Ruby_Grant_2014_Report.md

Threading Support for Byebug

Motivation

Having a fully featured stable debugger is important for most programming languages. It makes the language more attractive for beginners and for users coming from other languages, because it's a very adequate tool not only for bug fixing but also just for playing around with a language's features or studying code not written by ourselves. With this in mind, the main purpose of Byebug since it was started was to make it an atractive tool for beginners (I was actually a Ruby beginner during the initial development phase of Byebug so I was making heavy use of my own tool too).

The main features supported by Byebug are:

- Breaking. Pause the program at some event or specified instruction, to examine the current state. Related commands: `break` , `catch` , `condition` , `delete` , `enable` , `disable` .
- Analyzing. Studying program status at a certain point during its execution (including right after termination). Specifically, we can:
  - Inspect and move around the backtrace (`backtrace` , `up` , `down` and `frame` commands).
  - Have a basic REPL functionality, evaluating custom code (`eval` , `irb` , `pry` , `method` , `pp` , `ps` , `putl` , `var` commands).
  - Look and change the program's source code (`edit` , `list` , `info` commands).
- Stepping: Running your program one line or instruction at a time, or until specific points in the program are reached. Related commands: `step` , `next` , `continue` , `finish` , `kill` , `quit` , `restart` .
- Tracking: Keeping track of the different values of your variables or the different lines executed by your program. Related commands: `display` , `undisplay` , `tracevar` , `untracevar` , `set linetrace` .

This features have been working very well as long as the debugged program would have no multiple Ruby threads, but would just not work when the program would use different threads. Notice that this would affect developers making use of threads, but was also affecting users not necessarily knowing anything about threads, because very well know libraries out there transparently make use of them (for example, `capybara-webkit` or Ruby's stdlib `Timeout` module).

So Byebug needed a way to debug multithreaded programs that was both:

- Reliable: no deadlock, no killed threads when they are not related to user's code.
- Useful: allow debugging issues with multithreaded programs. To do that, we would need to provide the user with the ability to stop/resume specific threads, list active threads and switch between threads.

This is what this grant is about.

The feature

The addition of threading support to Byebug's debugger allows users to properly debug programs making use of Ruby's threads. This includes listing active threads and their statuses, switching execution to specific threads and temporarily pausing/resuming threads.

To try out the feature, you might want to use a real application (a Rails app for example) using threads or just follow the sample session about threads included in Byebug's Guide. See here for details.

The feature is also fully tested. You can clone *byebug*'s repo and then run

```
bundle install # Install dependencies
rake compile # Compile the C-extension
ruby -w -Ilib test/test_helper.rb test/commands/thread_test.rb
```

This is the list of available commands and a short explanation of its usage:

- *thread list*: Lists threads. This is equivalent to Ruby's `Thread.list`, but it has the following format:
    - A mark '+' for the current thread.
    - A mark '$' for a stopped thread.
    - An internal `id` for the thread, specific to Byebug.
    - Ruby's id and status for the thread, in the format `#<Thread:0x0123456789ABCD (run|sleep)>`. In Ruby 2.2.x, also the file and line number where the thread is defined are included. This feature is very useful to correctly identify threads, because otherwise the only way to tell which thread is which is from the order they are defined.
    - Current file and line number location of the thread's execution, in the format `file:line`.
- *thread current*: Shows the `thread list` entry for the current thread, just like the `frame` command shows the current frame whereas the `backtrace` command shows the whole

backtrace.

- *thread stop*: Allows the user to temporarily stop the execution of a thread. This is useful when we want to focus on debugging specific threads and want to make sure some other thread stays unchanged, or if we want our main thread to "wait for us" and don't finish until we tell it to.
- *thread resume*: Allows resuming threads previously stopped with `thread stop`. It can be used to resume normal program execution, once we've introduced a change that could fix our issue, for example.
- *thread switch*: Switches current thread and context to another thread. After issuing this command, execution will be stopped in a different place in the source code and we'll get a different backtrace. The target thread can't be in the sleeping state so we might have to issue the `thread resume` command before running this command.

