22nd June 2009

Multi threaded GTK+ applications

It's been a while since I wrote something. And I'm happy to be back (although I didn't go anywhere actually;).

In this post I'll talk about how threads should be used together with GTK+ and try to emphasize things that have a potential to go wrong. I won't talk about mutexes and other thread related stuff that isn't specific to GTK+, so you should find another source of information for those things.

One of the most frequently asked questions about GTK+ is: "Is GTK+ thread safe?". And the most likely answer you'll get to this question is: "No, but it can be made thread aware.". What exactly is thread aware? And how does this awareness helps us when developing application? Let's start from the beginning.

Relaxed introduction

In GTK+ world, everything starts with glib. Glib can be made thread safe simply by calling g_thread_init function at the beginning of our application. And here we have the first possible show stopper: g_thread_init should be called only once. Any subsequent calls will abort with an error. Fortunately for us, there is simple solution for this potential pitfall: call g_thread_init function in construct like this:

```
if( ! g_thread_supported() )
    g_thread_init( NULL );
```

This will prevent calling g_thread_init again it the glib's thread system has been initialized already. And this is all that is needed to be safe when using glib from multiple threads.

GTK+ on the other hand cannot be made completely thread safe. We'll need to take some preventive measures ourselves in order to be able to use GTK+ from multiple threads safely (this is why GTK+ is called thread aware). First thing we should do in our application is call gdk_threads_init function. This will set up gtk's global mutex which can be controlled (locked and unlocked) by calling gdk_threads_enter and gdk_threads_leave functions. From now on, any gtk function call should be enclosed by gdk_threads_enter/gdk_threads_leave, which will ensure that gtk's functions are called from only one thread at a time.

If we put everything that we learned so far into minimalistic example, this is what we get:

```
/* Compile me with:
 *  gcc -o sample1 sample1.c $(pkg-config --cflags --libs gtk+-2.0 gthread-2.0)
 */
#include <gtk/gtk.h>

int
main( int    argc,
      char **argv )
{
    GtkWidget *window;
    GtkWidget *button;

    /* Secure glib */
    if( ! g_thread_supported() )
```

```
        g_thread_init( NULL );

    /* Secure gtk */
    gdk_threads_init();

    /* Obtain gtk's global lock */
    gdk_threads_enter();

    /* Do stuff as usual */
    gtk_init( &argc, &argv );

    window = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    g_signal_connect( G_OBJECT( window ), "destroy",
                      G_CALLBACK( gtk_main_quit ), NULL );

    button = gtk_button_new_with_label( "Initial value" );
    gtk_container_add( GTK_CONTAINER( window ), button );

    gtk_widget_show_all( window );

    gtk_main();

    /* Release gtk's global lock */
    gdk_threads_leave();

    return( 0 );
}
```

Simple enough.

More in-depth hows and whys

Observant readers may have noticed that gtk's global lock is obtained at the start of the application and not released until the end. How is it possible to call gtk functions from other threads then? The secret lies in gtk's main loop, which releases the global lock on each iteration. This gives other threads opportunity to obtain the global lock and execute gtk functions safely.

What about callbacks? Here things become a little complex. Callbacks for gtk signals are made with global mutex locked, so you don't need to surround them with gdk_threads_enter/gdk_threads_leave. Idle and timeout callbacks on the other hand are executed without obtaining global lock, so you're obliged to enclose any gtk calls in those callbacks by gdk_threads_enter/gdk_threads_leave. And since this may become annoying, gtk offers variants of g_idle_add and g_timeout_add that obtain the lock automatically: gdk_treads_add_idle and gdk_threads_add_timeout.

