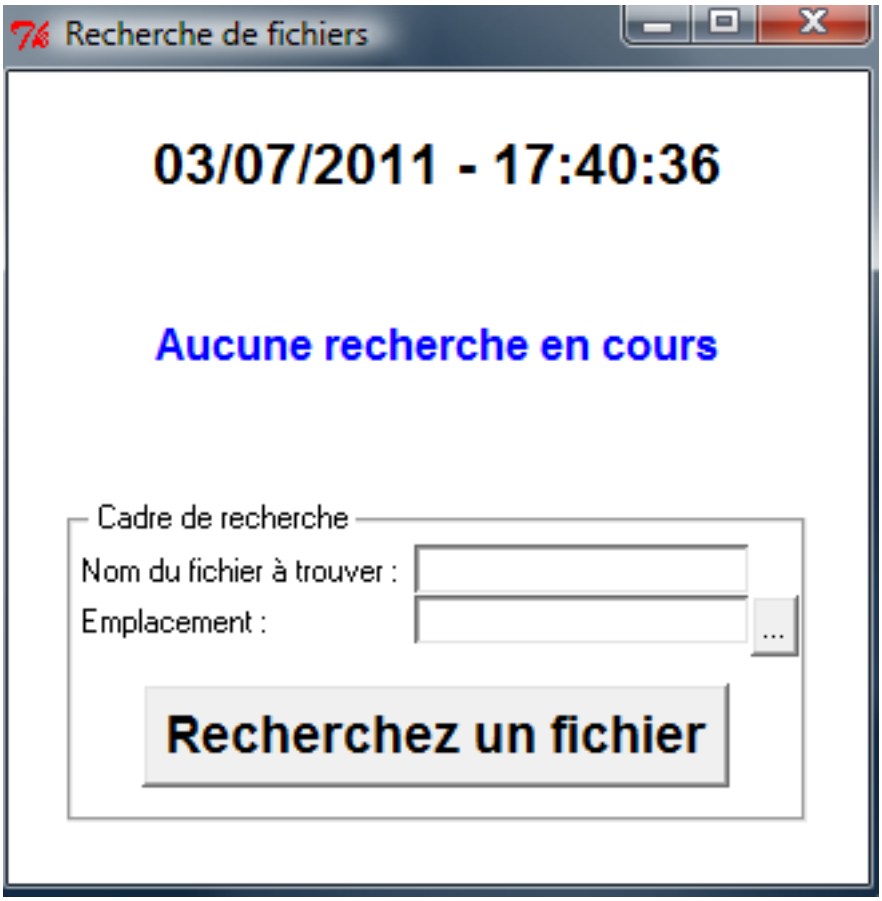
```
#!/usr/bin/perl
#=============================================================================
# Auteur : djibril
# Date   : 03/07/2011 17:47:45
# But    : Script Perl/Tk utilisant pour rechercher nos fichiers
#=============================================================================
use Carp;
use strict;
use warnings;
use Tk;
use Tk::Dialog;
use Tk::LabFrame;
use File::Find;

my $fenetre = new MainWindow(
  -title      => 'Recherche de fichiers',
  -background => 'white',
);

# Affichage de l'heure
my $date_heure = date();
my $label_date = $fenetre->Label(
  -textvariable => \$date_heure,
  -background   => 'white',
  -font         => '{Arial} 16 {bold}',
)->pack(qw/ -pady 20 /);
```

recherche_fichier.pl

```
# État de la recherche du fichier
my $etat_recherche = 'Aucune recherche en cours';
my $label_etat     = $fenetre->Label(
  -textvariable => \$etat_recherche,
  -background   => 'white',
  -foreground   => 'blue',
  -font         => '{Arial} 12 {bold}',
)->pack(qw/ -pady 20 /);

# Cadre de recherche
my $cadre = $fenetre->LabFrame(
  -label      => 'Cadre de recherche',
  -background => 'white',
)->pack(qw/ -pady 20 -padx 20 /);

my ( $motif_recherche, $repertoire_emplacement );
my $label1 = $cadre->Label( -text => 'Nom du fichier à trouver : ', -background => 'white' );
my $entry_nom_fichier = $cadre->Entry( -textvariable => \$motif_recherche );
my $label2 = $cadre->Label( -text => 'Emplacement : ', -background => 'white' );
my $entry_emplacement = $cadre->Entry( -textvariable => \$repertoire_emplacement );

my $bouton_emplacement = $cadre->Button(
  -text    => '...',
  -command => sub {
    $repertoire_emplacement = $cadre->chooseDirectory(
      -title     => 'Sélectionner un emplacement',
      -mustexist => 1,
    );
  },
);

# Affichage d'un bouton pour rechercher un fichier
my $bouton = $cadre->Button(
  -text    => 'Recherchez un fichier',
  -command => [ \&recherchez_fichier ],
  -font    => '{Arial} 14 {bold}',
);

$label1->grid( $entry_nom_fichier, '-',                -sticky => 'nw' );
$label2->grid( $entry_emplacement, $bouton_emplacement, -sticky => 'nw' );
$bouton->grid( '-',                '-',                qw/ -padx 10 -pady 10 / );

# Centrer ma fenêtre
centrer_widget($fenetre);

# Toutes les secondes, la date et l'heure évoluent
$fenetre->repeat( 1000, sub { $date_heure = date(); } );

MainLoop;

#================================================
# But : Centrer un widget automatiquement
#================================================
sub centrer_widget {
  my ($widget) = @_;

  # Height and width of the screen
  my $largeur_ecran = $widget->screenwidth();
  my $hauteur_ecran = $widget->screenheight();

  # update le widget pour récupérer les vraies dimensions
  $widget->update;
  my $largeur_widget = $widget->width;
  my $hauteur_widget = $widget->height;

  # On centre le widget en fonction de la taille de l'écran
  my $nouvelle_largeur = int( ( $largeur_ecran - $largeur_widget ) / 2 );
  my $nouvelle_hauteur = int( ( $hauteur_ecran - $hauteur_widget ) / 2 );
  $widget->geometry( $largeur_widget . "x" . $hauteur_widget . "+$nouvelle_largeur+$nouvelle_hauteur"
 );
```

recherche_fichier.pl

```
    $widget->update;

    return;
}

sub date {
    my $time = shift || time;     #$time par défaut vaut le time actuel
    my ( $seconde, $minute, $heure, $jour, $mois, $annee, $jour_semaine, $jour_annee, $heure_hiver_ou_ete
 )
       = localtime($time);
    $mois  += 1;
    $annee += 1900;

    # On rajoute 0 si le chiffre est compris entre 1 et 9
    foreach ( $seconde, $minute, $heure, $jour, $mois, $annee ) { s/^(\d)$/0$1/; }
    return "$jour/$mois/$annee - $heure:$minute:$seconde";
}

sub recherchez_fichier {

    # Recherchons le fichier
    my $fichier_trouve = 0;
    $etat_recherche = 'Recherche de fichier en cours...';
    find(
      { wanted => sub {
          if ( $_ =~ m{$motif_recherche}i ) { $fichier_trouve++; }
          }
      },
      $repertoire_emplacement
    );
    $etat_recherche = "fichier trouvé : $fichier_trouve fois";
    return;
}
```



Comme vous pouvez le constater, à l'exécution du programme, l'heure reste figée tant que la recherche de fichiers est en cours. Pour effectuer la recherche de fichiers, nous ne faisons pas appel à un programme externe, mais à une procédure que nous avons conçue « **recherchez_fichier** ». Pour éviter que notre programme ne reste figé, il est possible d'utiliser une méthode interne à Tk qui nous permettra de rafraîchir régulièrement la fenêtre afin de pouvoir voir l'heure défiler. La solution est d'utiliser la méthode **update**.

Exemple du fichier **recherche_fichier_update**. Nous avons juste effectué une légère modification de la procédure « recherchez_fichier » en faisant appel de la méthode **update** à chaque recherche de fichier. Cela permet de rafraîchir la fenêtre très régulièrement.

recherche_fichier_update.pl

```
sub recherchez_fichier {

  # Recherchons le fichier
  my $fichier_trouve = 0;
  $etat_recherche = 'Recherche de fichier en cours...';
  $fenetre->update;
  find(
    { wanted => sub {
        if ( $_ =~ m{$fichier_recherche}i ) { $fichier_trouve++; }
        $fenetre->update;
        }
    },
    $repertoire_emplacement
  );
  $etat_recherche = "fichier trouvé : $fichier_trouve fois";
  $fenetre->update;

  return;
}
```

L'usage de la méthode « **update** » permet la plupart du temps de résoudre les soucis de fenêtres figées. Bien que, dans le cas précédent, l'affichage de l'heure ne soit pas strictement régulier, le comportement reste correct. Tout cela est possible car la tâche à exécuter est conçue au sein même de notre programme. Si nous devions faire appel à un programme externe, il serait impossible de mettre un « **update** » dans le programme, vu que nous n'aurions pas accès à son contenu.

