

Multithread programming

Multithread programming in GLib/GTK+ applications

Carlos García Campos
carlosgc@gnome.org



GNOMETM

Introduction

- Concurrent programming based on threads
- Thread: independent execution flow inside a process.
- Multiple threads can run simultaneously (or pseudo-simultaneously)
- Creating a new thread is lighter than creating a new process



Advantages

- Avoiding blocks in applications that do I/O operations
- Improving feedback in GUI applications
- Communication among threads in the same process is quite easy and fast
- Taking advantage of the hardware parallelism in SMP systems



Disadvantages

- Concurrent programming is hard
- Non determinism execution
- Debugging is more difficult
- Typical concurrent issues: dead blocks, performance, and so on



¿Why alternatives?

- Disadvantages might be enough worth not to use threads
- General advice: use threads only when it's absolutely needed
- Many situations can be addressed in an easy way without using threads



Alternatives

- Use of `g_idle_add` functions to quickly finish a callback
- Explicitly dispatch pending work in the main loop
- Use of async API when it's available: `gnome_vfs_async_*`, `g_spawn_async_*`, etc.



Processes

- Processes may be a good alternative to threads in many situations
- Processes are independent execution units containing their own state information and their own address spaces.
- IPC mechanisms (managed by the OS) are required to communicate processes with each other
- GLib provides a convenient API for creating new processes (typically `fork + exec`) called GSpawn



GSpawn

- `g_spawn_async_with_pipes`

```
gboolean g_spawn_async_with_pipes (const gchar      *working_directory,
                                   gchar             **argv,
                                   gchar             **envp,
                                   GSpawnFlags       flags,
                                   GSpawnChildSetupFunc child_setup,
                                   gpointer          user_data,
                                   GPid              *child_pid,
                                   gint              *standard_input,
                                   gint              *standard_output,
                                   gint              *standard_error,
                                   GError           **error);
```

- `g_spawn_async`: it just calls `g_spawn_async_with_pipes` without any pipe. There's also a synchronous version where standard output/error and return value are returned by arguments
- `g_spawn_command_line_async`: simple version of `g_spawn_async` which automatically parses the command line. There's a synchronous version too.



GLib Threads

- GLib provides an API that, in UNIX systems, is a wrapper of `p_thread` (POSIX threads)
- Threads management functions:
 - `g_thread_create`
 - `g_thread_create_full`
 - `g_thread_exit`
 - `g_thread_join`
 -
- Concurrency functions:
 - `g_mutex_*`
 - `g_cond_*`



Multithread programming in GLib/GTK+

- GLib is thread-safety when `g_thread_init` is used
- GDK provides `gdk_threads_init` too that has to be used in addition to `g_thread_init`, but it's not always recommendable
- Two options:
 - Using `gdk_threads_init`, `gdk_threads_enter()` and `gdk_threads_leave`
 - Accessing the program GUI exclusively from the main thread (where the gtk main loop is running)
- The second options is preferable in most of the cases



Threads communication

- Main thread
- One or more threads that perform the hard work
- `g_idle` or `g_timeout` functions to send tasks to the main thread from others



Design Patterns

- Boss/Workers: tasks are assigned by the boss to the workers. There are several variations of this model
- Pipeline model: a task is split into several steps so that there is a thread for every step



Debugging

Xlib: unexpected async reply
(sequence 0x1421)!



¿What to do?

- Look for X error handlers in the intermediate layers
 - Bonobo: `bonobo_x_error_handler`
 - GDK: `gdk_x_error`
 - Cairo: `_noop_error_handler`
- Set a break point in gdb for every handler
- Run the program in gdb with the `-sync` modifier (so that the X window calls will be synchronous)
- Once you are stopped in the break point you can just get a backtrace like if it were a crash
- In order to get a backtrace for every running thread use the following command in gdb: **`thread apply all bt`**