Next code snippet is simple upgrade from previous one and displays how callbacks should be treated.

```
/* Compile me with:
 *  gcc -o sample2 sample2.c $(pkg-config --cflags --libs gtk+-2.0 gthread-2.0)
 */
#include <gtk/gtk.h>

static void
```

```
cb_clicked( GtkButton *button,
            gpointer   data )
{
    /* No need to call gdk_threads_enter/gdk_threads_leave,
       since gtk callbacks are executed withing main lock. */
    gtk_button_set_label( button, "Clicked" );
}

static gboolean
cb_idle( gpointer data )
{
    /* Idle callback don't automatically obtain main lock,
       so we need to do it manually. */
    gdk_threads_enter();
    gtk_button_set_label( GTK_BUTTON( data ), "Idle" );
    gdk_threads_leave();

    return( FALSE );
}

static gboolean
cb_timeout( gpointer data )
{
    /* Timeouts also don't automatically obtain main lock, but
       since we added this one using gdk_threads_add_timeout,
       lock has been obtained for us. */
    gtk_button_set_label( GTK_BUTTON( data ), "Timeout" );

    return( TRUE );
}

int
main( int    argc,
      char **argv )
{
    GtkWidget *window;
    GtkWidget *button;

    /* Secure glib */
    if( ! g_thread_supported() )
        g_thread_init( NULL );

    /* Secure gtk */
    gdk_threads_init();

    /* Obtain gtk's global lock */
    gdk_threads_enter();

    /* Do stuff as usual */
    gtk_init( &argc, &argv );
```

```
    window = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    g_signal_connect( G_OBJECT( window ), "destroy",
                      G_CALLBACK( gtk_main_quit ), NULL );

    button = gtk_button_new_with_label( "Initial value" );
    g_signal_connect( G_OBJECT( button ), "clicked",
                      G_CALLBACK( cb_clicked ), NULL );
    gtk_container_add( GTK_CONTAINER( window ), button );

    g_idle_add( cb_idle, (gpointer)button );
    gdk_threads_add_timeout( 2300, cb_timeout, (gpointer)button );

    gtk_widget_show_all( window );

    gtk_main();

    /* Release gtk's global lock */
    gdk_threads_leave();

    return( 0 );
}
```

Still relatively simple.

Adding thread

Have you noticed that we're talking about multi threaded applications and all of my samples have been single threaded? It's time to become double threaded.

I again modified my sample code, which now sports second thread, which changes button's label every 3 seconds. Additionally, we have a timeout function that changes button label every 2300 milliseconds and a possibility of manually changing button label by clicking on it.

```
/* Compile me with:
 *  gcc -o sample3 sample3.c $(pkg-config --cflags --libs gtk+-2.0 gthread-2.0)
 */
#include <gtk/gtk.h>

static gpointer
thread_func( gpointer data )
{
    while( TRUE )
    {
        sleep( 3 );

        gdk_threads_enter();
        gtk_button_set_label( GTK_BUTTON( data ), "Thread" );
        gdk_threads_leave();
    }

    return( NULL );
}
```

```
static void
cb_clicked( GtkButton *button,
            gpointer   data )
{
    /* No need to call gdk_threads_enter/gdk_threads_leave,
       since gtk callbacks are executed withing main lock. */
    gtk_button_set_label( button, "Clicked" );
}

static gboolean
cb_timeout( gpointer data )
{
    /* Timeouts also don't automatically obtain main lock, but
       since we added this one using gdk_threads_add_timeout,
       lock has been obtained for us. */
    gtk_button_set_label( GTK_BUTTON( data ), "Timeout" );

    return( TRUE );
}

int
main( int    argc,
      char **argv )
{
    GtkWidget *window;
    GtkWidget *button;
    GThread   *thread;
    GError    *error = NULL;

    /* Secure glib */
    if( ! g_thread_supported() )
        g_thread_init( NULL );

    /* Secure gtk */
    gdk_threads_init();

    /* Obtain gtk's global lock */
    gdk_threads_enter();

    /* Do stuff as usual */
    gtk_init( &argc, &argv );

    window = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    g_signal_connect( G_OBJECT( window ), "destroy",
                      G_CALLBACK( gtk_main_quit ), NULL );

    button = gtk_button_new_with_label( "Initial value" );
    g_signal_connect( G_OBJECT( button ), "clicked",
                      G_CALLBACK( cb_clicked ), NULL );
    gtk_container_add( GTK_CONTAINER( window ), button );
```

```
    gdk_threads_add_timeout( 2300, cb_timeout, (gpointer)button );

    /* Create new thread */
    thread = g_thread_create( thread_func, (gpointer)button,
                              FALSE, &error );
    if( ! thread )
    {
        g_print( "Error: %s\n", error->message );
        return( -1 );
    }

    gtk_widget_show_all( window );

    gtk_main();

    /* Release gtk's global lock */
    gdk_threads_leave();

    return( 0 );
}
```