2-B - Appel de programmes externes

Nous allons maintenant concevoir un programme Tk qui fait appel à un autre programme Perl, dont le but est de rechercher des fichiers en fonction d'un motif donné. Étant donné que nous parlons le langage Perl, ce programme sera traduit dans cette même langue !
Voici notre programme externe que l'on nomme « **trouve_fichier.pl** ». Il prend en arguments un motif et le nom d'un répertoire dans lequel la recherche s'effectue. Le résultat de la recherche est imprimé dans un fichier résultat dont le nom sera fourni en argument. Nous appellerons notre programme de la sorte :

```
wperl trouve_fichier.p -d "C:\repertoire" -m "motif" -o "resultat.txt"
```

wperl permet de lancer Perl sans afficher la console DOS. Voici le code :

Programme externe : trouve_fichier.pl

```
#!/usr/bin/perl
#================================================================================
# Auteur : djibril
# Date   : 03/07/2011 18:10:19
# But    : Trouver les fichiers d'un répertoire matchant avec un motif
#================================================================================
use Carp;
use strict;
```

Programme externe : trouve_fichier.pl

```
use warnings;
use Getopt::Long;
use File::Find;

my ( $motif, $repertoire, $resultat_fichier ) = ();
GetOptions(
  'motif|m=s'      => \$motif,
  'repertoire|d=s' => \$repertoire,
  'output|o=s'     => \$resultat_fichier,
);

if ( ( not defined $motif ) || ( not defined $repertoire ) || ( not defined $resultat_fichier ) ) {
  die "USAGE : perl $0 -m <motif> -d <repertoire> -o <resultat.txt>";
}

open my $fh, '>', $resultat_fichier
  or die("Impossible d'écrire dans le fichier $resultat_fichier\n");

print {$fh} "Voici la liste de fichiers trouvés : \n";

# Rechercher le fichier
find(
  { wanted => sub {
      if ( $_ =~ m{$motif}i ) {
        print {$fh} "- $File::Find::name\n";
      }
      }
  },
  $repertoire
);

close $fh;
```

Voici maintenant le programme Tk « **recherche_tk.pl** » appelant le programme externe :

recherche_tk.pl

```
#!/usr/bin/perl
use Carp;
use strict;
use warnings;
use Tk;
use Tk::LabFrame;

my $fenetre = new MainWindow(
  -title      => 'Recherche de fichiers',
  -background => 'white',
);

# Affichage de l'heure
my $date_heure = date();
my $label_date = $fenetre->Label(
  -textvariable => \$date_heure,
  -background   => 'white',
  -font         => '{Arial} 16 {bold}',
)->pack(qw/ -pady 20 /);

# État de la recherche du fichier
my $etat_recherche = 'Aucune recherche en cours';
my $label_etat     = $fenetre->Label(
  -textvariable => \$etat_recherche,
  -background   => 'white',
  -foreground   => 'blue',
  -font         => '{Arial} 12 {bold}',
)->pack(qw/ -pady 20 /);

# Cadre de recherche
my $cadre = $fenetre->LabFrame(
  -label      => 'Cadre de recherche',
```

recherche_tk.pl

```
   -background => 'white',
)->pack(qw/ -pady 20 -padx 20 /);

my ( $fichier_recherche, $repertoire_emplacement );
my $label1 = $cadre->Label( -text => 'Nom du fichier à trouver : ', -background => 'white' );
my $entry_nom_fichier = $cadre->Entry( -textvariable => \$fichier_recherche );
my $label2 = $cadre->Label( -text => 'Emplacement : ', -background => 'white' );
my $entry_emplacement = $cadre->Entry( -textvariable => \$repertoire_emplacement );

my $bouton_emplacement = $cadre->Button(
  -text    => '...',
  -command => sub {
    $repertoire_emplacement = $cadre->chooseDirectory(
      -title     => 'Sélectionner un emplacement',
      -mustexist => 1,
    );
  },
);

# Affichage d'un bouton pour rechercher un fichier
my $bouton = $cadre->Button(
  -text    => 'Recherchez un fichier',
  -command => [ \&recherchez_fichier ],
  -font    => '{Arial} 14 {bold}',
);

$label1->grid( $entry_nom_fichier, '-',                -sticky => 'nw' );
$label2->grid( $entry_emplacement, $bouton_emplacement, -sticky => 'nw' );
$bouton->grid( '-',                '-',                qw/ -padx 10 -pady 10 / );

# Centrer ma fenêtre
centrer_widget($fenetre);

# Toutes les secondes, la date et l'heure évoluent
$fenetre->repeat( 1000, sub { $date_heure = date(); } );

MainLoop;

#================================================
# But : Centrer un widget automatiquement
#================================================
sub centrer_widget {
  my ($widget) = @_;

  # Height and width of the screen
  my $largeur_ecran = $widget->screenwidth();
  my $hauteur_ecran = $widget->screenheight();

  # update le widget pour récupérer les vraies dimensions
  $widget->update;
  my $largeur_widget = $widget->width;
  my $hauteur_widget = $widget->height;

  # On centre le widget en fonction de la taille de l'écran
  my $nouvelle_largeur = int( ( $largeur_ecran - $largeur_widget ) / 2 );
  my $nouvelle_hauteur = int( ( $hauteur_ecran - $hauteur_widget ) / 2 );
  $widget->geometry( $largeur_widget . "x" . $hauteur_widget . "+$nouvelle_largeur+$nouvelle_hauteur"
 );

  $widget->update;

  return;
}

sub date {
  my $time = shift || time;     #$time par defaut vaut le time actuel
  my ( $seconde, $minute, $heure, $jour, $mois, $annee, $jour_semaine, $jour_annee, $heure_hiver_ou_ete
 )
    = localtime($time);
  $mois  += 1;
  $annee += 1900;
```

recherche_tk.pl

```
  # On rajoute 0 si le chiffre est compris entre 1 et 9
  foreach ( $seconde, $minute, $heure, $jour, $mois, $annee ) { s/^(\d)$/0$1/; }
  return "$jour/$mois/$annee - $heure:$minute:$seconde";
}

sub recherchez_fichier {

  # Recherchons le fichier
  my $fichier_trouve = -1;
  $etat_recherche = 'Recherche de fichier en cours...';
  $fenetre->update;

  # Lancement de la recherche
  system("trouve_fichier.pl -m $fichier_recherche -d $repertoire_emplacement -o resultat.txt");

  # Lecture du fichier résultat
  open my $fh, '<', 'resultat.txt';
  while (<$fh>) { chomp; $fichier_trouve++; }
  close $fh;
  $etat_recherche = "fichier trouvé : $fichier_trouve fois";
  $fenetre->update;

  return;
}
```

Comme vous pouvez le constater à l'exécution du programme, la fenêtre est bien figée pendant la recherche et il est impossible de faire autrement. Nous essayerons de trouver des solutions via des modules Perl.

3 - Utilisation de modules Perl externes

Il existe des modules internes (**threads**...) et externes (**Win32::Process**...) nous permettant de jouer avec des processus. Nous allons en étudier quelques-uns.

3-A - Module Win32::Process

Comme vous l'aurez sans doute compris de part son nom, ce module n'est utilisable que sous Windows et vous devez l'installer. Ce module permet l'accès aux fonctions de contrôle de processus de l'API Win32. Il nous est ainsi possible de créer un nouveau processus dans lequel nous lancerons notre programme externe de recherche de fichiers. Ainsi, le programme externe tournera indépendamment de notre application Tk. Nous mettrons en place un moyen de vérifier que le processus tourne toujours ou est terminé afin de récupérer le résultat final. Avant de passer au programme en question, je vous recommande de relire les notions de références en Perl, si vous n'avez pas l'habitude de les utiliser. Notre **FAQ** pourra vous aider. Pour concevoir notre programme, nous modifierons la procédure « **recherchez_fichier** » et créerons une nouvelle « **verifier_resultat** ».

- **procédure « recherchez_fichier »**

Procédure recherchez_fichier

```
sub recherchez_fichier {

  # Recherchons le fichier
  $etat_recherche = 'Recherche de fichier en cours...';
  $fenetre->update;

  # Lancement de la recherche
  my $fichier_resultat = 'resultat.txt';
  my $commande         = "trouve_fichier.pl -m \"$motif_recherche\" "
    . " -d \"$repertoire_emplacement\" -o \"$fichier_resultat\"";
  unlink $fichier_resultat;
  my $process_objet;
  Win32::Process::Create( $process_objet, $EXECUTABLE_NAME, " $commande", 0, NORMAL_PRIORITY_CLASS, '.'
 )
```

Procédure recherchez_fichier
```
    || die Win32::FormatMessage( Win32::GetLastError() );
  my $id;
  $id = $fenetre->repeat( 500, [ \&verifier_resultat, $process_objet, $fichier_resultat, \$id ] );

  return;
}
```

Dans le code ci-dessus, nous remarquons que nous lançons le programme «**trouve_fichier.pl**» non pas à travers une commande système, mais via le module **Win32::Process**.

```
Win32::Process::Create( $process_objet, $EXECUTABLE_NAME, " $commande", 0, NORMAL_PRIORITY_CLASS, '.' )
    || die Win32::FormatMessage( Win32::GetLastError() );
```

La méthode **Create** crée un nouveau processus.
Le premier argument est une variable qui contiendra l'objet du module.
Le deuxième contient le chemin vers l'exécutable perl (*perl.exe*). Nous avons utilisé la variable prédéfinie **$EXECUTABLE_NAME** de Perl issue du module appelé en début de script.

```
use English '-no_match_vars';
```

Le troisième argument contient la commande lancé par l'API Win32.
Le quatrième argument définit la priorité avec laquelle le processus sera lancée et le dernier précise le répertoire de travail du nouveau processus.
L'important est maintenant d'être capable de savoir quand le programme lancé sera terminé, car ce dernier n'est plus lié à notre application. Pour ce faire, nous allons appeler une procédure « **verifier_resultat** » via la fonction « **repeat** » de Perl/Tk qui permet de l'exécuter à un intervalle de temps en millisecondes.

```
my $id;
$id = $fenetre->repeat( 500, [ \&verifier_resultat, $process_objet, $fichier_resultat, \$id ] );
```

La procédure est lancée toutes les 500 millisecondes et prend en argument l'objet du processus, le nom du fichier résultat (dans lequel sont listés tous les fichiers trouvés) et une référence à la variable $id. Cette dernière permettra de supprimer cet événement une fois que notre processus aura achevé sa tâche.

