GTK4
L'essentiel

Création d'interfaces graphiques

Gérald Dumas

Programmation C

Gtk 4
L'essentiel

Gérald Dumas

4 juillet 2026

.

Édition et impression :
BoD · Books on Demand GmbH, Überseering 33, 22297 Hambourg, Allemagne

ISBN : 978-2-3226-7604-0

Dépôt légal : juillet 2026

Préface

Conducteur de train, je suis électronicien de formation. En 1981 mes parents m'offrent à Noël un Oric 1. De ce jour le virus de l'informatique ne m'a plus quitté.

En 1996 je découvre au détour d'un rayon chez un libraire un livre sur Linux qui explique comment installer la distribution Slackware. L'aventure commence non sans s'arracher quelques cheveux au passage. De fil en aiguille je me plonge dans la programmation C et jette mon dévolu sur la bibliothèque Gtk+ qui en était à la version 2. Aujourd'hui la version 3 est sur la fin et la version 4 est déjà sortie depuis un bon moment.

Je me suis dit qu'il était peut-être temps que je me lance dans l'écriture d'un livre sur cette version. Le dernier en français traitait de la version 2. Autant dire une antiquité.

À qui s'adresse cet ouvrage ?

À tous les passionnés de programmation fenêtrée qui désirent utiliser cette bibliothèque. Comme le sujet est vaste, je ne peux malheureusement pas m'appesantir sur les fondements du C. Le niveau du lecteur pour ce langage devra donc être confirmé. Il doit maîtriser la manipulation des pointeurs, sans quoi la lecture risque de lui être un peu difficile.

Puisque vous êtes en train de lire ces lignes, c'est que l'aventure ne vous fait pas peur. Je vous souhaite bonne lecture et bon apprentissage.

Introduction

Au commencement il y avait une équipe de programmeurs qui s'était lancée le défit d'écrire un programme de traitement graphique. Le projet "The Gimp" venait de naître. Pour arriver à leur fin ils avaient besoin d'outils adaptés pour gérer l'aspect graphique. La bibliothèque Gtk+ était née. Depuis elle est utilisée dans de nombres logiciels dont l'environnement Gnome qui développe ses propres widgets.

Cette bibliothèque a deux particularités. Elle est écrite en C et elle est orientée objet. C'est antinomique me direz-vous. Certes mais c'est fonctionnelle. Il existe de nombreuses passerelles dans d'autres langages comme Gtkmm+ en C++. Il existe aussi des portages en Objective-C, Guile/Scheme, Perl, Python, JavaScript, Rust, Go, TOM, Ada95, Free Pascal et Eiffel. Je vous laisse le loisir de rechercher sur le net pour ceux qui seraient intéressés.

Après la version 2 et la version 3 nous voici donc arrivés à la version 4, sujet de cet ouvrage. La bibliothèque Gtk+ est constituée de plusieurs petites bibliothèques chacune dédiée à un sujet précis. Nous aurons l'occasion de les aborder histoire de faire au maximum le tour du propriétaire.

Pour parfaire cette petite introduction ce livre n'est pas un glossaire de toutes les fonctions présentes. Il est plutôt une initiation à son utilisation. Pour avoir des précisions sur toutes les fonctions disponibles il faudra se référer à la documentation officielle accessible à l'adresse https ://www.gtk.org/docs.

Vous pouvez aussi consulter l'excellent forum https ://www.developpez.net à la rubrique C et C++. Il y a une sous-rubrique dédiée à Gtk+. Vous pourrez d'ailleurs m'y retrouver sous le pseudonyme "gerald3d".

Table des matières

Première partie

Mise en œuvre

Chapitre 1

Installation

1.1 Linux

Gtk+ vient du monde Linux. Son installation sur ce système d'exploitation est relativement simple. Utiliser votre gestionnaire d'application préféré et installez la bibliothèque Gtk+ ainsi que les paquets de développement. Pour exemple, votre serviteur est sous Debian. Il a installé les paquets suivants :

— libgtk-4-1
— libgtk-4-dev
— libgtk-4-doc (documentation officielle)

Avec ces paquets toutes les dépendances ont été installées. Comme vous pouvez le voir j'ai aussi installé le paquet de la documentation. Je vous invite à en faire de même. Installez si ce n'est déjà fait le paquet **devhelp**. Cette application vous permet de consulter la plupart des documentations développeur installées sur votre machine, dont Gtk+ il va de soit.

1.2 MacOS

Référez-vous à la page officielle de Gtk+ pour connaître la démarche à suivre dont voici le lien : https ://www.gtk.org/docs/installations/macos

1.3 Windows

Comme pour MacOS référez-vous à la page officielle de Gtk+ pour connaître la démarche à suivre dont voici le lien : https ://www.gtk.org/docs/installations/windows/

Je vous invite fortement à utiliser la méthode msys2. J'ai écrit un tutoriel à ce sujet que vous pouvez retrouver à l'adresse https ://gerald3d.developpez.com/tutoriels/gtk/codeblocks/

Chapitre 2

Compilation

2.1 Linux

Sous l'environnement natif de Gtk+ la compilation s'effectue grâce à une application qui affiche toutes les options nécessaires. Cette application est "pkg-config". Dans une console tapons la ligne suivante :

pkg-config gtk4 –cflags

Nous obtenons une liste d'options :

```
-mfpmath=sse -msse -msse2 -pthread -I/usr/include/gtk-4.0 -I/usr/include/gio-unix-2.0
-I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/harfbuzz
-I/usr/include/pango-1.0 -I/usr/include/fribidi -I/usr/include/harfbuzz
-I/usr/include/gdk-pixbuf-2.0 -I/usr/include/libpng16 -I/usr/include/x86_64-linux-gnu
-I/usr/include/cairo -I/usr/include/pixman-1 -I/usr/include/uuid -I/usr/include/freetype2
-I/usr/include/libpng16 -I/usr/include/graphene-1.0
-I/usr/lib/x86_64-linux-gnu/graphene-1.0/include
-I/usr/include/libmount -I/usr/include/blkid -I/usr/include/glib-2.0
-I/usr/lib/x86_64-linux-gnu/glib-2.0/include
```