Do I hear someone complaining that button should be protected by a mutex? You're right, it should be. And it actually is protected - by gtk's global mutex. How? We access button only through gtk function calls, never directly. And since gtk functions are allowed to be called only from one thread at a time, our button is automatically being modified only from single thread at a time. (BTW, if you access some parts of the widgets directly, you should really try to replace those direct accesses with accessor function calls. This will make you code more robust across version changes.)

Real world deployment

There are two quite distinct ways of writing multi threaded gtk applications. One way is to protect all gtk calls with main lock as I've been doing in my previous examples, second way of doing it is to call gtk only from single thread. Which method to use? Probably the right answer is "The one that suits you best", but I would recommend the second method. Facts that attribute to this decision can be roughly summoned as:

- No need to call gdk_threads_enter/leave.
- Works both on Linux and Windows.
- Data is separated from it's displayed form (this is a simple form of Model-View-Controller design)

So why did I bother to show you the first way and then marked it as inferior? Because this knowledge will enable you to make an educated choice (or at least an educated guess;) next time you decide to start coding. Still angry?;)

And for the finish, here is the last variation of my sample code that demonstrates the call-gtk-from-single-thread-only approach in action.

```
/* Compile me with:
 *  gcc -o sample4 sample4.c $(pkg-config --cflags --libs gtk+-2.0 gthread-2.0)
 */
#include <gtk/gtk.h>
```

```
/* Progress variable and it's associated mutex */
gint progress = 0;
G_LOCK_DEFINE_STATIC( progress );


static gpointer
thread_func( gpointer data )
{
    gint i;

    for( i = 0; i < 1000; i++ )
    {
        g_usleep( 10000 );

        G_LOCK( progress );
        progress = i;
        G_UNLOCK( progress );
    }

    return( NULL );
}

static gboolean
cb_timeout( gpointer data )
{
    gchar *label;

    G_LOCK( progress );
    label = g_strdup_printf( "Finished %d of 999", progress );
    G_UNLOCK( progress );

    gtk_button_set_label( GTK_BUTTON( data ), label );
    g_free( label );

    return( TRUE );
}

int
main( int    argc,
      char **argv )
{
    GtkWidget *window;
    GtkWidget *button;
    GThread   *thread;
    GError    *error = NULL;

    /* Secure glib */
    if( ! g_thread_supported() )
        g_thread_init( NULL );

    /* Do stuff as usual */
```

```
    gtk_init( &argc, &argv );

    window = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    g_signal_connect( G_OBJECT( window ), "destroy",
                      G_CALLBACK( gtk_main_quit ), NULL );

    button = gtk_button_new_with_label( "Initial value" );
    gtk_container_add( GTK_CONTAINER( window ), button );

    gdk_threads_add_timeout( 100, cb_timeout, (gpointer)button );

    /* Create new thread */
    thread = g_thread_create( thread_func, (gpointer)button,
                              FALSE, &error );
    if( ! thread )
    {
        g_print( "Error: %s\n", error->message );
        return( -1 );
    }

    gtk_widget_show_all( window );
    gtk_main();

    return( 0 );
}
```

That's it. I hope you enjoyed and learned something new. Until next time, stay healthy.

Bye.

Posted 22nd June 2009 by Tadej Borovšak

Labels: GTK+, threads, tutorial

12 View comments

sabri challouf 30 October, 2009 14:18

Many thinks

Reply

Saiph 18 November, 2009 01:25

Dude, thanks, this tutorial is freaking awesome!
I've been following you since you started with the glade things, I love your blog! Please keep posting!

