16th July 2009

# Spawning processes using glib

Hello.

It's been almost a month since my last post, but here I am with a new post. And I'll be talking about spawning child processes from callback functions and how to display their output in our main GUI.

## Spawning process

If we want to spawn child process and avoid freezing our GUI, we need to do this asynchronously, which means that we don't wait for child process to return before we resume with our work.

Glib offers quite a few functions that help us with this task, but we'll focus on most powerful one: g_spawn_async_with_pipes [http://library.gnome.org/devel/glib/stable/glib-Spawning-Processes.html#g-spawn-async-with-pipes] . If you look at the API documentation, you can see that this function takes quite a few arguments. Meaning of each argument is nicely described in official docs, so I won't duplicate the effort here. Inspect this invocation from our sample code and try to determine what exactly each parameter does.

```
    ret = g_spawn_async_with_pipes( NULL, argv, NULL,
                                    G_SPAWN_DO_NOT_REAP_CHILD, NULL,
                                    NULL, &pid, NULL, &out, &err, NULL );
    if( ! ret )
    {
        g_error( "SPAWN FAILED" );
        return;
    }

    /* Add watch function to catch termination of the process. This function
     * will clean any remnants of process. */
    g_child_watch_add( pid, (GChildWatchFunc)cb_child_watch, data );
```

## Reading output

Now we would like to read whatever our spawned process outputs. First step towards this was made when we spawned process and specified locations for standard output and error file descriptors. We can now read data directly from those file descriptors using standard functions like read, but those functions would block the application flow until something is sent down the pipe.

To avoid that, we'll use GIOChannels [http://library.gnome.org/devel/glib/stable/glib-IO-Channels.html] and only read data from file descriptor on demand. The following code snippet shows how channels should be created in platform-independent way.

```
    /* Create channels that will be used to read data from pipes. */
#ifdef G_OS_WIN32
    out_ch = g_io_channel_win32_new_fd( out );
    err_ch = g_io_channel_win32_new_fd( err );
#else
    out_ch = g_io_channel_unix_new( out );
```

```
    err_ch = g_io_channel_unix_new( err );
#endif

    /* Add watches to channels */
    g_io_add_watch( out_ch, G_IO_IN | G_IO_HUP, (GIOFunc)cb_out_watch, data );
    g_io_add_watch( err_ch, G_IO_IN | G_IO_HUP, (GIOFunc)cb_err_watch, data );
```

What exactly those watches do? We instruct glib to monitor our channel and inform us when either some data has been sent down the pipe or pipe has been broken.

Putting it all together

To show you how things are set-up in real world application, I created a sample application that reads output from child process and displays it inside GtkTextView. Feel free to experiment.

And this is it for today. Stay healthy and keep of the sun. Bye.

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

typedef struct _Data Data;
struct _Data
{
    /* Buffers that will display output */
    GtkTextBuffer *out;
    GtkTextBuffer *err;

    /* Progress bar that will be updated */
    GtkProgressBar *progress;

    /* Timeout source id */
    gint timeout_id;
};

static void
cb_child_watch( GPid  pid,
                gint  status,
                Data *data )
{
    /* Remove timeout callback */
    g_source_remove( data->timeout_id );

    /* Close pid */
    g_spawn_close_pid( pid );
}

static gboolean
cb_out_watch( GIOChannel   *channel,
```

```
              GIOCondition  cond,
              Data         *data )
{
    gchar *string;
    gsize  size;

    if( cond == G_IO_HUP )
    {
        g_io_channel_unref( channel );
        return( FALSE );
    }

    g_io_channel_read_line( channel, &string, &size, NULL, NULL );
    gtk_text_buffer_insert_at_cursor( data->out, string, -1 );
    g_free( string );

    return( TRUE );
}

static gboolean
cb_err_watch( GIOChannel   *channel,
              GIOCondition  cond,
              Data         *data )
{
    gchar *string;
    gsize  size;

    if( cond == G_IO_HUP )
    {
        g_io_channel_unref( channel );
        return( FALSE );
    }

    g_io_channel_read_line( channel, &string, &size, NULL, NULL );
    gtk_text_buffer_insert_at_cursor( data->err, string, -1 );
    g_free( string );

    return( TRUE );
}

static gboolean
cb_timeout( Data *data )
{
    /* Bounce progress bar */
    gtk_progress_bar_pulse( data->progress );

    return( TRUE );
}

static void
cb_execute( GtkButton *button,
```

```
            Data      *data )
{
    GPid        pid;
    gchar      *argv[] = { "./helper", NULL };
    gint        out,
                err;
    GIOChannel *out_ch,
               *err_ch;
    gboolean    ret;

    /* Spawn child process */
    ret = g_spawn_async_with_pipes( NULL, argv, NULL,
                                    G_SPAWN_DO_NOT_REAP_CHILD, NULL,
                                    NULL, &pid, NULL, &out, &err, NULL );
    if( ! ret )
    {
        g_error( "SPAWN FAILED" );
        return;
    }

    /* Add watch function to catch termination of the process. This function
     * will clean any remnants of process. */
    g_child_watch_add( pid, (GChildWatchFunc)cb_child_watch, data );