Avec l'option –**libs** nous obtenons cette fois-ci celle pour l'édition des liens :

pkg-config gtk4 –libs

```
-lgtk-4 -lpangocairo-1.0 -lpango-1.0 -lharfbuzz -lgdk_pixbuf-2.0 -lcairo-gobject -lcairo
-lgraphene-1.0 -lgio-2.0 -lgobject-2.0 -lglib-2.0
```

Nous pouvons donc les utiliser pour avoir une ligne de commande concise :

gcc example.c -o example ‘pkg-config –cflags gtk4‘ ‘pkg-config –libs gtk4‘
(documentation officielle)

Les quotes encadrant les expressions ‘pkg-config...‘ ne sont pas des apostrophes mais les quotes accessibles via la commande AltGr+7.

2.2 MacOS

Toutes les explications nécessaires se trouvent ici :

https ://wiki.gnome.org/Projects/GTK/OSX/Building

2.3 Windows

Si vous avez utilisez la méthode msys2 vous trouverez sur la même page d'installation la manière de compiler une application Gtk+ et aussi comment créer un installateur distribuable.

Deuxième partie

GLib

Entrons directement dans le vif du sujet. La première bibliothèque est une "boite à outils". Elle fournit tout un tas d'outils de base qui permet de se faciliter la vie. Elle a aussi le mérite d'être compatible quelque soit la plateforme utilisée.

Comment l'utiliser ?

Prenons un exemple simple. Nous désirons afficher un texte en ligne de commande. Vous me voyez venir :). Essayons d'afficher un joli "Hello world" en console. Rien de plus simple me direz-vous. Effectivement avec le code suivant tout va bien dans le meilleur des mondes.

```
#include "stdio.h"

int main (int argc, char **argv) {
  printf ("Hello world\n");
  return 0;
}
```

Nous désirons maintenant émettre ce message non pas dans le canal standard mais dans le canal "error". Nous transformons notre `printf("Hello world\n");` en `fprintf(stderr, "Hello world\n");`. La GLib met à notre disposition deux fonctions qui traduisent ces deux possibilités. Nous pouvons donc écrire notre nouveau code ainsi :

```
#include <stdlib.h>
#include <glib.h>

gint
main(gint argc, gchar *argv[])
{
  g_print    ("Hello world\n"); // canal standard
  g_printerr ("Hello world\n"); // canal error

  return 0;
}
```

Quel intérêt me direz-vous ? Ici pas grand chose à part un code plus lisible. Mais dans les faits la GLib met à notre disposition une multitude de fonctions (framework) qui vont rendre notre code plus homogène et parfois plus interopérable. Si nous disposons d'une fonction qui a un comportement différent selon le système d'exploitation utilisé, alors la GLib aura une fonction équivalente qui nous permettra d'être sûr de son comportement quelque soit le système. Ainsi, avec cet exemple, nous sommes sûr d'écrire dans le bon canal quelque soit la plateforme utilisée.

Chapitre 1

Les types

1.1 les types de base

La GLib redéfinit les types de base comme suit :

char	gchar
unsigned char	guchar
int	gint
unsigned int	guint
float	gfloat
double	gdouble
...	...

La liste est longue. Je vous invite à aller voir la documentation officielle pour en connaître tout son contenu. Il existe entre autre toute une liste dont les noms finissent par 8, 16, 32 ou 64. Par exemple gint8, gint16, gint32, gint64. La GLib vous garantit la taille (en bits) de ces types quelque soit le système d'exploitation.

Toutes les déclarations de cet ouvrage utiliseront les types fournis par la GLib.

En reprenant le code exemple cela va donner :

```
#include "glib.h"

gint main (gint *argc, gchar **argv)
{
  g_print    ("Hello world\n"); // canal standrard
  g_printerr ("Hello world\n"); // canal error
  return 0;
}
```

1.1.1 Le type gboolean

Il définit le type bool au sein de Gtk+. Nous utiliserons les définitions **FALSE** et **TRUE**. Toutes les fonctions Gtk+ qui ont des variables de type booléen ou qui renvoient un booléen utilisent ce type.

1.1.2 Le type gpointer

Le type particulier `void*` est redéfinit en `gpointer`. Il existe aussi une variante constante qui prend la forme `gconstpointer`. À ces deux types est associée la valeur particulière **NULL**. elle est définie comme suit :

```
# define NULL (0L)
```

1.2 Les macros

La GLib définit aussi des macros bien utiles. Parmi elles on peut trouver celles qui commencent toutes par `G_MIN...` et `G_MAX...` Elles définissent, pour un type donné, la valeur minimale et maximale du type considéré. Par exemple si vous voulez déclarer un entier non signé et lui affecter la valeur maximale possible il suffit d'écrire :

`guint val = G_MAXUINT;` ou `guint16 val = G_MAXUINT16;`
si vous désirez travailler sur 16 bits.

Les macros standards

Plusieurs macros sont liées au système d'exploitation. Elles permettent de savoir sur quel système l'application est lancée, quel séparateur utilisé etc. Voici la liste avec une courte explication.

