GTTask

Developing asynchronous applications for multi-core efficiency

February 2009
SCALE 7x
Los Angeles

Christian Hergert
What Is It?

GTask is a “mini-framework” to help you write asynchronous code.
Dependencies

GLib and GObject
Platforms

Linux, BSD, and OpenSolaris have been tested

x86, x86_64, and ARM

Probably others
The Application Stack

- C
- Python
- Vala
- JS
- C#
- C++

Your Application

Configuration
- GConf

User Interface
- GTK+

Multimedia
- GStreamer

IO
- GIO

Multi-Core
- GTask

Portability and Object Orientation
- GLib / GObject
#include <glib.h>

Useful routines for C
Portable fundamental types (gint, gfloat, gint64, ...)
Portable atomic operations (g_atomic_int_inc, ...)
HashTable, Queue, Lists, Trees
Locks and Conditions
Errors (like exceptions)
UTF-8, various encodings, and conversions
etc ...
#include <glib-object.h>

Object oriented programming for C
Dynamic type system
Objects, Interfaces, Polymorphism
Properties
Signals (aka, “events”)
Closures
Automatic API bindings to compiled and interpreted languages (Java, Python, Ruby, JS, C++, C#, and more)
Memory management with reference counting
GTask
Abstracts closures one more level
Single shot execution (only executed once)
Ability to respond to execution results/errors
Using callback chains, you can emulate features like try/catch/finally
Cancellation of tasks
Task dependencies to prevent premature execution
Custom schedulers for the problem domain
I Think I've Seen This Before?

Python Twisted
Microsoft's CCR
Apple's NSOperation (OS X 10.5)
Intel's Threading Building Blocks

GTask is probably best described as a blend of Python Twisted and NSOperation

Unlike twisted, you do not need radical changes to your application
Also,

Removes concept of the Stack
The Basics

1. Create a task
   g_task_new (...);

2. Add a callback
   g_task_add_callback (...);

3. Schedule the task
   g_task_scheduler_schedule (...);
Task Phases

WAITING – Waiting on dependent tasks
READY – Ready for execution by scheduler
EXECUTING – Currently executing by scheduler
CALLBACKS – Handling post execution callbacks/errbacks
FINISHED – Execution and callbacks have completed
CANCELED – Task was canceled during execution
Callbacks and Errbacks

[Diagram showing the concept of callbacks and errbacks in a task structure.]

1. Task
   - Callback
   - Errback (Task Handler 1)
2. Task
   - Callback
   - Errback (Task Handler 2)
3. Task
   - Callback
   - Errback (Task Handler 3)
/* Task Execution Method */
typedef GTask* (*GTaskFunc) (GTask *task, GValue *result, gpointer data);

/* Callback (After Successful Execution) */
typedef GTask* (*GTaskCallback) (GTask *task, GValue *result, gpointer data);

/* Errback (After Errored Execution or Callback) */
typedef GTask* (*GTaskErrback) (GTask *task, const GError *error, GValue *result, gpointer data);
Our First Task

#include <gtask/gtask.h>
#define SCHEDULER (g_task_scheduler_get_default ())

static void
do_something (void)
{
    GTask *task;

    task = g_task_new (lsdir_cb, "/usr/bin", NULL);
    g_task_scheduler_schedule (SCHEDULER, task);

    g_object_unref (task);
}
```c
#include <gtask/gtask.h>
#define SCHEDULER (g_task_scheduler_get_default ())

static void
do_something (void)
{
    GTask *task;

    task = g_task_new (lsdir_cb, "/usr/bin", NULL);
g_task_scheduler_schedule (SCHEDULER, task);
    g_object_unref (task);
}
```
```c
#include <gtask/gtask.h>
#define SCHEDULER (g_task_scheduler_get_default ())

static void
do_something (void)
{
    GTask *task;

    task = g_task_new (lsdir_cb, "/usr/bin", NULL);
    g_task_scheduler_scheduler_schedule (SCHEDULER, task);

    g_object_unref (task);
}
```
```c
#include <gtask/gtask.h>
#define SCHEDULER (g_task_scheduler_get_default ())

static void
do_something (void)
{
    GTask *task;

    task = g_task_new (lsdir_cb, "/usr/bin", NULL);
    g_task_scheduler_schedule (SCHEDULER, task);
    g_object_unref (task);
}
```
static void
lsdir_cb (GTask    *task,
         GValue   *result,
         gpointer user_data)
{
    const gchar *path = user_data;
    const gchar *name;
    GDir        *dir;

    dir = g_dir_open (path, 0, NULL);
    while ((name = g_dir_read_name (dir) != NULL)
        g_print ("%s\n", name);
}
Remember the Result Parameter?

Each prototype has access to current “result”.

Great for passing state between Callbacks and Errbacks.