    /* Create channels that will be used to read data from pipes. */
#ifdef G_OS_WIN32
    out_ch = g_io_channel_win32_new_fd( out );
    err_ch = g_io_channel_win32_new_fd( err );
#else
    out_ch = g_io_channel_unix_new( out );
    err_ch = g_io_channel_unix_new( err );
#endif

    /* Add watches to channels */
    g_io_add_watch( out_ch, G_IO_IN | G_IO_HUP, (GIOFunc)cb_out_watch, data );
    g_io_add_watch( err_ch, G_IO_IN | G_IO_HUP, (GIOFunc)cb_err_watch, data );

    /* Install timeout fnction that will move the progress bar */
    data->timeout_id = g_timeout_add( 100, (GSourceFunc)cb_timeout, data );
}

int
main( int    argc,
      char **argv )
{
    GtkWidget *window,
              *table,
              *button,
              *progress,
              *text;
    Data      *data;
```

```
    data = g_slice_new( Data );

    gtk_init( &argc, &argv );

    window = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    gtk_window_set_default_size( GTK_WINDOW( window ), 400, 300 );
    g_signal_connect( G_OBJECT( window ), "destroy",
                      G_CALLBACK( gtk_main_quit ), NULL );

    table = gtk_table_new( 2, 2, FALSE );
    gtk_table_set_row_spacings( GTK_TABLE( table ), 6 );
    gtk_table_set_col_spacings( GTK_TABLE( table ), 6 );
    gtk_container_add( GTK_CONTAINER( window ), table );

    button = gtk_button_new_from_stock( GTK_STOCK_EXECUTE );
    g_signal_connect( G_OBJECT( button ), "clicked",
                      G_CALLBACK( cb_execute ), data );
    gtk_table_attach( GTK_TABLE( table ), button, 0, 1, 0, 1,
                      GTK_FILL, GTK_SHRINK | GTK_FILL, 0, 0 );

    progress = gtk_progress_bar_new();
    data->progress = GTK_PROGRESS_BAR( progress );
    gtk_table_attach( GTK_TABLE( table ), progress, 1, 2, 0, 1,
                      GTK_FILL, GTK_SHRINK | GTK_FILL, 0, 0 );

    text = gtk_text_view_new();
    data->out = gtk_text_view_get_buffer( GTK_TEXT_VIEW( text ) );
    gtk_table_attach( GTK_TABLE( table ), text, 0, 1, 1, 2,
                      GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL, 0, 0 );

    text = gtk_text_view_new();
    data->err = gtk_text_view_get_buffer( GTK_TEXT_VIEW( text ) );
    gtk_table_attach( GTK_TABLE( table ), text, 1, 2, 1, 2,
                      GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL, 0, 0 );

    gtk_widget_show_all( window );

    gtk_main();

    g_slice_free( Data, data );

    return( 0 );
}



/*
 * Compile me with:
 *   gcc -o helper helper.c
 */
```

```
#include

int
main( int    argc,
      char **argv )
{
    int i;

    for( i = 0; i < 10; i++ )
    {
        char stdout_string[] = "Normal message no:  .";
        char stderr_string[] = "Error message no:  .";

        stdout_string[19] = '0' + i;
        stderr_string[18] = '0' + i;

        sleep( 1 );
        fprintf( stdout, "%s\n", stdout_string );
        sleep( 1 );
        fprintf( stderr, "%s\n", stderr_string );
    }

    return( 0 );
}
```

Posted 16th July 2009 by Tadej Borovšak

Labels: glib, GTK+

4 View comments

**swilmet** 09 September, 2009 19:05

Hello,
thank you for the example, it was very useful for me :)

A little remark, if you want to run a command which outputs a lot of lines in a short time (latex, pdflatex for examples ;), we must add this code after inserting the line into the buffer:

while (gtk_events_pending ())
gtk_main_iteration ();

Without that, the result is ugly and the lines are not inserted directly.

I've encountered a strange problem with my program: cb_out_watch () is called several times (~20 times) after the call of cb_child_watch (). Maybe cb_out_watch () run too slowly and then when the command is finished, all the cb_out_watch () are not still executed.

So what I've done is to delete cb_child_watch () and the flag G_SPAWN_DO_NOT_REAP_CHILD. And I moved the instructions of cb_child_watch () into the "if( cond == G_IO_HUP )". And it works!

But maybe is there a best solution?

Sébastien

Reply

**swilmet** 10 September, 2009 01:02

I think I've found a better solution.
In the beginning of cb_child_watch () we add:

while (cb_out_watch () || cb_err_watch ());

I will test that tomorrow.

Reply

**onelineproof** 25 June, 2011 09:24

Hi. I know this is an old post, but I'm using it and it is very useful. However, I would like to know something:
Is stdout supposed to update line by line? stderr updates line by line for me, but stdout only updates all 10 lines at once. Not sure why...

Reply

**Anonymous** 15 March, 2014 23:37

That's probably because stdout is buffered, while stderr is not.

Reply

Enter your comment...

Comment as: Google Accou ▼

Publish Preview