Reply

Anonymous 04 August, 2010 12:06

Very nice explaination, and a nice web site style... good job!

Reply

Robert 20 September, 2010 21:28

Dude, i just love you for that . not tested yet but ive no doubt it surely works perfectly.
Finally ican move my apps to threads and get rid of that nasty redraw evetns in my time important routines.

Keep it up.

Reply

Anonymous 22 September, 2010 16:46

Thanks for this article it helps me very much

Reply

Debmalya Sinha - দেবমাল্য সিংহ 17 December, 2010 01:23

thank you soO much Tadej..
i was banging my head for *just* that.

thanks a lot.

Reply

Debmalya Sinha - দেবমাল্য সিংহ 17 December, 2010 01:24

Tjank you soO much tadej. =)

Reply

Benoît 17 December, 2010 11:14

I would join Saiph to say this tutorial is freaking awesome! Thanks

Reply

Farooq 18 December, 2010 10:56

Thanks a lot, I was searching for this.
Thanks a lot again

Reply

Anonymous 13 January, 2011 15:27

Very clear and easy to understand, thanks!

Reply

Frank 31 January, 2011 14:36

I'm having problems implementing this in a GUI I've made with Glade:

typedef struct Data data;
typedef struct {
GtkWidget *t1view,
*t2view, *t2button;

```
} Data;

GtkTextBuffer *textbuffer1, *textbuffer2;

// Function prototypes:
void on_mainwindow_destroy (GtkObject *, gpointer);
void on_t1button_clicked (GtkButton *, Data *);
void on_t2button_clicked (GtkButton *, Data *);

int main (int argc, char *argv[]) {
GtkBuilder *builder;
GtkWidget *window;
Data *data;

if( ! g_thread_supported() )
g_thread_init( NULL );
gdk_threads_init();
gtk_init (&argc, &argv);


builder = gtk_builder_new ();
gtk_builder_add_from_file (builder,
"tviewgui.glade", NULL);
window = GTK_WIDGET (gtk_builder_get_object
(builder, "window"));

data = g_slice_new(Data);
data->t1view =
GTK_WIDGET(gtk_builder_get_object(builder,
"t1view"));
data->t2view =
GTK_WIDGET(gtk_builder_get_object(builder,
"t2view"));
data->t2button =
GTK_WIDGET(gtk_builder_get_object(builder,
"t2button"));

textbuffer1 = gtk_text_view_get_buffer(GTK_TEXT_VIEW(data->t1view));
textbuffer2 = gtk_text_view_get_buffer(GTK_TEXT_VIEW(data->t2view));

gtk_builder_connect_signals (builder,
data);
g_object_unref (G_OBJECT (builder));

gtk_widget_show (window);
gtk_main ();

return 0;
}

void on_window_destroy (GtkObject *object, gpointer user_data) {
gtk_main_quit();
}

void on_t1button_clicked (GtkButton *t1button, Data *data) {
GtkTextIter ei;
gtk_text_buffer_get_end_iter(textbuffer1, &ei);
gtk_text_buffer_insert(textbuffer1, &ei, "Text
1...\n", -1);
int i;
for (i=0 ; i<5 ; i++) {
```

```
gtk_text_buffer_insert(textbuffer1,
&ei, "Text 1...\n", -1);
while (g_main_context_iteration(NULL,
FALSE));
sleep(2);
}
}

void on_t2button_clicked (GtkButton *t2button, Data *data) {
GtkTextIter ei; //
gtk_text_buffer_get_end_iter(textbuffer2,
&ei);
gtk_text_buffer_insert(textbuffer2, &ei, "Text
2...\n", -1);
}
```

If I want a thread to start when I click t1button (on_t1button_clicked), and to keep running while I do other studd (f.ex. clicking t2button) - where and how do I start the thread?

Reply

jackyalcine 25 February, 2011 16:28

Do you think you can implement an example in Gtkmm? I can try, but I'm not sure how to go about this.

Reply

Enter your comment...

Comment as: Google Accou ▼

Publish Preview