- **Procédure « verifier_resultat »**

Procédure verifier_resultat
```
sub verifier_resultat {
  my ( $process_objet, $fichier_resultat, $ref_id ) = @_;
  my $process_id = $process_objet->GetProcessID();
  my $exitcode;
  $process_objet->GetExitCode($exitcode);

  # Lecture du fichier résultat
  my $fichier_trouve = 0;
  open my $fh, '<', $fichier_resultat;
  while (<$fh>) { chomp; $fichier_trouve++; }
  close $fh;
  $fichier_trouve = $fichier_trouve > 0 ? $fichier_trouve-- : 0;

  # Processus terminé
  if ( $exitcode == 0 ) {
    $etat_recherche = "Fini : $fichier_trouve fichier(s) trouvé(s)";
    ${$ref_id}->cancel;
  }
  else {
    $etat_recherche
      = "Process : $process_id\n"
      . "Code de sortie : $exitcode\n"
      . "Fichiers en cours de recherche trouvés $fichier_trouve fois";
  }
```

Procédure verifier_resultat
```
    return;
}
```

Dans notre procédure, nous récupérons l'id du processus via la méthode « **GetProcessID** ». Puis nous récupérons le code de sortie du processus. Ce code est **259** lorsque le processus est en cours. Nous lisons ensuite le fichier résultat afin de compter le nombre de fichiers trouvés. Si le nombre de fichiers est supérieur à 0, le nombre de fichiers est réduit de 1 car dans le fichier résultat, la première ligne est une ligne explicative et non un fichier trouvé.
Ensuite, nous vérifions le code de sortie du processus. Si ce dernier est **« 0 »**, le processus est terminé et nous détruisons l'id (qui arrêtera le lancement la procédure « verifier_resultat ». Sinon, on met à jour l'affichage et la recherche continue.

```
Voici la liste de fichiers trouvés :
- C:/Article_Dvp/documents/perl-tk-threads-win32-poe/fichiers/perltk.odt
- C:/Article_Dvp/documents/perl-tk-threads-win32-poe/fichiers/recherche_fichier_update.pl
...
```

Remarque : Dans le gestionnaire de tâches, vous pouvez voir le processus lancé par l'API grâce au numéro du processus. Voici le programme final :

recherche_tk_win32 : programme final utilisant Win32::Process
```
#!/usr/bin/perl
use Carp;
use strict;
use warnings;
use Tk;
use Tk::LabFrame;
use Win32::Process;
use Win32;
use English '-no_match_vars';

my $fenetre = new MainWindow(
  -title      => 'Recherche de fichiers',
  -background => 'white',
);

# Affichage de l'heure
my $date_heure = date();
my $label_date = $fenetre->Label(
  -textvariable => \$date_heure,
  -background   => 'white',
  -font         => '{Arial} 16 {bold}',
)->pack(qw/ -pady 20 /);

# État de la recherche du fichier
my $etat_recherche = 'Aucune recherche en cours';
my $label_etat     = $fenetre->Label(
  -textvariable => \$etat_recherche,
  -background   => 'white',
  -foreground   => 'blue',
  -font         => '{Arial} 12 {bold}',
)->pack(qw/ -pady 20 /);

# Cadre de recherche
my $cadre = $fenetre->LabFrame(
  -label      => 'Cadre de recherche',
  -background => 'white',
)->pack(qw/ -pady 20 -padx 20 /);

my ( $motif_recherche, $repertoire_emplacement );
my $label1 = $cadre->Label( -text => 'Nom du fichier à trouver : ', -background => 'white' );
my $entry_nom_fichier = $cadre->Entry( -textvariable => \$motif_recherche );
my $label2 = $cadre->Label( -text => 'Emplacement : ', -background => 'white' );
my $entry_emplacement = $cadre->Entry( -textvariable => \$repertoire_emplacement );

my $bouton_emplacement = $cadre->Button(
```

recherche_tk_win32 : programme final utilisant Win32::Process

```
  -text    => '...',
  -command => sub {
    $repertoire_emplacement = $cadre->chooseDirectory(
      -title     => 'Sélectionner un emplacement',
      -mustexist => 1,
    );
  },
);

# Affichage d'un bouton pour rechercher un fichier
my $bouton = $cadre->Button(
  -text    => 'Recherchez un fichier',
  -command => [ \&recherchez_fichier ],
  -font    => '{Arial} 14 {bold}',
);

$label1->grid( $entry_nom_fichier, '-',                -sticky => 'nw' );
$label2->grid( $entry_emplacement, $bouton_emplacement, -sticky => 'nw' );
$bouton->grid( '-',                '-',                qw/ -padx 10 -pady 10 / );

# Centrer ma fenêtre
centrer_widget($fenetre);

# Toutes les secondes, la date et l'heure évoluent
$fenetre->repeat( 1000, sub { $date_heure = date(); } );

MainLoop;

#================================================
# But : Centrer un widget automatiquement
#================================================
sub centrer_widget {
  my ($widget) = @_;

  # Height and width of the screen
  my $largeur_ecran = $widget->screenwidth();
  my $hauteur_ecran = $widget->screenheight();

  # update le widget pour récupérer les vraies dimensions
  $widget->update;
  my $largeur_widget = $widget->width;
  my $hauteur_widget = $widget->height;

  # On centre le widget en fonction de la taille de l'écran
  my $nouvelle_largeur = int( ( $largeur_ecran - $largeur_widget ) / 2 );
  my $nouvelle_hauteur = int( ( $hauteur_ecran - $hauteur_widget ) / 2 );
  $widget->geometry( $largeur_widget . "x" . $hauteur_widget . "+$nouvelle_largeur+$nouvelle_hauteur"
 );

  $widget->update;

  return;
}

sub date {
  my $time = shift || time;     #$time par defaut vaut le time actuel
  my ( $seconde, $minute, $heure, $jour, $mois, $annee, $jour_semaine, $jour_annee, $heure_hiver_ou_ete
 )
    = localtime($time);
  $mois  += 1;
  $annee += 1900;

  # On rajoute 0 si le chiffre est compris entre 1 et 9
  foreach ( $seconde, $minute, $heure, $jour, $mois, $annee ) { s/^(\d)$/0$1/; }
  return "$jour/$mois/$annee - $heure:$minute:$seconde";
}

sub recherchez_fichier {

  # Recherchons le fichier
  $etat_recherche = 'Recherche de fichier en cours...';
```

recherche_tk_win32 : programme final utilisant Win32::Process

```
    $fenetre->update;

    # Lancement de la recherche
    my $fichier_resultat = 'resultat.txt';
    my $commande         = "trouve_fichier.pl -m \"$motif_recherche\" "
      . " -d \"$repertoire_emplacement\" -o \"$fichier_resultat\"";
    unlink $fichier_resultat;
    my $process_objet;
    Win32::Process::Create( $process_objet, $EXECUTABLE_NAME, " $commande", 0, NORMAL_PRIORITY_CLASS, '.'
  )
      || die Win32::FormatMessage( Win32::GetLastError() );
    my $id;
    $id = $fenetre->repeat( 500, [ \&verifier_resultat, $process_objet, $fichier_resultat, \$id ] );

    return;
}

sub verifier_resultat {
    my ( $process_objet, $fichier_resultat, $ref_id ) = @_;
    my $process_id = $process_objet->GetProcessID();
    my $exitcode;
    $process_objet->GetExitCode($exitcode);

    # Lecture du fichier résultat
    my $fichier_trouve = 0;
    open my $fh, '<', $fichier_resultat;
    while (<$fh>) { chomp; $fichier_trouve++; }
    close $fh;
    $fichier_trouve = $fichier_trouve > 0 ? $fichier_trouve-- : 0;

    # Processus terminé
    if ( $exitcode == 0 ) {
      $etat_recherche = "Fini : $fichier_trouve fichier(s) trouvé(s)";
      ${$ref_id}->cancel;
    }
    else {
      $etat_recherche
        = "Process : $process_id\n"
        . "Code de sortie : $exitcode\n"
        . "Fichiers en cours de recherche trouvés $fichier_trouve fois";
    }

    return;
}
```

Cette technique est très simple à mettre en place et vous permettra de pouvoir lancer des programmes externes effectuant des tâches très longues sans qu'ils ne bloquent votre application Perl/Tk. Étant donné que le processus tourne en tâche de fond, l'utilisateur de votre application peut continuer à travailler. Il peut même relancer x fois le même programme sans que les anciens ne soient terminés. C'est donc à vous de réfléchir à ce que vous souhaitez faire (par exemple, désactiver le bouton pendant que le processus tourne...). Vous pouvez tuer ce processus pour x raisons. Tout est possible avec un peu d'huile de coude !
Le seul inconvénient est qu'il permet de lancer certes, un programme externe, mais vous ne pouvez pas lancer une procédure Perl issue de votre programme. Normal, ce n'est pas son but. Pour ce faire, nous allons utiliser un autre module présent dans le Core de Perl : **threads** (Avec «**t**» en minuscule).

3-B - Module threads

3-B-1 - Qu'est ce qu'un thread ?

Un thread se traduit en français par processus léger. C'est un composant du processus principal (votre script). Chaque thread se partage la mémoire virtuelle du processus, mais possède sa propre pile d'appel (structure de données). Celui qui utilise des threads a donc l'impression que ces derniers travaillent en parallèle.

Il est important de ne pas confondre processus légers et multitâches, dont le principe est plutôt d'utiliser des processus différents.

3-B-2 - Exemple basique d'utilisation

Pour la suite de cet article, vous devez installer les modules **threads** et **threads::shared**. Ces modules font normalement déjà partie du CORE de Perl, mais la version présente n'est pas à jour et il se peut que certaines méthodes utilisées dans cet article ne fonctionnent pas. Je vous recommande donc de les installer.
Voici un script Perl (non Tk) qui vous permet de créer plusieurs threads. Il a pour but de créer dix threads qui afficheront deux lignes.