G_OS_WIN32	Seulement définit sous Windows.
G_OS_UNIX	Seulement définit sous Linux.
G_DIR_SEPARATOR	Retourne / sous Linux et \sous Windows
G_DIR_SEPARATOR_S	Idem que G_DIR_SEPARATOR mais est retourné sous forme de chaîne de caractères
G_SEARCHPATH_SEPARATOR	Séparateur de recherche. " :" sous Linux et ";" sous Windows
G_SEARCHPATH_SEPARATOR_S	Idem que le précédent mais est retourné sous forme de chaîne de caractères
G_MEM_ALIGN	Indique le nombre d'octets alignés en mémoire

Il existe des macros plus généralistes. En voici une liste non exhaustive. Référez-vous à la documentation officielle pour toutes les consulter.

MIN(a, b)	Renvoie la valeur la plus petite entre a et b
MAX(a, b)	Renvoie la valeur la plus grande entre a et b
ABS(a)	Renvoie la valeur absolue de a (quelque soit sont type)
CLAMP(x, low, high)	Renvoie x s'il est compris entre low et high ou low dans le cas contraire. Si low est supérieur à high le retour est indéterminé
G_APPROX_VALUE(a, b, epsilon)	Renvoie TRUE si la différence absolue entre a et b est inférieure à epsilon. FALSE dans le cas contraire

Les macros de conversion

Ces macros nous seront utiles lors de la programmation fenêtrée pour passer sous forme de pointeur un nombre.

GINT_TO_POINTER(i)	Convertit l'entier signé i en pointeur
GPOINTER_TO_INT(p)	Convertit le pointeur p en entier signé
GUINT_TO_POINTER(u)	Convertit l'entier non signé en pointeur
GPOINTER_TO_UINT(p)	Convertit le pointeur p en entier non signé
GSIZE_TO_POINTER(s)	Convertit la taille s (entier long non signé) en pointeur
GPOINTER_TO_SIZE(p)	Convertit le pointeur p en taille (entier long non signé)

Les définitions numériques

Il existe quelques définitions numériques. En voici les principales :

G_PI	π
G_PI_2	$\pi/2$
G_PI_4	$\pi/4$
G_SQRT2	$\sqrt{2}$

Il existe bien d'autres macros que celles présentées ici. Je vous invite à feuilleter la documentation officielle pour vous faire une idée des possibilités offertes.

Chapitre 2

La boucle principale

La GLib est plus qu'une boite à outils. Elle permet aussi de générer une boucle principale qui interagit avec ses propres commandes ou celles de Gtk+. En d'autres termes Gtk+ utilise la GLib pour avoir sa propre boucle de gestion d'évènements. Nous aurons l'occasion d'y revenir au chapitre GTK page 55.

Cette boucle gère tous les évènements sous forme de descripteur (fichier, socket, pipe...). Il est aussi possible de créer ses propres descripteurs.

La manière la plus simple de créer une boucle principale est d'utiliser la fonction suivante :

```
MainLoop *g_main_loop_new (GMainContext *context, gboolean is_running);
```

Une boucle GMainLoop n'est pas un fork !

GMainContext peut être fourni mais il est possible de transmettre la valeur **NULL** pour utiliser le contexte courant. C'est la deuxième solution que nous utiliserons le plus souvent. **is_running** n'est pas très important ici. S'il vaut **TRUE** la boucle est lancée immédiatement.

Pour avoir notre propre contexte, nous utiliserons la fonction suivante :

```
GMainContext *g_main_context_new (void);
```

La création d'un GMainLoop effectuée, nous pouvons y ajouter des références. Le principe de référence fonctionne sous forme d'un compteur. Une grande partie des objets des bibliothèques de Gtk+ utilise ce système. Si nous désirons ajouter une référence à un objet il suffit d'incrémenter son compteur interne de 1. ***Ce n'est pas une copie***. L'adresse de l'objet est inchangée. Les objets de la bibliothèque ont soit leur propre fonction de référencement soit elles utilisent la fonction héritée d'un de leurs parents.

Pour ajouter une référence à un GMainLoop il faut utiliser la fonction suivante :

```
/* Cette fonction renvoie le pointeur "loop" transmis. */
GMainLoop *g_main_loop_ref (GMainLoop *loop);
```

Pourquoi voir ce principe de référence maintenant ? Nous venons de créer une nouvelle boucle principale avec `g_main_loop_new ();`. Cette fonction renvoie un pointeur sur une

zone mémoire allouée dans le tas. Il faut pouvoir libérer cette mémoire lorsque son utilisation est obsolète. Utiliser un simple `free ();` ici causerait sûrement une fuite mémoire. Il y a peut-être des allocations internes que nous ne maîtrisons pas.

La méthode à employer pour libérer cette mémoire est de déréférencer notre objet avec la fonction

```
void g_main_loop_unref (GMainLoop *loop);
```

Si le compteur interne passe à 0 alors toute la mémoire allouée est libérée. Pour en finir avec cette boucle principale, trois fonctions importantes permettent de la gérer :

- `void g_main_loop_run (GMainLoop *loop);`. Lance la boucle.
- `void g_main_loop_quit (GMainLoop *loop);`. Arrête la boucle.
- `gboolean g_main_loop_is_running (GMainLoop *loop);`. Teste si la boucle tourne toujours.

2.1 Les sources

Notre boucle doit comporter des sources pour pouvoir être gérer. Sinon, nous ne pourrons plus l'arrêter.

