Debugging of CPython processes with gdb

KharkivPy

January 28th, 2017

by Roman Podoliaka, Development Manager at Mirantis

twitter: @rpodoliaka

blog: http://podoliaka.org

slides: http://podoliaka.org/talks/

Goals of this talk

- make gdb "a known unknown", so that you consider it as an option in the future
- highlight the common gotchas

Why debugging?

- working on a huge open source cloud platform - OpenStack
- dozens of various (micro-)services - from REST APIs to system daemons
- new features are important, but high availability and overall stability are even more important
- continuous functional / performance / scale testing
- numerous customer deployments
- things break... pretty much all the time!

Caveats

The described debugging techniques assume that you use:

- Linux
  - Darwin (macOS): must be similar, but lldb is the debugger of choice there
  - Windows: should work, if you use a recent gdb build with Python support enabled and have debugging symbols for CPython
- CPython 2.7 or 3.x
  - debugging scripts are interpreter-specific, so no PyPy/Jython/IronPython/etc
  - 2.6 works too, but up-to-date scripts are more useful

What's wrong with pdb?

It's a nice and easy to use debugger, that should be your default choice, but it:

- can't attach to a running process
- can't step into native code (e.g. shared libraries, C/C++ extensions or CPython itself)
- can't be used for debugging of interpreter crashes (i.e. core dumps)

Typical problems: hung process

- a process is stuck in `S (sleeping)` state and does not respond
- `strace`'ing shows that it is trying to acquire a lock (i.e. `futex(...)`)
- one needs a way to map this to the exact line in the application code
- especially important if you use cooperative concurrency (i.e. asyncio, eventlet, gevent, etc)

Typical problems: going into native code

- ~14000 unit tests, one or a few create a temporary directory in the git working tree and do not clean up after themselves. How do you identify those?
- pdb does not allow to set breakpoints in built-in functions (like `os.makedirs()`)

Typical problems: interpreter crashes

- rarely happen in common applications
- but still do with things like mod_wsgi (https://github.com/GrahamDumpleton/mod_wsgi/issues/81)
- or calls to native libraries via cffi (https://bitbucket.org/cffi/cffi/issues/240/cffi-crash-on-debian-unstable-with-gcc-5)

gdb

- a general purpose debugger, that is mostly used for debugging of C and C++ applications (supports Objective-C, Pascal, Rust, Go and more)
- allows attaching to a running process without instrumenting it in advance
- allows taking a core dump (a state of process memory at a specific moment of time) in order to analyze it later
- allows post-mortem debugging of core dumps of crashed processes saved by the kernel (if `ulimit` allows for it)
- allows switching between threads

ptrace: the secret power behind gdb and strace

```
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

provides a means by which one process (the *"tracer"*) may observe and control the execution of another process (the *"tracee"*)

Debugging of interpreted languages

- Python code is not compiled into a native binary for a target platform. Instead there is an interpreter (e.g. CPython, the reference implementation of Python), which executes compiled byte-code
- when you attach to a Python process with gdb, you'll debug the interpreter instance and introspect the process state at the interpreter level, not the application level

Debugging of interpreted languages: interpreter level traceback

```
#0  0x00007fcce9b2faf3 in __epoll_wait_nocancel () at ../sysdep
#1  0x0000000000435ef8 in pyepoll_poll (self=0x7fccdf54f240, ar
#2  0x000000000049968d in call_function (oparg=<optimized out>,
#3  PyEval_EvalFrameEx () at ../Python/ceval.c:2666
#4  0x0000000000499ef2 in fast_function () at ../Python/ceval.c
#5  call_function () at ../Python/ceval.c:4041
#6  PyEval_EvalFrameEx () at ../Python/ceval.c:2666
```

Debugging of interpreted languages: application level traceback

```
/usr/local/lib/python2.7/dist-packages/eventlet/greenpool.py:82
    `func(*args, **kwargs)`
/opt/stack/neutron/neutron/agent/l3/agent.py:461 in _process_rc
    `for rp, update in self._queue.each_update_to_next_router()
/opt/stack/neutron/neutron/agent/l3/router_processing_queue.py:
    `next_update = self._queue.get()`
/usr/local/lib/python2.7/dist-packages/eventlet/queue.py:313 in
    `return waiter.wait()`
/usr/local/lib/python2.7/dist-packages/eventlet/queue.py:141 in
    `return get_hub().switch()`
/usr/local/lib/python2.7/dist-packages/eventlet/hubs/hub.py:294
    `return self.greenlet.switch()`
```

PyEval_EvalFrameEx