ExempleThreads
```
#!/usr/bin/perl
#=============================================================================
# Auteur : djibril
# But    : Exemple de threads
#=============================================================================
use warnings;
use strict;

use threads;

my @stockage_threads;

# Création de 10 threads.
for ( 0 .. 9 ) {
  $stockage_threads[$_] = threads->create( \&fun, $_ );
}

print "Threads crées, passons à autre chose !\n";
sleep 2;
print "Allons récupérer nos valeurs de retour...\n";

for ( 0 .. 9 ) {
  print "thread num $_  est terminé et nous retourne la valeur : "
 . $stockage_threads[$_]->join() . "\n";
}

sub fun {
  my $number = shift;
  print "Bonjour, je suis le thread num : $number\n";
  print "Mon id est : " . threads->tid() . "\n";
  sleep 2;
  print "le thread num $number meurt\n";

  return threads->tid();
}
```

Résultats
```
Threads créés, passons à autre chose!
Bonjour, je suis le thread num : 0
Mon id est : 1
Bonjour, je suis le thread num : 4
Mon id est : 5
Bonjour, je suis le thread num : 8
Bonjour, je suis le thread num : 3
Bonjour, je suis le thread num : 6
Bonjour, je suis le thread num : 1
Mon id est : 2
Bonjour, je suis le thread num : 2
Mon id est : 4
Bonjour, je suis le thread num : 5
Mon id est : 7
Bonjour, je suis le thread num : 7
Mon id est : 9
```

Résultats
```
Bonjour, je suis le thread num : 9
Mon id est : 10
Mon id est : 6
Mon id est : 8
Mon id est : 3
Allons récupérer nos valeurs de retours...
le thread num 0 meurt
le thread num 4 meurt
thread num 0  est terminé et nous retourne la valeur : 1
le thread num 1 meurt
thread num 1  est terminé et nous retourne la valeur : 2
le thread num 6 meurt
le thread num 3 meurt
le thread num 8 meurt
le thread num 5 meurt
le thread num 9 meurt
le thread num 2 meurt
le thread num 7 meurt
thread num 2  est terminé et nous retourne la valeur : 3
thread num 3  est terminé et nous retourne la valeur : 4
thread num 4  est terminé et nous retourne la valeur : 5
thread num 5  est terminé et nous retourne la valeur : 6
thread num 6  est terminé et nous retourne la valeur : 7
thread num 7  est terminé et nous retourne la valeur : 8
thread num 8  est terminé et nous retourne la valeur : 9
thread num 9  est terminé et nous retourne la valeur : 10
```

Vous remarquez que l'on a créé dix threads qui se sont exécutés en même temps. C'est la raison pour laquelle les messages sont affichés dans un ordre aléatoire.
Le code ci-dessous nous permet de créer un thread et de le stocker (l'objet) dans un tableau.

```
$stockage_threads[$_] = threads->create( \&fun, $_ );
```

La méthode **tid** nous retourne le numéro id du thread. La méthode **join** permet d'attendre que le thread se termine, de le nettoyer et de retourner les valeurs de la procédure lancée dans le thread (notamment &fun dans notre exemple). Si vous ne souhaitez pas récupérer la/les valeur(s) de retour de **join**, utilisez la méthode **detach** qui prend moins de ressources et détache votre script du thread. Celui-ci sera nettoyé proprement par Perl une fois qu'il sera terminé.
Il est très important de maîtriser un minimum les **threads** pour la suite de cet article.
Il est également impératif de comprendre comment partager des données entre le script Perl et ses threads ; c'est important pour la suite. Nous utilisons le module **threads::shared**.

Voici un exemple provenant du site **enstimac**, suivi des explications.

Partage de données - Code du site enstimac
```
use threads;
use threads::shared;

my $toto : shared = 1;
my $tata = 1;
threads->new(sub { $toto++; $tata++ })->join;

print "$toto\n"; # affiche 2 car $toto est partagé
print "$tata\n"; # affiche 1 car $tata n'est pas partagé
```

Dans le cas d'un tableau partagé, tous les éléments du tableau sont partagés, et pour une table de hachage partagée, toutes les clés et les valeurs sont partagées. Cela place des restrictions sur ce qui peut être affecté à des éléments de tableaux et de tables de hachage partagés : seules des valeurs simples ou des références à des variables partagées sont autorisées - de façon à ce qu'une variable privée ne puisse accidentellement devenir partagée. Une affectation incorrecte entraîne la mort du thread (die).
Par exemple :

Code du site enstimac
```
#!/usr/bin/perl
use threads;
use threads::shared;

my $var = 1;
my $svar : shared = 2;
my %hash : shared;

# ... créer quelques threads ...

$hash{a} = 1;      # pour tous les threads, exists($hash{a}) et $hash{a} == 1
$hash{a} = $var;   # ok - copie par valeur : même effet que précédemment
$hash{a} = $svar;  # ok - copie par valeur : même effet que précédemment
$hash{a} = \$svar; # ok - référence à une variable partagée
$hash{a} = \$var;  # entraîne la terminaison (I<die>)
delete $hash{a};   # ok - pour tous les threads, !exists($hash{a})
```

Le but de cet article n'est pas de vous faire un cours sur Perl et les threads, mais de vous exposer une méthode pour utiliser les threads avec Perl/Tk. Pour en savoir plus sur les threads, vous avez la documentation du module, des cours enstimac et une documentation interne à votre PC (**perldoc perlthrtut**).
Pour la suite de cet article, je considère que vous avez de bonnes notions sur les modules utilisés. De toute façon, les codes seront expliqués afin que vous puissiez les adapter à vos besoins.

3-B-3 - Perl Tk et les threads

3-B-3-A - Avantages et inconvénients

Avantages
1 L'utilisateur peut continuer à interagir avec l'interface Perl/Tk pendant qu'une tâche s'effectue.
2 La fenêtre Perl/Tk n'est plus figée, car la tâche s'effectue dans un autre processus léger.
3 Il est possible de partager des données entre le script et les threads.

Inconvénients
1 La version actuelle de Perl/Tk (Tk-804.028) n'est pas ***"thread safe"*** d'après les auteurs.
2 L'utilisation des threads avec Perl/Tk n'est pas simple.
3 Le partage des données entre processus légers et/ou script principal n'est pas toujours évident.
4 Il est recommandé de créer ses threads avant tout code TK et ne pas faire apparaître de code TK dans les threads.
5 On ne peut donc pas créer des threads à la volée comme bon nous semble via un clic bouton.

Parmi les inconvénients de Perl/Tk et des threads, ayez surtout conscience des pièges même des threads.

Exemple :
- les threads peuvent modifier l'état du processus complet, affectant ainsi les autres threads ;
- **chdir** dans un thread modifie le répertoire courant des autres threads et du script principal (excepté sous Windows).

Lisez la documentation **BUGS AND LIMITATIONS** de la documentation CPAN du module **threads** et **threads::shared**.
Il est important de ne pas être surpris d'un mauvais comportement de votre script à cause d'une mauvaise maîtrise des modules **threads::*** !

3-B-3-B - Erreurs courantes

Pour vous montrer l'erreur classique que l'on est amené à faire la première fois que l'on souhaite utiliser les threads avec TK, on reprend notre exercice sur la recherche de fichiers.

thread_erreurs_classiques.pl

```
#!/usr/bin/perl
#===============================================================================
# Auteur : djibril
# Date   : 03/07/2011 20:12:26
# But    : Script Perl/Tk utilisant pour les threads - erreurs classiques
#===============================================================================
use Carp;
use strict;
use warnings;
use Tk;
use Tk::LabFrame;
use File::Find;
use threads;

my $fenetre = new MainWindow(
  -title      => 'Recherche de fichiers',
  -background => 'white',
);

# Affichage de l'heure
my $date_heure = date();
my $label_date = $fenetre->Label(
  -textvariable => \$date_heure,
  -background   => 'white',
  -font         => '{Arial} 16 {bold}',
)->pack(qw/ -pady 20 /);

# État de la recherche du fichier
my $etat_recherche = 'Aucune recherche en cours';
my $label_etat     = $fenetre->Label(
  -textvariable => \$etat_recherche,
  -background   => 'white',
  -foreground   => 'blue',
  -font         => '{Arial} 12 {bold}',
)->pack(qw/ -pady 20 /);

# Cadre de recherche
my $cadre = $fenetre->LabFrame(
  -label      => 'Cadre de recherche',
  -background => 'white',
)->pack(qw/ -pady 20 -padx 20 /);

my ( $motif_recherche, $repertoire_emplacement );
my $label1 = $cadre->Label( -text => 'Nom du fichier à trouver : ', -background => 'white' );
my $entry_nom_fichier = $cadre->Entry( -textvariable => \$motif_recherche );
my $label2 = $cadre->Label( -text => 'Emplacement : ', -background => 'white' );
my $entry_emplacement = $cadre->Entry( -textvariable => \$repertoire_emplacement );

my $bouton_emplacement = $cadre->Button(
  -text    => '...',
  -command => sub {
    $repertoire_emplacement = $cadre->chooseDirectory(
      -title     => 'Sélectionner un emplacement',
      -mustexist => 1,
    );
  },
);

# Affichage d'un bouton pour rechercher un fichier
my $bouton = $cadre->Button(
  -text    => 'Recherchez un fichier',
  -command => sub {
    $etat_recherche = 'Recherche de fichier en cours...';
    $fenetre->update;
    my $Thread = threads->create( \&recherchez_fichier, $motif_recherche, $repertoire_emplacement );
    $Thread->detach();

    #$Thread->join();
    $etat_recherche = "fichier trouvé";
```

thread_erreurs_classiques.pl

```
    $fenetre->update;
  },
  -font => '{Arial} 14 {bold}',
);

$label1->grid( $entry_nom_fichier, '-',                -sticky => 'nw' );
$label2->grid( $entry_emplacement, $bouton_emplacement, -sticky => 'nw' );
$bouton->grid( '-',                '-',                qw/ -padx 10 -pady 10 / );

# Centrer ma fenêtre
centrer_widget($fenetre);

# Toutes les secondes, la date et l'heure évoluent
$fenetre->repeat( 1000, sub { $date_heure = date(); } );

MainLoop;

#================================================
# But : Centrer un widget automatiquement
#================================================
sub centrer_widget {
  my ($widget) = @_;

  # Height and width of the screen
  my $largeur_ecran = $widget->screenwidth();
  my $hauteur_ecran = $widget->screenheight();

  # update le widget pour récupérer les vraies dimensions
  $widget->update;
  my $largeur_widget = $widget->width;
  my $hauteur_widget = $widget->height;

  # On centre le widget en fonction de la taille de l'écran
  my $nouvelle_largeur = int( ( $largeur_ecran - $largeur_widget ) / 2 );
  my $nouvelle_hauteur = int( ( $hauteur_ecran - $hauteur_widget ) / 2 );
  $widget->geometry( $largeur_widget . "x" . $hauteur_widget . "+$nouvelle_largeur+$nouvelle_hauteur"
 );

  $widget->update;

  return;
}

sub date {
  my $time = shift || time;    #$time par defaut vaut le time actuel
  my ( $seconde, $minute, $heure, $jour, $mois, $annee, $jour_semaine, $jour_annee, $heure_hiver_ou_ete
 )
    = localtime($time);
  $mois  += 1;
  $annee += 1900;

  # On rajoute 0 si le chiffre est compris entre 1 et 9
  foreach ( $seconde, $minute, $heure, $jour, $mois, $annee ) { s/^(\d)$/0$1/; }
  return "$jour/$mois/$annee - $heure:$minute:$seconde";
}

sub recherchez_fichier {
  my ( $motif_recherche, $repertoire_emplacement ) = @_;

  # Recherchons le fichier
  my $fichier_trouve = 0;
  find(
    { wanted => sub {
        if ( $_ =~ m{$motif_recherche}i ) {
          $fichier_trouve++;
          print "$fichier_trouve- $File::Find::name\n";
        }
        }
    },
    $repertoire_emplacement
  );
```

thread_erreurs_classiques.pl

```
    return $fichier_trouve;
}
```

En exécutant ce programme, l'affichage est comme nous le souhaitons. Pendant l'affichage des fichiers trouvés, notre application n'est plus figée, mais on remarque ceci :