Une source a besoin de différentes fonctions pour être initialisée, vérifiée, et libérée. Elle doit aussi disposer d'une fonction "poll" équivalente à celle de la bibliothèque standard mais portable. Tous les pointeurs de ces fonctions doivent être déclarés dans une structure GSourceFuncs. Voici les noms des éléments à affecter avec leur prototype :

prepare	gboolean (*prepare)(GSource *source, gint *timeout) ;
check	gboolean (*check)(GSource *source) ;
dispatch	gboolean (*dispatch)(GSource *source, GSourceFunc callback, gpointer user_data) ;
finalize	void (*finalize)(GSource *source) ;

Il reste encore à affecter la fonction poll() ; qui attendra une réponse d'un descripteur quelconque. L'affectation de cette fonction se fait avec

```
void g_source_add_poll (GSource *source, GPollFD *fd);
```

La structure GPollFD se compose ainsi :

```
struct GPollFD {
#if defined (G_OS_WIN32) && GLIB_SIZEOF_VOID_P == 8
#endif
#else
  gint                fd;
#endif
  gushort         events;
  gushort         revents;
};
```

Lorsque tout ce petit monde est correctement initialisé il ne reste plus qu'à l'ajouter au contexte de la boucle principale grâce à la fonction

```
/* Cette fonction renvoie l'identifiant de la source
 * dans le contexte. */
guint g_source_attach (GSource *source, GMainContext *context);
```

Ajouter une source au contexte lance la boucle. Cette première description montre la manière la plus bas niveau pour créer une boucle principale avec gestion d'évènements entièrement créés par l'utilisateur. La GLib nous mâche tout de même un peu le travail. Elle met à notre disposition des routines qui permettent d'exécuter des fonctions à intervalle régulier. L'exemple qui suit montre comment mettre en place une boucle principale avec deux sources. Elles sont lancées à intervalle régulier. L'une toutes les demi-seconde, l'autre toutes les quatre secondes. Pour des raisons de commodité et de clarté du code, les deux sources appellent la même fonction callback. Cette fonction affiche le n° de la source et un compteur d'appel associé.

Exemple de boucle principale :

```
1  #include <glib.h>
2
3  gboolean source_func_callback (gpointer user_data)
4  {
5    GSource *source = (GSource*)user_data;
6    static gint count1 = 0;
7    static gint count2 = 0;
8    g_print ("Enter in %s();\n", __func__);
9    g_print ("Source %d\t", g_source_get_id (source));
10
11   switch (g_source_get_id (source)) {
12   case 1:
13     count1++;
14     g_print ("count %d\n", count1);
15     break;
16   case 2:
17     count2++;
18     g_print ("count %d\n", count2);
19   }
20
21    if (count1 > 5) {
22     count1=-1000;
23     return G_SOURCE_REMOVE;
24   }
25   if (count2 > 5) return G_SOURCE_REMOVE;
26
27   return G_SOURCE_CONTINUE;
28 }
29
30 gint main(int argc, gchar **argv)
31 {
32   /* Création d'une boucle principale avec le contexte par défaut */
33   GMainLoop *loop = g_main_loop_new (NULL, FALSE);
34
35   /* Création de deux sources.
36    * La source1 est exécutée toutes les demi-secondes,
37    * la source2 toutes les quatre secondes */
38   GSource *source1 = g_timeout_source_new (500);
39   GSource *source2 = g_timeout_source_new_seconds (4);
40
41   /* Affectation des fonctions appelées aux différentes sources */
42   g_source_set_callback (source1,
```

```
43                          G_SOURCE_FUNC (source_func_callback),
44                          source1,
45                          NULL);
46    g_source_set_callback (source2,
47                          G_SOURCE_FUNC (source_func_callback),
48                          source2,
49                          NULL);
50
51    /* Attachement des deux sources au contexte par défaut */
52    g_source_attach (source1, NULL);
53    g_source_attach (source2, NULL);
54
55    /* Lancement de la boucle principale */
56    g_main_loop_run (loop);
57
58    /* Déréférencement de 1 du pointeur de la boucle principale.
59     * Comme il n'y a ici qu'une référence la mémoire est libérée. */
60    g_main_loop_unref (loop);
61
62    return 0;
63  }
```

Le résultat en console :

```
Enter in source_func_callback ();
Source 1        count 1
Enter in source_func_callback ();
Source 1        count 2
Enter in source_func_callback ();
Source 1        count 3
Enter in source_func_callback ();
Source 1        count 4
Enter in source_func_callback ();
Source 1        count 5
Enter in source_func_callback ();
Source 1        count 6
Enter in source_func_callback ();
Source 2        count 1
```

Seule la source 1 est appelée puis quatre secondes plus tard la source 2 apparaît. En augmentant la valeur d'échappement (fixée ici à 5 ligne 21) nous pouvons vérifier que la source 2 est appelée toutes les quatre secondes. Il est important de noter que le temps n'est pas garanti. Ce ne sont pas des threads, donc ce n'est pas de la programmation multi-tâche. Si une des deux sources met dix secondes à s'exécuter, la boucle est en attente. Les autres sources attendent aussi.

Pour créer les sources les fonctions `g_timeout_source_new (500);` et `g_timeout_source_new_seconds (4);` sont utilisées lignes 38 et 39. Elles créent respectivement une source qui sera appelée toutes les 500 millisecondes et toutes les 4 secondes.

Lignes 42 et 46 la fonction `g_source_set_callback ();` permet d'affecter une fonction callback à chaque source.

```
void
g_source_set_callback (GSource *source,
                       GSourceFunc func,
                       gpointer data,
                       GDestroyNotify notify);
/* source : source pour laquelle on affecte un callback
```

```
 * func   : callback qui sera appelé
 * data   : donnée quelconque qui peut être transmise au callback
 * notify : fonction appelée lors de l'arrêt de la source
 *          (peut être NULL)
 */
```

Le prototype de **GSourceFunc** est
`gboolean (*GSourceFunc) (gpointer user_data);`.

