CGreenlet Manual

Release 1.0

Geert Jansen

Apr 17, 2017

Contents

This is the reference manual for cgreenlet. CGreenlet provides an API to work with coroutines in C and C++.

Contents:

CHAPTER 1

Introduction

CGreenlet provides an API for working with coroutines in C and C++. The API is modeled after the Python *greenlet* API[1].

For some, including me before i made an effort to understand them, coroutines are a mysterious type of function. In The Art of Computer Programming, Donald Knuth introduces coroutines as generalizations of subroutines. Instead of returning to the caller, they can also return to a different coroutine.

That explanation didn't do it for me. However I think coroutines can be relatively easy explained in terms of function call stacks. When you call a function, the called function will return to you. If the function you call, calls a nested function, that nested function will first return to the function that called it, which then may return to you as the original caller. The call history is a stack. Inner functions always pop one level (or "frame") off the stack when they return to their calling function. And calling a nested function will push one level (or "frame") onto the stack. The stack idea is actually not just a visualization, it is how virtually all function call ABIs are implemented.

With the idea of a function call stack in mind, coroutines can be explained as having multiple call stacks next to each other. Each coroutine corresponds to one function call stack. Normal function can only move up and down their call stack. Co-routines however, in addition to moving up and down like regular functions, can also move sideways to other stacks. Moving sideways is called "yielding" or "switching", as opposed to "calling" and "returning" wich moves up and down. Crucially, when moving sideways to another coroutine, the point of switching is remembered. And if one of the other coroutines switches back to the original coroutine, it continues exactly where it was left.

In my view, that is all there is to coroutines. So why are coroutines useful? It turns out that there are a couple of use cases that are ideally suited to being solved with coroutines. Two very important ones are:

1. Producer / consumer patterns

   This happens for example in a scanner / parser. The scanner produces tokens from an input. The parser consumes tokens from the scanner. And both functions need context to remember where they are.

   Without coroutines, it is normal to implement one of the functions as a callback that saves state in some area that is preserved between function calls. However callback programming signifcanlty complicates things because the program execution is no longer sequential. With coroutines, both the producer and the consumer can be implemented as sequential functions, that switch to each other when a token is available (the scanner switches to the parser) or when a token is needed (the parser switches to the scanner).

[1] http://pypi.python.org/pypi/greenlet

2. Multiplexed I/O

   Multiplexing I/O means handling multiple streams of input and output in a single process. Traditionally, this can be done by using non-blocking I/O and select(). This has the drawback that you need to write your application as callbacks again, which greatly complicates things. Another solution is to use threads. Threads can use blocking I/O and can therefore implement sequential program logic. However threads are complicated to get right when they need to access global state. Also threads need to be managed as they can eat up system resources quickly.

   Co-routines are ideal for doing multiplexed I/O. They allow you to write sequential code, without having to deal with the complexities of threads.

CHAPTER 2

C API Reference

Greenlet type

The central data structure in the C API is the `greenlet_t`:

```
typedef void *(*greenet_start_func_t)(void *);

typedef struct
{
    greenlet_t *gr_parent;
    void *gr_stack;
    long gr_stacksize;
    int gr_flags;
    greenlet_start_func_t gr_start;
    /* private members follow */
} greenlet_t;
```

Most functions in the cgreenlet library take a greenlet_t as their first argument.

Creating greenlets

New greenlets are created using the `greenlet_new()` function:

```
greenlet_t *greenlet_new(greenlet_start_func_t start_func,
                         greenlet_t *parent, long stacksize);
```

The *start_func* argument specifies the greenlet's main function. The *parent* argument specifies the greenlet's parent. If *parent* is NULL this creates a greenlet that is a child of the special root greenlet. The *stacksize* argument specifies the size of the stack to allocate. If *stacksize* is 0, this allocates a stack of a platform specific default size.

Switching between greenlets

A greenlet is started with the "greenlet_switch_to()" function:

```
void *greenlet_switch_to(greenlet_t *greenlet, void *arg);
```

The first time this function is called on a greenlet, a new execution context is created on the stack that was allocated by `greenlet_new()`, and that greenlet's *start_func* method is called with *arg* as its argument. The greenlet will now be in the "STARTED" state.

After the greenlet has been started, it can either return from its main routine, or switch to another greenlet. If the greenlet returns, the greenlet is marked as "DEAD", and execution switches to its parent. If the greenlet switches to another greenlet, its execution is paused at the point where the greenlet calls `greenlet_switch_to()`. Should another greenlet switch back into this greenlet, then `greenlet_switch_to()` returns and resumes execution from that point.