Utilisation de la méthode detach

1 la fenêtre Tk n'est pas figée ;
2 l'heure est actualisée régulièrement ;
3 le script s'arrête brusquement à la fin de l'exécution du thread avec un message de ce type *"Free to wrong pool 2ccee28 not 235e40 at C:/Perl/site/lib/Tk/Widget.pm line 98 during global destruction."*.

Utilisation de la méthode join à la place de detach

1 la fenêtre Tk reste figée tant que le thread n'est pas terminé (ce qui est bien dommage) ;
2 si on ajoute un **update** dans la procédure *"recherchez_fichier"*, on a un message d'erreur de ce type *Attempt to free non-existent shared string '_TK_RESULT_', Perl interpreter: 0x2ccc244 at ...* ;
3 Une fois le thread terminé, le script s'arrête anormalement avec les mêmes messages d'erreur cités ci-dessus.

Pourquoi ces arrêts brusques du script ?
En fait, nous violons les règles actuelles de Perl/Tk car il n'est pas **"thread safe"**.
On ne doit absolument pas mettre de code Perl/Tk dans une procédure lancée dans un thread. Or, c'est le cas ici, puisque l'on fait un *update*.

Ne me demandez pas pourquoi et quelles sont ces règles  ! Le *README* du module nous dit ceci : *Tk804.027 builds and loads into a threaded perl but is NOT yet thread safe.*
Les auteurs et personnes en charge de la maintenance de Perl/Tk ont prévu de le rendre *"thread-safe"* dans leur  **TODO** dans un futur proche  ! En attendant ces nouveautés, on va utiliser un autre procédé qui est recommandé et plus sûr.

3-B-4 - Mise en place des threads dans notre exemple

J'espère que vous n'êtes pas fatigué ! Après tous ces paragraphes et exemples de code, nous allons enfin voir comment créer proprement des threads en Perl/Tk  !

Vous devez vous mettre en tête ceci :

1 on doit créer nos threads en début de script avant même d'écrire du code Tk ;
2 on ne doit pas faire appel à du code Perl Tk dans les procédures que l'on souhaite utiliser dans nos threads.

Je vais vous exposer le concept de notre script.

Nous allons créer un thread qui tournera en tâche de fond. Son but sera de dormir si on ne lui demande rien  ou de travailler si on le met à contribution. Pour lui dire de travailler, on lui enverra un signal qu'il interceptera. Ce signal mentionne la procédure à appeler et on récupère le résultat de notre procédure.
Pour commencer, faisons appel aux modules dont on aura besoin.

Chargement des modules

```
#!/usr/bin/perl
#================================================================================
# Auteur : djibril
# Date   : 03/07/2011 14:08:50
# But    : Script Perl/Tk utilisant des threads pour rechercher nos fichiers
#================================================================================
use warnings;
```

Chargement des modules
```
use strict;

use Tk;                          # Pour créer notre GUI
use Tk::LabFrame;
use File::Find;
use threads;                     # Pour créer nos threads
use threads::shared;             # Pour partager nos données entre threads
use Time::HiRes qw( sleep );     # Pour faire des sleeps < à une seconde
```

Créons une liste associative dans laquelle on mentionne les fonctions à appeler dans notre thread. Ce hash a en clé le nom de la fonction et en valeur la référence à la procédure.

Liste associative contenant nos fonctions à lancer dans un thread
```
# Contient les fonctions à appeler dans le thread si besoin
my %fonctions_a_lancer_dans_thread = (
  'recherchez_fichier' => \&recherchez_fichier,
);
```

Ce hash sera visible dans notre thread car il a été déclaré avant même la création de ce dernier.
Déclarons maintenant les variables qui seront partagées entre le thread et le thread principal (le script) :

Déclaration des variables partagées
```
#==================================
# Threads et variables partagées
#==================================
my $tuer_thread : shared;          # Permet de tuer le thread proprement
my $nom_fonction : shared;         # Contient le nom de la fonction à appeler
my $thread_travail : shared;       # Contient la valeur permettant au thread de lancer une procédure
my @arguments_thread : shared;     # Contient les arguments à passer à une éventuelle procédure
my @resultat_fonction : shared;    # Contient le résultat des fonctions lancées dans le thread

$thread_travail = 0;               # 0 : thread ne fait rien, 1 : il bosse
$tuer_thread    = 0;               # 0 : thread en vie, 1 : thread se termine
```

Nous avons choisi de partager cinq variables :

i **$tuer_thread** contient la valeur 0 ou 1. C'est ainsi que l'on demande au thread de mourir ou non ;
ii **$nom_fonction** contient le nom de la fonction que l'on souhaite appeler dans notre thread (grâce aux hash *%fonctions_a_lancer_dans_thread*) ;
iii **$thread_travail** contient la valeur 0 ou 1. C'est ainsi que l'on demande au thread de lancer une procédure ou de dormir ;
iv **@arguments_thread** contient les arguments que l'on souhaite passer aux procédures lancées dans le thread ;
v **@resultat_fonction** contient les résultats de la procédure lancée dans le thread.

Ne vous inquiétez pas si cela reste ambigu pour l'instant, tout sera clarifié avec la suite du code !
Créons maintenant notre thread.

Création du processus léger (thread)
```
# Création du thread
my $thread = threads->create( \&notre_processus_leger );
```

Créons la procédure « **notre_processus_leger** » qui tournera continuellement dans le processus léger.

Processus léger : notre_processus_leger
```
#================================================
# notre_processus_leger
#================================================
sub notre_processus_leger {

  # Tourne en rond
  while (1) {
```

Processus léger : notre_processus_leger

```
    # demande au thread de travailler
    if ( $thread_travail == 1 ) {

      # Lance la procédure
      $fonctions_a_lancer_dans_thread{$nom_fonction}->(@arguments_thread);

      # demande au thread de dormir
      $thread_travail = 0;
    }

    # Terminer le thread
    last if ( $tuer_thread == 1 );
    sleep 0.5;
  }
  return;
}
```

Explication :
Dans notre procédure, nous avons fait une boucle *while* infinie qui permet au thread de ne jamais mourir, sauf si on le lui demande. Dans un premier temps, le thread vérifie si la variable **$thread_travail** est à 1. Si c'est le cas, cela signifie que l'on a demandé au thread de lancer une procédure (on verra plus tard comment on s'y prend). Dans le cas contraire, on vérifie si le thread doit mourir ou dormir pendant une demi-seconde.
Maintenant que nos variables sont déclarées et partagées, et notre thread créé, passons au code Perl/Tk.

Code Perl/Tk

```
#==================================
# Début du code principal Perl Tk
#==================================
my $fenetre = new MainWindow(
  -title      => 'Recherche de fichiers',
  -background => 'white',
);

# Affichage de l'heure
my $date_heure = date();
my $label_date = $fenetre->Label(
  -textvariable => \$date_heure,
  -background   => 'white',
  -font         => '{Arial} 16 {bold}',
)->pack(qw/ -pady 20 /);

# État de la recherche du fichier
my $etat_recherche = 'Aucune recherche en cours';
my $label_etat     = $fenetre->Label(
  -textvariable => \$etat_recherche,
  -background   => 'white',
  -foreground   => 'blue',
  -font         => '{Arial} 12 {bold}',
)->pack(qw/ -pady 20 /);

# Cadre de recherche
my $cadre = $fenetre->LabFrame(
  -label      => 'Cadre de recherche',
  -background => 'white',
)->pack(qw/ -pady 20 -padx 20 /);

my ( $motif_recherche, $repertoire_emplacement );
my $label1 = $cadre->Label( -text => 'Nom du fichier à trouver : ', -background => 'white' );
my $entry_nom_fichier = $cadre->Entry( -textvariable => \$motif_recherche );
my $label2 = $cadre->Label( -text => 'Emplacement : ', -background => 'white' );
my $entry_emplacement = $cadre->Entry( -textvariable => \$repertoire_emplacement );

my $bouton_emplacement = $cadre->Button(
  -text    => '...',
  -command => sub {
    $repertoire_emplacement = $cadre->chooseDirectory(
      -title     => 'Sélectionner un emplacement',
```

Code Perl/Tk

```
       -mustexist => 1,
     );
   },
);

# Affichage d'un bouton pour rechercher un fichier
my $bouton = $cadre->Button(
  -text    => 'Recherchez un fichier',
  -command => [ \&recherchez_fichier_tk ],
  -font    => '{Arial} 14 {bold}',
);

$label1->grid( $entry_nom_fichier, '-',                -sticky => 'nw' );
$label2->grid( $entry_emplacement, $bouton_emplacement, -sticky => 'nw' );
$bouton->grid( '-',                '-',                qw/ -padx 10 -pady 10 / );

# Centrer ma fenêtre
centrer_widget($fenetre);

# Toutes les secondes, la date et l'heure évoluent
$fenetre->repeat( 1000, sub { $date_heure = date(); } );

MainLoop;
```

Notre fenêtre Tk affiche régulièrement l'heure grâce à la méthode Tk **repeat**. Elle contient des cadres et champs permettant de réceptionner le motif de recherche et le répertoire où chercher nos fichiers. Notre bouton « recherchez un fichier » lancera le nécessaire pour la recherche. À la fermeture de l'application, la procédure « fermer_application » est appelée et nous permet de fermer proprement notre programme et d'arrêter le thread lancé.