Pour affecter le callback il faut utiliser la macro G_SOURCE_FUNC. C'est un caste nécessaire. Cette méthode est utilisée par toutes les bibliothèques Gtk+.

La ligne 42 transmet en donnée personnelle le pointeur de la source. Ceci permet de le récupérer dans le callback (dernier paramètre du prototype) et ainsi d'afficher son id et faire les traitements nécessaires. Le principe de la donnée personnelle permet de se passer tant que faire ce peut des variables/pointeurs déclarés en global. Non que ce soit interdit mais pour une visibilité et une robustesse du code il faut éviter le plus possible.

Le prototype du callback renvoie un booléen. Tant que la source doit être appelée le callback retourne TRUE. Pour arrêter la source, il suffit de renvoyer FALSE.

L'utilisation de G_SOURCE_REMOVE et G_SOURCE_CONTINUE donnent de la clarté au code source. Ces définitions ne sont que des alias de FALSE et TRUE.

Pour finir à la ligne 60 nous déréférençons le pointeur de la boucle principale. Comme dans notre exemple il n'y a qu'une référence cette fonction va libérer la mémoire allouée.

Il y a d'autres fonctions bien utiles pour ajouter des évènements à une boucle. Leur utilisation est plus spécifique à un environnement fenêtré. Leur étude se fera donc au chapitre Gtk+.

2.2 Les threads

Il est possible d'utiliser les threads avec la GLib. Pour rappel les threads permettent d'avoir des processus fonctionnant en parallèle mais qui ont une mémoire partagée. Ce dernier point pose naturellement des problèmes si plusieurs processus tentent de modifier un espace mémoire identique en même temps. Les mutex, qui permettent d'éliminer ce problème, sont donc bien entendus disponibles aussi. Toutes ces fonctions sont regroupées dans une même section.

Le plus simple pour créer un nouveau thread est l'utilisation de la fonction suivante :

```
GThread *
g_thread_new (const gchar *name,
              GThreadFunc func,
              gpointer data);
/* name : nom du thread
 * func : pointeur de la fonction qui sera lancée comme nouveau thread
 * data : donnée transmise au thread
 */
```

Le prototype de GThreadFunc est

```
/* data est la donnée transmise (son pointeur) lors de la création.*/
gpointer(*GThreadFunc) (gpointer data);
```

Cette fonction de création renvoie un pointeur pour le nouveau thread créé ou bien plante l'application en cas d'erreur. Si nous voulons avoir le contrôle total des erreurs qui peuvent survenir, ce qui est d'ailleurs la chose à faire, il est possible d'utiliser en lieu et place la fonction

```
GThread *
g_thread_try_new (const gchar *name,
                  GThreadFunc func,
                  gpointer data,
                  GError **error);
/* name  : nom du thread
 * func  : pointeur de la fonction qui sera lancée comme nouveau thread
 * data  : donnée transmise au thread
 * error : retourne le code erreur ainsi qu'un texte si une erreur est
           survenue
 */
```

GError permet d'avoir un code et un texte associé en cas d'erreur. Ce qui est bien utile pour afficher ce message en console dans le canal d'erreur. Voila un petit exemple d'utilisation :

```
  GError *error = NULL;
  GThread *thread = g_thread_try_new ("thread",
                                      (GThreadFunc)threadfunc,
                                      "transmitted data",
                                      &error);

  if (!thread) {
    g_printerr ("%s\n", error->message);
    g_error_free (error);
    error = NULL;
  }
```

Le pointeur **error** est initialisé à NULL. Ceci est primordial pour le bon fonctionnement des fonctions associées à ce pointeur. Si après la tentative de création du thread le retour est NULL alors la tentative a échoué. Dans ce cas **error** est alloué et sa structure remplie. Nous affichons en console le message d'erreur. **error** est ensuite libéré avec `g_error_free (error);`. La mise à NULL de ce pointeur ensuite permet de l'utiliser à nouveau dans la même fonction au cas où.

La fonction du thread aura cette forme :

```
gpointer
threadfunc (gpointer data)
{
  g_print ("Lancement du thread avec la donnée personnelle %s\n",
           (gchar*)data);

  return NULL;
}
```

La fonction d'affichage est là pour l'exemple. Elle montre comment récupérer la donnée personnelle transmise. Elle retourne NULL toujours pour l'exemple. On pourra utiliser la fonction
`gpointer g_thread_join (GThread *thread);` pour récupérer cette valeur lorsque le thread est terminé si besoin.

Il existe aussi la fonction `void g_thread_exit (gpointer retval);`. Elle permet de terminer le thread courant dans lequel elle se trouve. Le pointeur **retval** sera la valeur

retournée alors par le thread.

Exemple :

Reprenons les principes de la boucle principale vus précédemment pour y ajouter deux threads. Avec ce code source nous verrons l'utilisation d'une structure personnelle pour permettre de transmettre aux threads des données utiles. Ce principe sera utilisé tout au long de cet ouvrage. C'est la méthode à privilégier pour ne pas avoir de déclaration de variable en globale.

Déclarons notre structure ainsi :

```
typedef struct {
  GThread *thread;
  GMainLoop *loop;
} data_t *memory;
```

Initialisons ensuite une variable avec ce type dans la boucle principale. Nous en profitons aussi pour créer deux threads, une nouvelle boucle principale dans laquelle nous insérons une fonction appelée toutes les 5 ms.