The implementation

The TracePoint API was not well suited for this feature. It includes a `THREAD_BEGIN` and a `THREAD_END` events, but they are generated when the execution is first delegated to the thread and not when the thread is created. We want the threads to be available to the user (in a "sleep" state) from their creation, so we need to resort to "other trickery".

Mantaining a global thread list

Byebug mantains a global hash table of active threads which is constructed dinamically as TracePoint API events are received. Every time an event is processed, we look for a thread matching the current thread in our threads table and we set up that context to be the current context (when we talk about "context" in Byebug we mean the program's state in a specific moment during its execution). If the thread is not found (first event of the thread), we create a new entry in the table. This is done by the `thread_context_lookup` method in `threads.c`.

Periodically, the table is cleaned up of dead threads, using the `cleanup_dead_threads` method in `threads.c`. This method needs to make use of Ruby because the C-extension API does not seem to have utilities to check for thread status. This might be a big performance penalty for programs using a big number of threads, so at some point we might want to either have a `rb_thread_status` method available to C-extensions, or add a workaround inside Byebug such as not cleaning the threads table for every event but only every "N" events, where "N" is big enough so this cleanup is not a performance bottleneck anymore. Nevertheless, I've tried latest byebug with some Rails apps and haven't noticed any performance issues.

Thread syncronization

The biggest challenge of implementing threading in Byebug has been this one. While our user is stopping at his Byebug prompt, the scheduler can (and does) schedule different threads to be run, so TracePoint API events are generated for other threads. We want everything halted while the user is in control so we need to lock the processing of this events until the user gives control back to the debugger. To do that we've used a global lock in the C-extension, that ensures that a single TracePoint API event is processed at the same time.

At the beginning of the processing of every event, we call the `acquire_lock` method that will either:

- Obtain the global lock if it's free (or the current thread already has it because the previous event was also from the same thread). In this case, the event is processed normally.
- Go to sleep and pass execution on to another thread if the lock is currently being hold by another thread. Notice that we need to specifically call `rb_thread_stop` here because C-extensions are not preemptive in the sense that the scheduler won't automatically switch thread execution while in a C-extension just like it does when running "regular Ruby code". So if we don't call `rb_thread_stop` , the execution will just deadlock here.

At the end of the processing of every event, we call the `release_lock` method that will release the lock and pass on the execution to another thread (that will probably be halted in `acquire_lock` and will be able to pass through once the lock is released.

Specifics of some commands

thread list

We currently manually syncronize our thread list with the one given by Ruby (`Thread.list`) when this command is executed, but once this feature is well tested we can probably get rid of that check and just trust our table that should always be up to date and exactly the same as `Thread.list` .

thread stop

To implement this command, we needed to add some global flags. A function `rb_thread_stop` is available for C-extensions to stop the current thread, but when the user issues this command, the target thread is not the current thread so we can't directly use that method as it doesn't accept a target thread argument. Instead, we set a global flag, `CTX_FL_SUSPEND` and check that in `acquire_lock` to prevent thread execution. So even if the global lock is free, we never delegate execution to the suspended thread.

thread resume

The only specific comment about the implementation of this command is the `CTX_FL_WAS_RUNNING` flag. This flag is used to remember the thread's status when a thread was suspended so the `thread` `resume` command can correctly restore it. It `CTX_FL_WAS_RUNNING` is set when we run `thread resume` we need to call `rb_thread_wakeup` to restore the "running" status.

thread switch

This command was a bit problematic. Users will probably expect nothing to be run when they issue this command, just a "context change". However, we actually need to let program's execution to succesfully achieve the context change, so that the new "current thread" is the one we are switching to and we can properly show backtrace and file location information for the new thread.