Feels like “functional” programming.
"Throwing" an Error

GError *error = g_error_new_literal (  
    SOME_ERROR_DOMAIN,  
    SOME_ERROR_CODE,  
    "An error occurred" 
);

/* steal our reference to the error */
g_task_take_error (task, error);

/* Or, instead copy the error */
g_task_set_error (task, error);

g_error_free (error);
“Catching” an Error

```c
static void
resolve_error (GTask        *task,
               const GError   *error,
               GValue         *result,
               gpointer       user_data)
{
    /* unsetting error allows further callbacks */
    g_task_set_error (task, NULL);
}

g_task_add_errback (task, resolve_error, NULL, NULL);
```
Try/Catch/Finally

We can emulate try/catch/finally using Callbacks and Errbacks!

/* the catch */
g_task_add_errback (task, task_errorback1, NULL);

/* the finally */
g_task_add_both (task, task_callback2, task_errback2, NULL);
Scheduling

Provides default scheduler, accessible as 
\texttt{g\_task\_scheduler\_get\_default()}

Default scheduler is very simple

Thread pool growth size defaults to $10 \times \text{N\_CPU}$

If implementing a scheduler, watch out for the bumper-to-bumper effect!
Scheduling (Continued)

Currently working on a Work Stealing Scheduler
Also a tertiary scheduler providing “tagged” tasks to pin to a given cpu (increase cpu cache hits).
Work Stealing Scheduler

Each thread has a local (double sided) queue of work items. If no more work, I'll see if my neighbor thread has work left. “Local” pop of work item occurs from tail. “Steal” pop of work item occurs from head.

Local pops attempt lockless for fast-path, uses lock if failed. Steal requires lock.

... Waiting on scheduler revamp.
Scheduler Revamp

Currently threads are managed by the scheduler.

No real reason for that, its lots of tedious code added to writing custom schedulers.

Often times causes lots of extra threads.

Pull thread management out into global controller.

Schedulers are given threads as needed, based on their MIN, MAX, and DESIRED thread count.
Achieving Higher Concurrency

Use Asynchronous tasks when necessary (and possible).

Try to avoid shared state. This isn't just related to gtask, but in general. The result field is a great place to store what state you need.
Async Tasks

g_task_set_async (task, TRUE);
Tasks that do not finish during g_task_execute().
Best way to achieve higher concurrency.
Good example would be using Async IO from within a task. (GIO, part of Glib)
Task is responsible for moving task to the callbacks phase.
  g_task_set_state (task, G_TASK_CALLBACKS)
Main Dispatch

User interfaces such as gtk+ are not thread safe. Callbacks and Errbacks are performed from within the default main loop.

Requires the use of a Main Loop such as GMainLoop or gtk_main().

This allows you to perform work as tasks, and update the user interface seamlessly within a callback or errback.
Enabling Main Dispatch

g_object_set (g_task_scheduler_get_default (),
    "main-dispatch", TRUE,
    NULL);
Language Bindings

Python and Vala currently supported
.NET (via Mono) soon
JavaScript soon (requires gobject-introspection)
C++ soon, needs someone familiar with gtkmm
Python

Simple wrapper around the GTask library

```python
import gtask, urllib

URL = "http://audidude.com"

task = gtask.Task(lambda: urllib.urlopen(URL).read())

task.add_callback(lambda data: file("/tmp/data","w").write(data))

gtask.schedule(task)
```
Python (continued)

Start by rapidly prototyping your code in Python using Tasks.

Optimize as needed simply by implementing your Task in C and calling from Python.
Vala

Vala provides a modern language on top of GLib and GObject.

Syntax very similar to C#.

Compiles to C, no runtime required.

Generics, Lambdas, Properties, Signals, all supported.

You can write implicitly asynchronous code.
void do_something () yields
{
    var task = new Task (_, => {
        return some_sync_operation ();
    });

    /* execution suspends at task.run(). *
    * result is filled upon completion */
    var result = task.run ();
    stdout.printf ("Result was %s\n", (string)result);
}

Implicitly Asynchronous
The Future

Parallel Constructs (foreach, sort, ...)

Local map/reduce

Work Stealing and Rate-Limiting Schedulers

Auto marshaling of task results (for intra-language task execution)

Integrated Profiling

Debugging helpers for visualizing “virtual stack”
Concurrency Helpers

\[
g_{\text{task\_n\_of}}(\ldots) \\
g_{\text{task\_all\_of}}(\ldots) \\
g_{\text{task\_any\_of}}(\ldots)
\]

Helps solve the problem of needing to finish a set of tasks before processing can continue.

Think about multiple database queries in parallel.
GTTask website: http://chergert.github.com/gtask
GTTask documentation: http://docs.dronelabs.com/gtask
GTTask API Reference: http://docs.dronelabs.com/gtask/api