Code complet :

```
1  #include <glib.h>
2
3  typedef struct {
4    GThread *thread1;
5    GThread *thread2;
6    GMainLoop *loop;
7    gint *memory;
8  } data_t;
9
10 gpointer
11 threadfunc1 (gpointer user_data)
12 {
13   data_t *data = (data_t*)user_data;
14
15   /* Accès en écriture de l'espace mémoire partagée sans protection */
16   for (gint i=0; i < 100; i++)
17     {
18       data->memory[i] = 1;
19       g_print ("data->memory[%d] = %d in thread1\n", i, data->memory[i]);
20     }
21
22   return NULL;
23 }
24
25 gpointer
26 threadfunc2 (gpointer user_data)
27 {
28   data_t *data = (data_t*)user_data;
29
30   /* Accès en écriture de l'espace mémoire partagée sans protection */
31   for (gint i=0; i < 100; i++)
32     {
33       data->memory[i] = 2;
```

```
34        g_print ("data->memory[%d] = %d in thread2\n", i, data->memory[i]);
35      }
36
37    return NULL;
38  }
39
40  gboolean
41  source_func_callback (gpointer user_data)
42  {
43    data_t *data = (data_t*) user_data;
44
45    /* Attente de la fin du thread1 */
46    g_thread_join (data->thread1);
47
48    /* Attente de la fin du thread2 */
49    g_thread_join (data->thread2);
50
51    /* Les deux threads sont finis. On arrête la boucle principale */
52    g_main_loop_quit (data->loop);
53
54    return G_SOURCE_REMOVE;
55  }
56
57  gint
58  main(gint argc, gchar *argv[])
59  {
60    data_t data;
61    data.memory = g_new (gint, 100);
62
63    GError *error = NULL;
64    data.thread1 = g_thread_try_new ("thread1",
65                                     (GThreadFunc)threadfunc1,
66                                     &data,
67                                     &error);
68
69    if (!data.thread1) {
70      g_printerr ("%s\n", error->message);
71      g_error_free (error);
72      return 1;
73    }
74
75    data.thread2 = g_thread_try_new ("thread2",
76                                     (GThreadFunc)threadfunc2,
77                                     &data,
78                                     &error);
79
80    if (!data.thread2) {
81      g_printerr ("%s\n", error->message);
82      g_error_free (error);
83      return 1;
84    }
85
86    /* Création d'une boucle principale avec le contexte par défaut */
87    data.loop = g_main_loop_new (NULL, FALSE);
88
```

```
89    /* Création d'une source.
90     * La source est exécutée toutes les 5 millisecondes,
91     * Cette source va permettre de vérifier si le thread est terminé.
92     * Dans l'affirmative la boucle sera arrêtée. */
93    GSource *source = g_timeout_source_new (5);
94
95    /* Ajout d'une fonction à la boucle principale appelée toutes les
96     * 0.5s.
97     * Transmission en donnée personnelle du pointeur du thread créé.
98     * Cette fonction permet d'arrêter la boucle principale lorsque
99     * le thread est fini. */
100   g_source_set_callback (source,
101                          G_SOURCE_FUNC (source_func_callback),
102                          &data,
103                          NULL);
104
105   /* Attachement des deux sources au contexte par défaut */
106   g_source_attach (source, NULL);
107
108   /* Lancement de la boucle principale */
109   g_main_loop_run (data.loop);
110
111   /* Déréférencement de 1 du pointeur de la boucle principale.
112    * Comme il n'y a ici qu'une référence la mémoire est libérée. */
113   g_main_loop_unref (data.loop);
114
115   /* Déréférencement des threads */
116   /* g_thread_unref (thread1); */
117   /* g_thread_unref (thread2); */
118
119   /* Libération de l'espace mémoire alloué dans le tas */
120   g_free (data.memory);
121
122   return 0;
123 }
```

`gpointer threadfunc1 (gpointer data);` et `gpointer threadfunc2 (gpointer data);` sont les fonctions des threads. Elles affichent en console leur affectation. Le comportement semble normal sur ce court exemple qui ne comporte aucune protection d'accès à la mémoire partagée. Mais selon le volume de données traitées et le nombre de threads en jeu il pourrait y avoir un comportement étrange voir planter l'application.
`gboolean source_func_callback (gpointer user_data);` est une source ajoutée à la boucle principale. On lui demande d'être exécutée toutes les 5 ms. Mais dans cette source nous utilisons `g_thread_join();` qui est bloquante. Nous l'utilisons ici pour nous permettre de mettre une condition pour arrêter la boucle principale. L'intérêt ici et de voir que même si la source est bloquée, donc par voie de conséquence la boucle principale aussi, les threads continuent de fonctionner. Nous sommes donc bien en présence d'une programmation parallèle.
Ligne 113 nous déréférençons le pointeur de la boucle principale. Il y a ensuite deux déréférencements pour les threads ligne 116 et 117. Elles sont volontairement commentées. On pourrait penser qu'il est nécessaire, pour ce code exemple, de libérer la mémoire aussi pour les threads. Cependant dans la fonction source nous utilisons `g_thread_join();`. Comme cette fonction attend que le thread se termine, elle en profite avant de nous renvoyer le résultat de le déréférencer. Ainsi la mémoire est déjà libérée.