Fermeture application

```
sub fermer_application {

  # Demande au thread de se terminer
  $tuer_thread = 1;

  # On attend que le thread se termine proprement
  $thread->detach();

  exit;
}
```

Nous avons également les procédures « date » et « centrer_widget » permettant respectivement de donner la date et l'heure, puis de centrer un widget (notamment notre fenêtre).

```
#================================================
# But : Obtenir la date et l'heure
#================================================
sub date {
  my $time = shift || time;    #$time par defaut vaut le time actuel
  my ( $seconde, $minute, $heure, $jour, $mois, $annee, $jour_semaine, $jour_annee, $heure_hiver_ou_ete
 )
    = localtime($time);
  $mois  += 1;
  $annee += 1900;

  # On rajoute 0 si le chiffre est compris entre 1 et 9
  foreach ( $seconde, $minute, $heure, $jour, $mois, $annee ) { s/^(\d)$/0$1/; }
  return "$jour/$mois/$annee - $heure:$minute:$seconde";
}

#================================================
# But : Centrer un widget automatiquement
#================================================
sub centrer_widget {
  my ($widget) = @_;

  # Height and width of the screen
```

```
  my $largeur_ecran = $widget->screenwidth();
  my $hauteur_ecran = $widget->screenheight();

  # update le widget pour récupérer les vraies dimensions
  $widget->update;
  my $largeur_widget = $widget->width;
  my $hauteur_widget = $widget->height;

  # On centre le widget en fonction de la taille de l'écran
  my $nouvelle_largeur = int( ( $largeur_ecran - $largeur_widget ) / 2 );
  my $nouvelle_hauteur = int( ( $hauteur_ecran - $hauteur_widget ) / 2 );
  $widget->geometry( $largeur_widget . "x" . $hauteur_widget . "+$nouvelle_largeur+$nouvelle_hauteur"
 );

  $widget->update;

  return;
}
```

Maintenant, regardons la procédure « recherchez_fichier_tk » qui est exécutée lorsque l'utilisateur lance la recherche de fichiers.

Procédure recherchez_fichier_tk

```
sub recherchez_fichier_tk {

    if
 ( not defined $motif_recherche or not defined $repertoire_emplacement or ! -d $repertoire_emplacement
 ) {
      return;
    }
    $etat_recherche = "Liste des fichiers en cours";

    # On lui indique la procédure à appeler
    $nom_fonction = "recherchez_fichier";

    # On lui donne les arguments
    @arguments_thread = ($motif_recherche, $repertoire_emplacement);

    # On va demander au thread de bosser
    $thread_travail   = 1;

  return;
}
```

Dans la procédure ci-dessus, on indique notre variable partagée : le nom de la procédure à appeler.

```
# On lui indique la procédure à appeler
    $nom_fonction = "recherchez_fichier";
```

On fait de même avec les bons arguments à passer à la procédure « recherchez_fichier » :

```
# On lui donne les arguments
    @arguments_thread = ($motif_recherche, $repertoire_emplacement);
```

Ensuite, nous demandons à notre processus léger de commencer à travailler.

```
# On va demander au thread de bosser
    $thread_travail   = 1;
```

Le fait de pointer la variable **$thread_travail** à 1 permettra au thread (dans le while) de lancer la recherche. Petit rappel :

notre_processus_leger

```
#================================================
# notre_processus_leger
```

notre_processus_leger
```
#=============================================
sub notre_processus_leger {

  # Tourne en rond
  while (1) {

    # demande au thread de travailler
    if ( $thread_travail == 1 ) {

      # Lance la procédure
      $fonctions_a_lancer_dans_thread{$nom_fonction}->(@arguments_thread);

      # demande au thread de dormir
      $thread_travail = 0;
    }

    # Terminer le thread
    last if ( $tuer_thread == 1 );
    sleep 0.5;
  }
  return;
}
```

La fonction de recherche lancée est la suivante :

Procédure recherchez_fichier
```
sub recherchez_fichier {
  my ($motif_recherche, $repertoire_emplacement) = @_;

  # Recherchons le fichier
  my $fichier_trouve = 0;
  find(
    { wanted => sub {
        if ( $_ =~ m{$motif_recherche}i ) {
          $fichier_trouve++;
          print "$fichier_trouve- $File::Find::name\n";
        }
      }
    },
    $repertoire_emplacement
  );

  return $fichier_trouve;
}
```

Voilà ! A ce stade, notre programme utilisant un thread fonctionne. Pour avoir une vue globale, le voici dans son intégralité :

Programme recherche_tk_thread.pl
```
#!/usr/bin/perl
#=============================================================================
# Auteur : djibril
# Date   : 03/07/2011 20:35:16
# But    : Script Perl/Tk utilisant des threads pour rechercher nos fichiers
#=============================================================================
use warnings;
use strict;

use Tk;                          # Pour créer notre GUI
use Tk::LabFrame;
use File::Find;
use threads;                     # Pour créer nos threads
use threads::shared;             # Pour partager nos données entre threads
use Time::HiRes qw( sleep );     # Pour faire des sleeps < à une seconde

# Contient les fonctions à appeler dans le thread si besoin
```

Programme recherche_tk_thread.pl

```
my %fonctions_a_lancer_dans_thread = (
  'recherchez_fichier' => \&recherchez_fichier,
);

#==================================
# Threads et variables partagées
#==================================
my $tuer_thread : shared;          # Permet de tuer le thread proprement
my $nom_fonction : shared;         # Contient le nom de la fonction à appeler
my $thread_travail : shared;       # Contient la valeur permettant au thread de lancer une procédure
my @arguments_thread : shared;     # Contient les arguments à passer à une éventuelle procédure
my @resultat_fonction : shared;    # Contient le résultat des fonctions lancées dans le thread

$thread_travail = 0;               # 0 : thread ne fait rien, 1 : il bosse
$tuer_thread    = 0;               # 0 : thread en vie, 1 : thread se termine

# Création du thread
my $thread = threads->create( \&notre_processus_leger );

#==================================
# Début du code principal Perl Tk
#==================================
my $fenetre = new MainWindow(
  -title      => 'Recherche de fichiers',
  -background => 'white',
);
$fenetre->protocol( "WM_DELETE_WINDOW", \&fermer_application );

# Affichage de l'heure
my $date_heure = date();
my $label_date = $fenetre->Label(
  -textvariable => \$date_heure,
  -background   => 'white',
  -font         => '{Arial} 16 {bold}',
)->pack(qw/ -pady 20 /);

# État de la recherche du fichier
my $etat_recherche = 'Aucune recherche en cours';
my $label_etat     = $fenetre->Label(
  -textvariable => \$etat_recherche,
  -background   => 'white',
  -foreground   => 'blue',
  -font         => '{Arial} 12 {bold}',
)->pack(qw/ -pady 20 /);

# Cadre de recherche
my $cadre = $fenetre->LabFrame(
  -label      => 'Cadre de recherche',
  -background => 'white',
)->pack(qw/ -pady 20 -padx 20 /);

my ( $motif_recherche, $repertoire_emplacement );
my $label1 = $cadre->Label( -text => 'Nom du fichier à trouver : ', -background => 'white' );
my $entry_nom_fichier = $cadre->Entry( -textvariable => \$motif_recherche );
my $label2 = $cadre->Label( -text => 'Emplacement : ', -background => 'white' );
my $entry_emplacement = $cadre->Entry( -textvariable => \$repertoire_emplacement );

my $bouton_emplacement = $cadre->Button(
  -text    => '...',
  -command => sub {
    $repertoire_emplacement = $cadre->chooseDirectory(
      -title     => 'Sélectionner un emplacement',
      -mustexist => 1,
    );
  },
);

# Affichage d'un bouton pour rechercher un fichier
my $bouton = $cadre->Button(
  -text    => 'Recherchez un fichier',
  -command => [ \&recherchez_fichier_tk ],
```

Programme recherche_tk_thread.pl

```
   -font    => '{Arial} 14 {bold}',
);

$label1->grid( $entry_nom_fichier, '-',                -sticky => 'nw' );
$label2->grid( $entry_emplacement, $bouton_emplacement, -sticky => 'nw' );
$bouton->grid( '-',                '-',                qw/ -padx 10 -pady 10 / );

# Centrer ma fenêtre
centrer_widget($fenetre);

# Toutes les secondes, la date et l'heure évoluent
$fenetre->repeat( 1000, sub { $date_heure = date(); } );

MainLoop;

#================================================
# notre_processus_leger
#================================================
sub notre_processus_leger {

  # Tourne en rond
  while (1) {

    # demande au thread de travailler
    if ( $thread_travail == 1 ) {

      # Lance la procédure
      $fonctions_a_lancer_dans_thread{$nom_fonction}->(@arguments_thread);

      # demande au thread de dormir
      $thread_travail = 0;
    }

    # Terminer le thread
    last if ( $tuer_thread == 1 );
    sleep 0.5;
  }
  return;
}

sub fermer_application {

  # Demande au thread de se terminer
  $tuer_thread = 1;

  # On attend que le thread se termine proprement
  $thread->detach();

  exit;
}

sub recherchez_fichier_tk {
    if
 ( not defined $motif_recherche or not defined $repertoire_emplacement or ! -d $repertoire_emplacement
 ) {
      return;
    }
    $etat_recherche = "Liste des fichiers en cours";

    # On lui indique la procédure à appeler
    $nom_fonction = "recherchez_fichier";

    # On lui donne les arguments
    @arguments_thread = ($motif_recherche, $repertoire_emplacement);

    # On va demander au thread de bosser
    $thread_travail   = 1;

  return;
}
```