Every greenlet (except the root greenlet) has a parent. If a greenlet was created with the *parent* parameter of `greenlet_new()` set to NULL, the greenlet is a child of the root greenlet.

When greenlets start up or switch between each other, `void *` pointers can be passed that allow you to pass arbitrary data.

Root and current greenlets

Each process has a special greenlet called the root greenlet. The root greenlet corresponds to the execution context that has been set up by the Operating System when the process was started. The root greenlet is retrieved using:

```
greenlet_t *greenlet_root(void);
```

Each process also has exactly one current greenlet. The current greenlet is retrieved using:

```
greenlet_t *greenlet_current(void);
```

A greenlet's parent is retrieved using:

```
greenlet_t *greenlet_parent(greenlet_t *greenlet);
```

The only greenlet without a parent is the root greenlet. Calling `greenlet_parent()` for the root greenlet returns NULL.

Greenlet states

The state of greenlet is stored in the `gr_flags` member in the `greenlet_t` structure. It is the bitwise OR of the following values:

```
enum greenlet_flags
{
    GREENLET_STARTED = 0x1,
    GREENLET_DEAD = 0x2
};
```

Every greenlet except the root greenlet starts in an empty state. Once the greenlet has been switched to for the first time, it status will have the GREENLET_STARTED bit set. Once a greenlet's main function has exited, it status will have the GREENLET_DEAD bit set. The root greenlet is always in the GREENLET_STARTED status.

The following two utility functions are provided to retrieve a greenlet's state:

```
int greenlet_isstarted(greenlet_t *greenlet);
int greenlet_isdead(greenlet_t *greenlet);
```

Terminating greenlets

Greenlets can be terminated in two ways. First, a greenlet can be reset. This destroys its execution context but keeps the stack allocated. Switching to the greenlet after it is reset would invoke the greenlet's *start_func* again from the beginning:

```
void greenlet_reset(greenlet_t *greenlet);
```

The second way to terminate a greenlet is to destroy it:

```
void greenlet_destroy(greenlet_t *greenlet);
```

Destroying a greenlet destroys its execution context and also deallocates its stack. The greenlet cannot be used anymore.

Thread and Greenlets

Each thread in a process has its own root greenlet and child greenlets. The cgreenlet library is thread-safe in this respect. Of course you need to make sure that when you are using multiple threads, that the greenlets in different threads are properly synchronized if they access shared data. Within a single thread, synchronization is never required because there is always one and only one active greenlet. This is actually the key difference between threads and greenlets.

Injecting code

You can inject a function into a greenlet by using `greenlet_inject()`:

```
typedef void (*greenlet_inject_func_t)(void *);

void greenlet_inject(greenlet_t *greenlet,
                     greenlet_inject_func_t inject_func);
```

When a function has been injected this way , the injected function will be called exactly once when switching to the greenlet, just before the point where execution would normally start or resume. The argument that is passed to *inject_func* is the one that will have been passed to the greenlet as well. If the injected function returns, the greenlet execution will resume as normal.

Injecting code can be useful in some special error situations or for debugging. It is also used by the C++ greenlet interface to inject exceptions from a child into a parent greenlet.

CHAPTER 3

C++ API Reference

The greenlet class

The greenlet class is a wrapper around the `greenlet_t` structure from the cgreenlet C API. Before reading this section, make sure you have read the C API first.

Two constructors and a destructor are provided for the `greenlet` class:

```
class greenlet
{
public:
    greenlet(greenlet_start_func_t start_func=0L,
                    greenlet *parent=0L, long stacksize=0L);
    greenlet(greenet_t *greenlet);
    ~greenlet();
};
```

The first constructor creates a new `greenlet` instance from scratch. All arguments are optional and have the same measing as in the C API. The second form creates a `greenlet` instance from a `greenlet_t` struct from the C API. This allows C and C++ greenlets to exist in the same program. The destructor peforms the same thing as `greenlet_destory()` in the C API.

Main function

The greenlet's main function is always its virtual `run()` method:

```
protected:
    virtual void *run(void *arg);
```

The default implementation of `run()` executes the *start_func* argument that was passed to the constructor. You can override `run()` to embed your main function in a derived class.

Greenlet methods

Many functions that exist in the C API exist as method on the `greenlet` class in the C++ API:

```
public:
    void *switch_to(void *arg);
    void inject(greenlet_inject_func_t inject_func);
    void reset();

    bool isstarted();
    bool isdead();
```

Exceptions

Exceptions that are not caught in a greenlet will exit the greenlet's main function and move it to the "DEAD" state. When this happens, the exception is then re-raised in the greenlet's parent, and the parent's parent, and so on, until either it is caught or until the root greenlet is reached. If the root greenlet does not catch the exception, the C++ program will terminate.