La sortie en console :

```
data->memory[0] = 1 in thread1
data->memory[1] = 1 in thread1
data->memory[2] = 1 in thread1
data->memory[3] = 1 in thread1
data->memory[0] = 2 in thread2
data->memory[1] = 2 in thread2
data->memory[2] = 2 in thread2
data->memory[3] = 2 in thread2
data->memory[4] = 2 in thread2
data->memory[5] = 2 in thread2
data->memory[6] = 2 in thread2
data->memory[4] = 1 in thread1
data->memory[5] = 1 in thread1
data->memory[6] = 1 in thread1
data->memory[7] = 1 in thread1
data->memory[7] = 2 in thread2
...
```

L'ordre de sortie en console est aléatoire en fonction de chaque lancement de l'application.

2.3 Les mutex

Puisque les threads peuvent manipuler un même espace mémoire il est nécessaire de mettre en place un système de protection d'accès. Les mutex sont là pour ça. Toutes les fonctions associées commencent par `g_mutex...();`.

Pour initialiser un mutex nous utilisons la fonction

```
void
g_mutex_init (GMutex *mutex);
```

Il faudra bien entendu libérer la mémoire allouée de ce mutex lorsque son utilisation ne sera plus nécessaire avec la fonction

```
void
g_mutex_clear (GMutex *mutex);
```

Il ne faut pas libérer un mutex qui est en position bloquant sous peine d'avoir un comportement indéterminé. De même il ne faut pas le libérer s'il a été déclaré en statique.

Nous allons modifier notre code précédent pour mettre en place un mutex de protection d'accès à la mémoire partagée.

```
1  #include <glib.h>
2
3  typedef struct {
4    GThread *thread1;
5    GThread *thread2;
6    GMainLoop *loop;
7    gint *memory;
8    GMutex mutex;
9  } data_t;
10
11 gpointer
12 threadfunc1 (gpointer user_data)
13 {
14   data_t *data = (data_t*)user_data;
15
16   /* Accès en écriture de l'espace mémoire partagée sans protection */
17   for (gint i=0; i < 100; i++)
18     {
19       /* Blocage du mutex. Si le thread2 a déjà pris la main est bloque
20        * l'accès la fonction attend sa libération */
21       g_mutex_lock (&data->mutex);
22       data->memory[i] = 1;
23       g_print ("data->memory[%d] = %d in thread1\n", i, data->memory[i]);
24       g_mutex_unlock (&data->mutex);
25     }
26
27   return NULL;
28 }
29
30 gpointer
31 threadfunc2 (gpointer user_data)
32 {
33   data_t *data = (data_t*)user_data;
34
35   /* Accès en écriture de l'espace mémoire partagée sans protection */
36   for (gint i=0; i < 100; i++)
37     {
38        /* Blocage du mutex. Si le thread1 a déjà pris la main est bloque
39        * l'accès la fonction attend sa libération */
40       g_mutex_lock (&data->mutex);
41       data->memory[i] = 2;
42       g_print ("data->memory[%d] = %d in thread2\n", i, data->memory[i]);
43       g_mutex_unlock (&data->mutex);
44     }
45
46   return NULL;
47 }
48
49 gboolean
50 source_func_callback (gpointer user_data)
51 {
52   data_t *data = (data_t*) user_data;
```

```
53
54    /* Attente de la fin du thread1 */
55    g_thread_join (data->thread1);
56
57    /* Attente de la fin du thread2 */
58    g_thread_join (data->thread2);
59
60    /* Les deux threads sont finis. On arrête la boucle principale */
61    g_main_loop_quit (data->loop);
62
63    return G_SOURCE_REMOVE;
64  }
65
66  gint
67  main(gint argc, gchar *argv[])
68  {
69    data_t data;
70    data.memory = g_new (gint, 100);
71    g_mutex_init (&data.mutex);
72
73    GError *error = NULL;
74    data.thread1 = g_thread_try_new ("thread1",
75                                     (GThreadFunc)threadfunc1,
76                                     &data,
77                                     &error);
78
79    if (!data.thread1) {
80      g_printerr ("%s\n", error->message);
81      g_error_free (error);
82      return 1;
83    }
84
85    data.thread2 = g_thread_try_new ("thread2",
86                                     (GThreadFunc)threadfunc2,
87                                     &data,
88                                     &error);
89
90    if (!data.thread2) {
91      g_printerr ("%s\n", error->message);
92      g_error_free (error);
93      return 1;
94    }
95
96    /* Création d'une boucle principale avec le contexte par défaut */
97    data.loop = g_main_loop_new (NULL, FALSE);
98
99    /* Création d'une source.
100    * La source est exécutée toutes les 5 millisecondes,
101    * Cette source va permettre de vérifier si le thread est terminé.
102    * Dans l'affirmative la boucle sera arrêtée. */
103   GSource *source = g_timeout_source_new (5);
104
105   /* Ajout d'une fonction à la boucle principale appelée toutes les
106    * 0.5s.
107    * Transmission en donnée personnelle du pointeur du thread créé.
```

```
108     * Cette fonction permet d'arrêter la boucle principale lorsque
109     * le thread est fini. */
110    g_source_set_callback (source,
111                           G_SOURCE_FUNC (source_func_callback),
112                           &data,
113                           NULL);
114
115    /* Attachement des deux sources au contexte par défaut */
116    g_source_attach (source, NULL);
117
118    /* Lancement de la boucle principale */
119    g_main_loop_run (data.loop);
120
121    /* Déréférencement de 1 du pointeur de la boucle principale.
122     * Comme il n'y a ici qu'une référence la mémoire est libérée. */
123    g_main_loop_unref (data.loop);
124
125    /* Déréférencement des threads */
126    /* g_thread_unref (thread1); */
127    /* g_thread_unref (thread2); */
128
129    /* Libération de l'espace mémoire alloué dans le tas */
130    g_free (data.memory);
131
132    /* Libération du mutex */
133    g_mutex_clear (&data.mutex);
134
135    return 0;
136  }
```

Ligne 8 un mutex est déclaré en tant que variable. Ligne 71 le mutex est initialisé. La ligne 21 passe le mutex en bloquant pour avoir l'exclusivité de l'accès mémoire. La ligne 24 rend la main pour un autre thread si besoin. L'opération est répétée pour la fonction du thread2. Ainsi plus de problème d'accès mémoire partagée. Les threads terminés nous pouvons libérer l'espace mémoire allouée pour le mutex ligne 133.