```
PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    /* variable declaration and initialization stuff */
    for (;;) {
        /* do periodic housekeeping once in a few opcodes */
        opcode = NEXTOP();
        if (HAS_ARG(opcode)) oparg = NEXTARG();
        switch (opcode) {
            case NOP:
                goto fast_next_opcode;
            /* lots of more complex opcode implementations */
            default:
                /* become rather unhappy */
        }
        /* handle exceptions or runtime errors, if any */
    }
    /* we are finished, pop the frame stack */
    tstate->frame = f->f_back;
    return retval;
}
```

gdb and Python

- gdb can be built with Python support enabled
- that essentially means one can extend gdb with Python scripts
    - e.g. pretty-printing for C++ STL containers: https://sourceware.org/gdb/wiki/STLSupport
- the very same mechanism is used for debugging of CPython: https://github.com/python/cpython/blob/master/Tools/gdb/libpython.py

Prerequisites: gdb with Python support

```
apt-get install gdb
```

or

```
yum install gdb
```

or something else depending on the distro you use, then

```
gdb -ex 'python print("ok")' -ex quit | tail -n 1
```

Prerequisites: CPython debugging symbols

- debugging symbols are information on the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code
- generated when applications are compiled with -g flag passed to gcc/clang
- consume a lot of disk space, thus, are usually stripped from compiled binaries and shipped separately
- the most popular and commonly used format of debugging symbols is called DWARF

Prerequisites: CPython debugging symbols

```
apt-get install python-dbg
```

or

```
yum install python-debuginfo
```

CentOS/RHEL put those into a separate repo

```
debuginfo-install python
```

Some distros (like Arch Linux) do not ship debugging symbols at all

Prerequisites: CPython scripts for gdb

- developed in CPython code tree: https://github.com/python/cpython/blob/master/Tools/gdb/libpython.py
- packaged and shipped by Linux distros
- loaded by gdb automatically when debugging python binary
- can also be loaded manually like
    - `(gdb) source ~/src/cpython/Tools/gdb/libpython.py`

Debug a process from the start

```
gdb /usr/bin/python

(gdb) run my_python_script.py
```

Attach to a running process

```
gdb /usr/bin/python -p $PID
```

or simply

```
gdb -p $PID
```

(note: gdb will stop all process threads)

Load the inferior state from a core dump

get a core dump of a running process

```
gcore $PID
```

open it in gdb

```
gdb /usr/bin/python core.$PID
```

Print a traceback

```
(gdb) py-bt
Traceback (most recent call first):

File "/usr/lib/python2.7/logging/__init__.py", line 872, in emi
  stream.write(ufs % msg)
File "/usr/lib/python2.7/logging/__init__.py", line 759, in har
  self.emit(record)
File "/usr/lib/python2.7/logging/__init__.py", line 1336, in ca
  hdlr.handle(record)
File "/usr/lib/python2.7/logging/__init__.py", line 1296, in ha
  self.callHandlers(record)
File "/usr/lib/python2.7/logging/__init__.py", line 1286, in _l
  self.handle(record)
File "/usr/lib/python2.7/logging/__init__.py", line 1155, in de
  self._log(DEBUG, msg, args, **kwargs)
File "/usr/lib/python2.7/logging/__init__.py", line 1440, in de
  self.logger.debug(msg, *args, **kwargs)
File "/opt/stack/nova/nova/compute/resource_tracker.py", line 6
  'pci_devices': pci_devices})
```

Print Python code

```
(gdb) py-list
 867 try:
 868     if (isinstance(msg, unicode) and
 869         getattr(stream, 'encoding', None)):
 870         ufs = u'%s\n'
 871         try:
>872             stream.write(ufs % msg)
 873         except UnicodeEncodeError:
 874             #Printing to terminals sometimes fails. For ex
 875             #with an encoding of 'cp1251', the above write
 876             #work if written to a stream opened or wrapped
 877             #the codecs module, but fail when writing to a
```

Print local variables