Programme recherche_tk_thread.pl

```
sub recherchez_fichier {
  my ($motif_recherche, $repertoire_emplacement) = @_;

  # Recherchons le fichier
  my $fichier_trouve = 0;
  find(
    { wanted => sub {
        if ( $_ =~ m{$motif_recherche}i ) {
          $fichier_trouve++;
          print "$fichier_trouve- $File::Find::name\n";
        }
      }
    },
    $repertoire_emplacement
  );

  return $fichier_trouve;
}

#================================================
# But : Obtenir la date et l'heure
#================================================
sub date {
  my $time = shift || time;    #$time par defaut vaut le time actuel
  my ( $seconde, $minute, $heure, $jour, $mois, $annee, $jour_semaine, $jour_annee, $heure_hiver_ou_ete
 )
    = localtime($time);
  $mois  += 1;
  $annee += 1900;

  # On rajoute 0 si le chiffre est compris entre 1 et 9
  foreach ( $seconde, $minute, $heure, $jour, $mois, $annee ) { s/^(\d)$/0$1/; }
  return "$jour/$mois/$annee - $heure:$minute:$seconde";
}

#================================================
# But : Centrer un widget automatiquement
#================================================
sub centrer_widget {
  my ($widget) = @_;

  # Height and width of the screen
  my $largeur_ecran = $widget->screenwidth();
  my $hauteur_ecran = $widget->screenheight();

  # update le widget pour récupérer les vraies dimensions
  $widget->update;
  my $largeur_widget = $widget->width;
  my $hauteur_widget = $widget->height;

  # On centre le widget en fonction de la taille de l'écran
  my $nouvelle_largeur = int( ( $largeur_ecran - $largeur_widget ) / 2 );
  my $nouvelle_hauteur = int( ( $hauteur_ecran - $hauteur_widget ) / 2 );
  $widget->geometry( $largeur_widget . "x" . $hauteur_widget . "+$nouvelle_largeur+$nouvelle_hauteur"
 );

  $widget->update;

  return;
}
```

N'hésitez à relire et lancer le programme pour comprendre son fonctionnement. Mais nous n'avons pas fini !
A ce stade, voici quelques remarques.

Avantages :

- lorsque l'on clique sur le bouton « Recherchez un fichier », la fenêtre n'est plus figée ;
- l'heure s'affiche normalement et régulièrement ;
- le thread ne bogue plus et ne s'arrête pas de façon brusque ;

- à la fermeture du script, on fait appel à la méthode **detach** sans souci. On aurait pu également appeler la méthode **join**.

Inconvénients :

- pour le moment, on ne sait pas comment récupérer le résultat de la procédure lancée dans le thread ;
- dans le thread principal, on ne sait pas concrètement quand le processus léger est terminé ;
- l'utilisateur peut cliquer sur le bouton « Recherchez un fichier » alors que la recherche est encore en cours.

On a quand même déjà beaucoup d'avantages par rapport aux inconvénients, non  ? Nous allons maintenant voir comment on peut améliorer notre programme.
Il est important que vous soyez bien familier avec les notions de références en Perl pour cette partie de l'article.
Pour récupérer les résultats des procédures lancées dans notre thread, on a prévu une variable partagée.

```
my @resultat_fonction : shared;    # Contient le résultat des fonctions lancées dans le thread
```

Nous allons l'utiliser dans notre procédure « Recherchez un fichier » en modifiant la ligne de code suivante :

Procédure notre_processus_leger
```
# Lance la procédure
      $fonctions_a_lancer_dans_thread{$nom_fonction}->(@arguments_thread);
```

comme suit :

Procédure notre_processus_leger
```
# Lance la procédure
      my @resultat = $fonctions_a_lancer_dans_thread{$nom_fonction}->(@arguments_thread);
```

On peut ainsi récupérer dans un premier temps le résultat retourné par la procédure lancée. Vous allez sûrement vous demander pourquoi on n'a pas tout simplement écrit :

perl
```
# Lance la procédure
      my @resultat_fonction = $fonctions_a_lancer_dans_thread{$nom_fonction}->(@arguments_thread);
```

Si vous mettez directement le résultat dans **@resultat_fonction**, le code sera bon tant que votre procédure ne retourne que des scalaires, un tableau de scalaires partagés ou des références de tableaux (ou hash) partagées. C'est bien expliqué dans la documentation du module **threads::shared**.
Si ce n'est pas le cas, vous obtiendrez un message d'erreur du type *Thread 1 terminated abnormally: Invalid value for shared scalar at ...* et le thread s'arrêtera. Donc, prenons tout de suite de bonnes habitudes en utilisant la méthode **shared_clone** (du module thread) qui prend en argument une référence de tableau ou hash et copie tous les éléments non partagés.
NB : La méthode **shared_clone** retourne une référence de hash ou de tableau.

```
my $RefHash = shared_clone( \@ARRAY); # => retourne une référence de tableau
my $RefARRAY = shared_clone( \%HASH); # => retourne une référence de hash
```

Dans tous les cas, on utilise la méthode **shared_clone**. Comme elle nous retourne une référence, on va partager une variable s'appelant **$ref_resultat_fonction** à la place de **@resultat_fonction**, ce qui nous donne :

Procédure notre_processus_leger
```
#================================================
...
my $ref_resultat_fonction : shared; # Contient le résultat des fonctions lancées dans le thread
...
...

#================================================
# notre_processus_leger
```

Procédure notre_processus_leger

```
#================================================
sub notre_processus_leger {

  # Tourne en rond
  while (1) {

    # demande au thread de travailler
    if ( $thread_travail == 1 ) {

      # Lance la procédure
      my @resutat = $fonctions_a_lancer_dans_thread{$nom_fonction}->(@arguments_thread);
      $ref_resultat_fonction = shared_clone( \@resutat);

      # demande au thread de dormir
      $thread_travail = 0;
    }

    # Terminer le thread
    last if ( $tuer_thread == 1 );
    sleep 0.5;
  }
  return;
}
```

Notre variable contient maintenant à chaque fois le résultat de notre procédure.
Nous déterminons le moment où la procédure lancée par le processus léger est terminée et affichons le résultat. On empêche également un autre clic sur le bouton « Recherchez un fichier » tant que la recherche est en cours. Modifions notre procédure « recherchez_fichier_tk » :

Procédure recherchez_fichier_tk

```
sub recherchez_fichier_tk {
    if
 ( not defined $motif_recherche or not defined $repertoire_emplacement or ! -d $repertoire_emplacement
 ) {
      return;
    }

    # On désactive le bouton ListerFichiers
    $bouton->configure(-state => 'disabled');

    $etat_recherche = "Liste des fichiers en cours";

    # On lui indique la procédure à appeler
    $nom_fonction = "recherchez_fichier";

    # On lui donne les arguments
    @arguments_thread = ($motif_recherche, $repertoire_emplacement);

    # On va demander au thread de bosser
    $thread_travail   = 1;

  my $id;
  $id = $fenetre->repeat( 500, sub {
    if ( $thread_travail == 0 ) {
      # Thread terminé t on affiche le resulat
      my $nombre_fichier = $ref_resultat_fonction->[0];
      $etat_recherche = "$nombre_fichier fichier(s) trouvé(s)";
      $bouton->configure(-state => 'normal');
      $id->cancel;
    }
  } );

  return;
}
```

Explication :
Une fois que le thread se met à travailler, on désactive le bouton. Ensuite, via la méthode Tk **repeat**, on lance un code Perl qui vérifie (sans toutefois bloquer l'interface) toutes les 500 millisecondes si le processus léger a terminé

sa tâche ou non en vérifiant que la variable **$thread_travail** est égale à 0 ou pas. Si la variable est à zéro, la tâche est terminée, on modifie le message qui sera affiché et on réactive le bouton et détruit l'événement Tk.
Voilà, cette fois notre programme est terminé et le voici :

Programme final : recherche_tk_thread_final

```
#!/usr/bin/perl
#================================================================================
# Auteur : djibril
# Date   : 03/07/2011 14:08:50
# But    : Script Perl/Tk utilisant des threads pour rechercher nos fichiers
#================================================================================
use warnings;
use strict;

use Tk;    # Pour créer notre GUI
use Tk::LabFrame;
use File::Find;
use threads;                    # Pour créer nos threads
use threads::shared;            # Pour partager nos données entre threads
use Time::HiRes qw( sleep );    # Pour faire des sleeps < à une seconde

# Contient les fonctions à appeler dans le thread si besoin
my %fonctions_a_lancer_dans_thread = ( 'recherchez_fichier' => \&recherchez_fichier, );

#==================================
# Threads et variables partagées
#==================================
my $tuer_thread : shared;              # Permet de tuer le thread proprement
my $nom_fonction : shared;             # Contient le nom de la fonction à appeler
my $thread_travail : shared;
  # Contient la valeur permettant au thread de lancer une procédure
my @arguments_thread : shared;         # Contient les arguments à passer à une éventuelle procédure
my @resultat_fonction : shared;        # Contient le résultat des fonctions lancées dans le thread
my $ref_resultat_fonction : shared;    # Contient le résultat des fonctions lancées dans le thread

$thread_travail = 0;                   # 0 : thread ne fait rien, 1 : il bosse
$tuer_thread    = 0;                   # 0 : thread en vie, 1 : thread se termine

# Création du thread
my $thread = threads->create( \&notre_processus_leger );

#==================================
# Debut du code principal Perl Tk
#==================================
my $fenetre = new MainWindow(
  -title      => 'Recherche de fichiers',
  -background => 'white',
);
$fenetre->protocol( "WM_DELETE_WINDOW", \&fermer_application );

# Affichage de l'heure
my $date_heure = date();
my $label_date = $fenetre->Label(
  -textvariable => \$date_heure,
  -background   => 'white',
  -font         => '{Arial} 16 {bold}',
)->pack(qw/ -pady 20 /);

# État de la recherche du fichier
my $etat_recherche = 'Aucune recherche en cours';
my $label_etat     = $fenetre->Label(
  -textvariable => \$etat_recherche,
  -background   => 'white',
  -foreground   => 'blue',
  -font         => '{Arial} 12 {bold}',
)->pack(qw/ -pady 20 /);

# Cadre de recherche
my $cadre = $fenetre->LabFrame(
```