Comme `g_mutex_lock();` est bloquante nous pouvons utiliser en lieu est place la fonction `gboolean g_mutex_trylock (GMutex *mutex);` qui renvoie FALSE si le mutex est en position bloquant. Ce qui peut permettre de faire un autre traitement en attendant.

2.4 Les thread pools

Imaginons que nous soyons en train d'écrire un logiciel de traitement d'images astronomiques. Dans ce domaine, pour obtenir les meilleurs rendus il est courant de prendre des heures de photos d'un même objet. Nous nous retrouvons avec des dizaines voir des centaines de photos de ce même objet. L'idée est de toutes les empiler et de faire un traitement pixel par pixel. Une des méthode est de calculer la valeur médiane. Comme aujourd'hui une photo peut facilement faire 80 Mo voir plus, il va être difficile de toutes les charger en même temps en mémoire pour les traiter. Les threads vont venir à notre secours ici.

Une idée serait de charger quelques photos, de les empiler, de sauvegarder la valeur trouvée, puis de passer a un autre groupe et ainsi de suite. Un thread pourrait être lancé pour chaque groupe. Comme le thread lancé va s'arrêter lorsque son traitement est terminé il faudra en lancer un nouveau tant qu'il y a des images à traiter. Cette manière

de procéder, fonctionnelle, pose un problème de performance. Lancer un thread demande du temps processeur, de l'allocation mémoire. Pour améliorer ce fonctionnement il serait préférable de lancer x threads. Chaque thread, lorsque son traitement est terminé ne s'arrête pas. Il charge un nouveau groupe d'images et reprend le travail. Pour gérer ce travail de groupe la GLib met à notre disposition les **GThreadPool**.

Pour initialiser un groupe de thread nous utilisons la fonction

```
GThreadPool *
g_thread_pool_new (GFunc func,
                   gpointer user_data,
                   gint max_threads,
                   gboolean exclusive,
                   GError **error);
/* func        La fonction des threads qui sera exécutée
 * user_data   donnée utile à transmettre aux threads
 * max_threads nombre de threads à lancer
 * exclusive   le groupe de thread doit-il être exclusif ?
 * error       erreur retournée si problème. Peut être mis
               à NULL pour être ignoré
 */
```

La variable **exclusive** indique si tous les threads sont exclusifs à ce groupe ou s'ils sont partagés avec un autre groupe. Si sa valeur est TRUE alors tous les threads sont lancés immédiatement. Dans le cas contraire les threads sont lancés en fonction des besoins.

la fonction à transmettre est du type
`void(*GFunc) (gpointer data, gpointer user_data);`. Nous pourrions nous étonner de ne transmettre qu'une seule fonction. En effet cette fonction sera lancée autant de fois que le nombre de threads demandés. Comme la mémoire est partagée, le code de la fonction aussi.

Combien peut-on lancer de threads ?

Impossible de répondre à cette question. Tout dépend de la longueur du fût du canon ! Disons pour être plus précis que tout dépend de la machine sur laquelle tourne l'application. La GLib nous aide ici en nous fournissant une fonction qui renvoie le nombre maximal approximatif :
`guint g_get_num_processors (void);`. Par exemple sur la machine de votre serviteur équipée d'un microprocesseur AMD Ryzen 7 le nombre renvoyé est 16. Ça tombe bien AMD précise qu'il y a 8 cœurs et 16 threads ;).

En reprenant notre exemple de traitement photo il va arriver un moment où certains threads n'auront plus rien à faire. Il est possible d'en connaître le nombre grâce à la fonction
`guint g_thread_pool_get_num_unused_threads (void);`. Il est même possible de demander de les arrêter avec
`void g_thread_pool_stop_unused_threads (void);`.

Pour envoyer des données la fonction
`gboolean g_thread_pool_move_to_front (GThreadPool *pool, gpointer data);` est faite pour ça. Elle renverra TRUE si la donnée à traiter a bien été prise en compte par un processus.

Bien entendu il faut pouvoir terminer le travail et libérer tout ce petit monde. Nous utiliserons

```
void
g_thread_pool_free (GThreadPool *pool,
                    gboolean immediate,
                    gboolean wait_);
/* pool      le groupe de thread à arrêter
 * immediate Si TRUE alors aucune nouvelle tâche est traitée.
 *           Dans le cas contraire toutes les tâches en cours
 *           sont terminées.
 * wait_     Permet de rendre cette fonction bloquante. Si
 *           vaut FALSE la fonction rend immédiatement la
 *           main. les threads en court feront en fonction de
 *           "immédiate"
 */
```

Cette fonction de libération permet à la fois de s'assurer que les threads sont terminés si **immediate** est fixé à FALSE et de désallouer les threads utilisés. Si on désire ne pas attendre il suffit de placer `wait_` à FALSE. Ceci évite de rester bloqué tant que les threads ne sont pas finis. Une interface fenêtrée utilise une boucle pour écouter des évènements et rafraîchir l'interface si besoin. Si à l'intérieur de cette boucle il y a une fonction bloquante nous nous retrouverons avec une interface inutilisable. Cela donnera l'impression à l'utilisateur que le logiciel a planté.