So the idea here is to force the scheduler to inmediately delegate execution control to the target thread so that the next TracePoint API event generated belongs to that thread and we can inmediately stop execution again without running anything else. To achieve this, `thread switch` does the following:

- Saves the target thread in a global `next_thread` variable.
- Sets a breakpoint for the next event to be receive.
- Releases the user prompt and gives control back to the debugger.

To make this work, we need to change the way we `release_lock` after every event has been processed. We needn't just release the lock but also force the scheduler the give control to the thread specified by `next_thread`. To implement this, we add a double linked list where we mantain the list of threads whose execution is being hold by Byebug's global lock. Threads are added to this list in `acquire_lock` and removed in `release_lock`. In the `release_lock` method we pop `next_thread` from the list if `next_thread` is set or *any* thread otherwise. Then we call `rb_thread_run` on the popped thread to delegate control to that thread.

Byebug's REPL and threads

Byebug's debugger includes a REPL aside from it's built-in commands. Anything that's not recognized as a Byebug command will be automatically evaluated as Ruby code. This has proven to be a very useful feature for users to the point the some people consider Byebug as an `irb` or `pry` alternative.

The new threading feature would not play nice with the REPL when the command to be evaluated included thread stuff. Users would get either a 'No threads alive. deadlock?' error or a proper deadlock. For an example of this issues, have a look at here.

This would happen because `byebug`'s global lock wouldn't be released before evaluating stuff, so if an evaluated command created new threads or switched to previously created threads, we would get a deadlock because those threads wouldn't be able to run because thread execution would be hold by Byebug's current thread.

To solve this issues, we exposed to Ruby a couple of methods `Byebug.lock` and `Byebug.unlock` to would call `acquire_lock` and `release_lock`, and then implemented the following method:

```
def allowing_other_threads
  Byebug.unlock
  res = yield
  Byebug.lock
  res
end
```

and call it before evaluating anything in Byebug's prompt. This solved issues when evaluating stuff from the user's prompt.