Programme final : recherche_tk_thread_final

```
   -label      => 'Cadre de recherche',
   -background => 'white',
)->pack(qw/ -pady 20 -padx 20 /);

my ( $motif_recherche, $repertoire_emplacement );
my $label1 = $cadre->Label( -text => 'Nom du fichier à trouver : ', -background => 'white' );
my $entry_nom_fichier = $cadre->Entry( -textvariable => \$motif_recherche );
my $label2 = $cadre->Label( -text => 'Emplacement : ', -background => 'white' );
my $entry_emplacement = $cadre->Entry( -textvariable => \$repertoire_emplacement );

my $bouton_emplacement = $cadre->Button(
  -text    => '...',
  -command => sub {
    $repertoire_emplacement = $cadre->chooseDirectory(
      -title     => 'Sélectionner un emplacement',
      -mustexist => 1,
    );
  },
);

# Affichage d'un bouton pour rechercher un fichier
my $bouton = $cadre->Button(
  -text    => 'Recherchez un fichier',
  -command => [ \&recherchez_fichier_tk ],
  -font    => '{Arial} 14 {bold}',
);

$label1->grid( $entry_nom_fichier, '-',                -sticky => 'nw' );
$label2->grid( $entry_emplacement, $bouton_emplacement, -sticky => 'nw' );
$bouton->grid( '-',                '-',                qw/ -padx 10 -pady 10 / );

# Centrer ma fenêtre
centrer_widget($fenetre);

# Toutes les secondes, la date et l'heure évoluent
$fenetre->repeat( 1000, sub { $date_heure = date(); } );

MainLoop;

#================================================
# notre_processus_leger
#================================================
sub notre_processus_leger {

  # Tourne en rond
  while (1) {

    # demande au thread de travailler
    if ( $thread_travail == 1 ) {

      # Lance la procédure
      my @resutat = $fonctions_a_lancer_dans_thread{$nom_fonction}->(@arguments_thread);
      $ref_resultat_fonction = shared_clone( \@resutat );

      # demande au thread de dormir
      $thread_travail = 0;
    }

    # Terminer le thread
    last if ( $tuer_thread == 1 );
    sleep 0.5;
  }
  return;
}

sub fermer_application {

  # Demande au thread de se terminer
  $tuer_thread = 1;

  # On attend que le thread se termine proprement
```

Programme final : recherche_tk_thread_final

```
  $thread->detach();

  exit;
}

sub recherchez_fichier_tk {
  if
 ( not defined $motif_recherche or not defined $repertoire_emplacement or !-d $repertoire_emplacement
 ) {
    return;
  }

  # On désactive le bouton ListerFichiers
  $bouton->configure( -state => 'disabled' );

  $etat_recherche = "Liste des fichiers en cours";

  # On lui indique la procédure à appeler
  $nom_fonction = "recherchez_fichier";

  # On lui donne les arguments
  @arguments_thread = ( $motif_recherche, $repertoire_emplacement );

  # On va demander au thread de bosser
  $thread_travail = 1;

  my $id;
  $id = $fenetre->repeat(
    500,
    sub {
      if ( $thread_travail == 0 ) {

        # Thread terminé, on affiche le résultat
        my $nombre_fichier = $ref_resultat_fonction->[0];
        $etat_recherche = "$nombre_fichier fichier(s) trouvé(s)";
        $bouton->configure( -state => 'normal' );
        $id->cancel;
      }
    }
  );

  return;
}

sub recherchez_fichier {
  my ( $motif_recherche, $repertoire_emplacement ) = @_;

  # Recherchons le fichier
  my $fichier_trouve = 0;
  find(
    { wanted => sub {
        if ( $_ =~ m{$motif_recherche}i ) {
          $fichier_trouve++;
          print "$fichier_trouve- $File::Find::name\n";
        }
        }
    },
    $repertoire_emplacement
  );

  return $fichier_trouve;
}

#================================================
# But : Obtenir la date et l'heure
#================================================
sub date {
  my $time = shift || time;     #$time par defaut vaut le time actuel
  my ( $seconde, $minute, $heure, $jour, $mois, $annee, $jour_semaine, $jour_annee, $heure_hiver_ou_ete
 )
    = localtime($time);
```

Programme final : recherche_tk_thread_final

```
   $mois  += 1;
   $annee += 1900;

   # On rajoute 0 si le chiffre est compris entre 1 et 9
   foreach ( $seconde, $minute, $heure, $jour, $mois, $annee ) { s/^(\d)$/0$1/; }
   return "$jour/$mois/$annee - $heure:$minute:$seconde";
}

#=============================================
# But : Centrer un widget automatiquement
#=============================================
sub centrer_widget {
  my ($widget) = @_;

  # Height and width of the screen
  my $largeur_ecran = $widget->screenwidth();
  my $hauteur_ecran = $widget->screenheight();

  # update le widget pour récupérer les vraies dimensions
  $widget->update;
  my $largeur_widget = $widget->width;
  my $hauteur_widget = $widget->height;

  # On centre le widget en fonction de la taille de l'écran
  my $nouvelle_largeur = int( ( $largeur_ecran - $largeur_widget ) / 2 );
  my $nouvelle_hauteur = int( ( $hauteur_ecran - $hauteur_widget ) / 2 );
  $widget->geometry( $largeur_widget . "x" . $hauteur_widget . "+$nouvelle_largeur+$nouvelle_hauteur"
 );

  $widget->update;

  return;
}
```

En résumé :

- la fenêtre n'est plus figée et l'heure s'affiche de manière régulière ;
- l'utilisateur ne peut pas cliquer sur le bouton lorsque la recherche est en cours ;
- on récupère proprement les résultats de notre listing de fichiers ;
- le thread ne s'arrête plus brusquement ;
- à la fermeture de la fenêtre Perl Tk, le thread est proprement détruit.

Voilà, vous savez maintenant comment utiliser les threads avec Perl/Tk ! Vous pouvez vous inspirer de ces programmes pour l'utilisation des threads. Adaptez-les à vos besoins ! Si vous voulez utiliser un seul thread pour pouvoir lancer diverses procédures, il vous suffit de modifier le hash **%fonctions_a_lancer_dans_thread**.

Exemples

```
my %fonctions_a_lancer_dans_thread = (
  'recherchez_fichier' => \&recherchez_fichier,
  'ZiperUnRepertoire'  => \&ZiperUnRepertoire,
  'CalculTresLong'     => \&CalculTresLong,
  'CalculTresTresLong' => \&CalculTresTresLong,
  ...
);
```

Dans notre exemple, on a attendu que le thread se rendorme pour poursuivre le script principal, mais ce n'est pas une obligation. Tout dépend de ce que vous voulez faire. Il faut juste faire attention *"à ne pas se mélanger les pinceaux"* et ne pas écraser les données par erreur. Il est possible de faire des choses plus complexes, tout est fonction de votre cahier des charges et de votre imagination. On a également décidé d'utiliser un thread, mais vous auriez pu en utiliser plusieurs, c'est toujours le même principe. A vous de bien définir ce que vous souhaitez, à penser à protéger les variables partagées si nécessaire via la méthode **lock** (du module **threads::shared**).
Pour conclure, voici quelques inconvénients à l'utilisation des threads avec Perl/Tk (et oui, il y en a quand même) :

- vous avez pu remarquer que l'on est obligé de les créer en début de script. Si l'on choisit d'en créer un seul, il nous sera impossible d'en rajouter ;

- si pour différentes raisons, la procédure lancée par votre thread produit un **die**, votre thread sera détruit. Si vous n'en aviez qu'un, votre script ne pourra donc plus fonctionner correctement ;
- il peut être important de tester que nos threads sont toujours en vie ;
- pour finir, vous avez pu constater que l'utilisation des threads en Perl/Tk n'est pas très évidente, il faut se creuser les méninges

.

Quoi qu'il en soit, un bon algorithme est nécessaire pour utiliser au mieux les threads et construire une application Perl Tk puissante  .

4 - Téléchargement des scripts

Tous les programmes conçus pour cet article sont téléchargeables en une fois **ici**. N'hésitez pas à les tester pour mieux comprendre le fonctionnement des threads ou du module Win32::Process avec Perl/Tk.

5 - Liens utiles

Quelques références sur Perl/Tk et les threads :

1. **threads** (CPAN) ;
2. **threads::shared** (CPAN) ;
3. **Win32::Process** (CPAN) ;
4. **Tutoriel sur les threads en Perl** ;
5. **http://www.perltk.org/** ;
6. **perldoc perlthrtut**.

6 - Conclusion

Dans cet article, nous avons appris comment ne pas figer une application de différentes façons : utiliser la méthode Tk (**update**), utiliser le module **Win32::Process** (uniquement sous Windows) et les threads. Pensez toujours à utiliser la méthode la plus simple tant que vous pouvez (notamment avec la méthode update), ou Win32::Process et en dernier recours, si vous n'avez vraiment pas le choix car vous utilisez un module externe pour effectuer de longs calculs, alors pensez aux threads. Gardez à l'esprit qu'en cas de "die" dans un thread, ce dernier meurt. De plus, il faut bien réfléchir au nombre de threads à créer en début de script, à la façon dont on souhaite protéger les données partagées...
N.B. Il existe sûrement d'autres modules externes permettant de lancer des programmes externes ou code Perl sans figer l'application. Je pense notamment aux modules **POE::***. Si certains lecteurs souhaitent m'aider à compléter cet article en utilisant ce module ou un autre, j'en serais ravi. Maintenant que vous êtes bien armé, à vous de jouer ! N'hésitez pas à faire des appréciations, suggestions, remarques ou corrections au sujet de cet article : .

7 - Remerciements

Je remercie **ced** et **ClaudeLELOUP** pour la relecture de cet article.