The code

After the explanation of the current implementation, I think we've gone through every bit of code relating to threads. I copy the relevant file `threads.c` in the C-extension for completeness.

```
#include <byebug.h>

/* Threads table class */
static VALUE cThreadsTable;

/* If not Qnil, holds the next thread that must be run */
VALUE next_thread = Qnil;

/* To allow thread syncronization, we must stop threads when debugging */
VALUE locker = Qnil;

static int
t_tbl_mark_keyvalue(st_data_t key, st_data_t value, st_data_t tbl)
{
  UNUSED(tbl);

  rb_gc_mark((VALUE) key);

  if (!value)
    return ST_CONTINUE;

  rb_gc_mark((VALUE) value);

  return ST_CONTINUE;
}

static void
t_tbl_mark(void *data)
{
  threads_table_t *t_tbl = (threads_table_t *) data;
  st_table *tbl = t_tbl->tbl;
```

```c
  st_foreach(tbl, t_tbl_mark_keyvalue, (st_data_t) tbl);
}

static void
t_tbl_free(void *data)
{
  threads_table_t *t_tbl = (threads_table_t *) data;

  st_free_table(t_tbl->tbl);
  xfree(t_tbl);
}

/*
*  Creates a numeric hash whose keys are the currently active threads and
*  whose values are their associated contexts.
*/
VALUE
create_threads_table(void)
{
  threads_table_t *t_tbl;

  t_tbl = ALLOC(threads_table_t);
  t_tbl->tbl = st_init_numtable();
  return Data_Wrap_Struct(cThreadsTable, t_tbl_mark, t_tbl_free, t_tbl);
}

/*
*  Checks a single entry in the threads table.
*
*  If it has no associated context or the key doesn't correspond to a living
*  thread, the entry is removed from the thread's list.
*/
static int
check_thread_i(st_data_t key, st_data_t value, st_data_t data)
{
  UNUSED(data);

  if (!value)
    return ST_DELETE;

  if (!is_living_thread((VALUE) key))
    return ST_DELETE;

  return ST_CONTINUE;
}

/*
*  Checks whether a thread is either in the running or sleeping state.
*/
int
is_living_thread(VALUE thread)
```

```
{
  VALUE status = rb_funcall(thread, rb_intern("status"), 0);

  if (NIL_P(status) || status == Qfalse)
    return 0;

  if (rb_str_cmp(status, rb_str_new2("run")) == 0
      || rb_str_cmp(status, rb_str_new2("sleep")) == 0)
    return 1;

  return 0;
}

/*
*  Checks threads table for dead/finished threads.
*/
void
cleanup_dead_threads(void)
{
  threads_table_t *t_tbl;

  Data_Get_Struct(threads, threads_table_t, t_tbl);
  st_foreach(t_tbl->tbl, check_thread_i, 0);
}

/*
* Looks up a context in the threads table. If not present, it creates it.
*/
void
thread_context_lookup(VALUE thread, VALUE * context)
{
  threads_table_t *t_tbl;

  Data_Get_Struct(threads, threads_table_t, t_tbl);

  if (!st_lookup(t_tbl->tbl, thread, context) || !*context)
  {
    *context = context_create(thread);
    st_insert(t_tbl->tbl, thread, *context);
  }
}

/*
* Holds thread execution while another thread is active.
*
* Thanks to this, all threads are "frozen" while the user is typing commands.
*/
void
acquire_lock(debug_context_t * dc)
{
  while ((!NIL_P(locker) && locker != rb_thread_current())
        || CTX_FL_TEST(dc, CTX_FL_SUSPEND))
  {
    add_to_locked(rb_thread_current());
    rb_thread_stop();

    if (CTX_FL_TEST(dc, CTX_FL_SUSPEND))
      CTX_FL_SET(dc, CTX_FL_WAS_RUNNING);
  }

  locker = rb_thread_current();
}

/*
* Releases our global lock and passes execution on to another thread, either
* the thread specified by +next_thread+ or any other thread if +next_thread+
* is nil.
*/
void
release_lock(void)
{
  VALUE thread;

  cleanup_dead_threads();

  locker = Qnil;

  if (NIL_P(next_thread))
    thread = pop_from_locked();
  else
  {
    remove_from_locked(next_thread);
    thread = next_thread;
  }

  if (thread == next_thread)
    next_thread = Qnil;

  if (!NIL_P(thread) && is_living_thread(thread))
    rb_thread_run(thread);
}

/*
*  call-seq:
*    Byebug.unlock -> nil
*
*  Unlocks global switch so other threads can run.
*/
static VALUE
Unlock(VALUE self)
{
  UNUSED(self);
  release_lock();

  return locker;
}

/*
*  call-seq:
*    Byebug.lock -> Thread.current
*
*  Locks global switch to reserve execution to current thread exclusively.
*/
static VALUE
Lock(VALUE self)
{
  debug_context_t *dc;
  VALUE context;

  UNUSED(self);

  if (!is_living_thread(rb_thread_current()))
    rb_raise(rb_eRuntimeError, "Current thread is dead!");

  thread_context_lookup(rb_thread_current(), &context);
  Data_Get_Struct(context, debug_context_t, dc);

  acquire_lock(dc);

  return locker;
}

/*
*
*    Document-class: ThreadsTable
*
*    == Sumary
*
*    Hash table holding currently active threads and their associated contexts
*/
void
Init_threads_table(VALUE mByebug)
{
  cThreadsTable = rb_define_class_under(mByebug, "ThreadsTable", rb_cObject);

  rb_define_module_function(mByebug, "unlock", Unlock, 0);
  rb_define_module_function(mByebug, "lock", Lock, 0);
}
```

Future work

With the tasks performed in this grant, threading support is finished. The next tasks will be to make sure the feature is working fine for our users and fix any issues that might come up.

Regarding Byebug as a whole, the idea is that the next major release will include a full rewrite / review of remote debugging support, and will make sure that editor plugins or graphical debuggers can easily use byebug under the hood.