```
(gdb) py-locals
self = <ColorHandler(...)>
stream = <file at remote 0x7fa76ebb61e0>
fs = '%s\n'
ufs = u'%s\n'
```

Set a breakpoint in native code

```
gdb /usr/bin/python

(gdb) break mkdir
Breakpoint 1 at 0x417600

(gdb) condition 1 $_regex((char*) $rdi, ".*/instances/.*")

(gdb) commands 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>py-bt
>end
end

(gdb) run -m testtools.run discover -s nova/tests/unit
```

Execute arbitrary Python code in the process context

```
(gdb) call PyGILState_Ensure()
$21 = PyGILState_UNLOCKED
(gdb) call PyRun_SimpleString("print('hello')")
hello
$22 = 0
(gdb) call PyGILState_Release(PyGILState_UNLOCKED)
```

Gotchas: virtual environments and custom CPython builds

- when attaching to a Python process started in a virtual environment debugging symbols *may* suddenly not be found anymore

```
gdb -p $2975
Attaching to process 2975
Reading symbols from .../venv/bin/python2...
(no debugging symbols found)...done.
```

- it happens because gdb looks for them in the *wrong place*: if you omit the *inferior* binary path, gdb tries to derive it from `/proc/$PID/exe` symlink and then load debugging symbols stored in the predefined path - e.g. `/usr/lib/debug/$PATH`. For a virtual environment it's not `/usr/lib/debug/usr/bin/python2`, thus, loading fails

Gotchas: virtual environments and custom CPython builds

- the solution is to *always* pass the inferior binary path explicitly when attaching to a process

```
gdb /usr/bin/python2.7 -p $PID
```

- alternatively, modern CPython builds (at least on Debian Testing or Ubuntu Xenial) have an associated `build-id` value, that is used to uniquely identify stripped debugging symbols

```
objdump -s -j .note.gnu.build-id /usr/bin/python2.7
```

```
Reading symbols from /usr/lib/debug/.build-id/8d/04a3ae38521cb7
```

Gotchas: virtual environments and custom CPython builds

- py- commands may be undefined for a very similar reason

```
(gdb) py-bt
Undefined command: "py-bt".  Try "help".
```

- gdb *autoloads* debugging scripts from `$PATH-gdb.py`

```
(gdb) info auto-load

gdb-scripts:  No auto-load scripts.
libthread-db:  No auto-loaded libthread-db.
local-gdbinit:  Local .gdbinit file was not found.
python-scripts:
Loaded  Script
Yes     /usr/share/gdb/auto-load/usr/bin/python2.7-gdb.py
```

Gotchas: virtual environments and custom CPython builds

- you can always load the scripts manually

```
(gdb) source /usr/share/gdb/auto-load/usr/bin/python2.7-gdb.py
```

- it's also useful for testing of the new versions of gdb scripts shipped with CPython

Gotchas: PTRACE_ATTACH not permitted

Controlled by `/proc/sys/kernel/yama/ptrace_scope`, possible values are

- `0` - a process can `PTRACE_ATTACH` to any other process running under the same uid
- `1` - only descendants can be traced (default on Ubuntu)
- `2` - admin-only attach, or through children calling `PTRACE_TRACEME`
- `3` - no processes may use ptrace with `PTRACE_ATTACH` nor via `PTRACE_TRACEME`

Gotchas: python-dbg

- a separate build of CPython (with `--with-pydebug` passed to `./configure`) with many run-time checks enabled, thus, *much* slower
- not required for using gdb

```
$ time python -c "print(sum(range(1, 1000000)))"
499999500000

real    0m0.096s
user    0m0.057s
sys 0m0.030s

$ time python-dbg -c "print(sum(range(1, 1000000)))"
499999500000
[18318 refs]

real    0m0.237s
user    0m0.197s
sys 0m0.016s
```

Gotchas: compiler build flags

- some Linux distros build CPython with `-g0` or `-g1` flags passed to gcc: the former produces a binary without debugging information at all, and the latter does not allow gdb to get information about local variables at runtime
- the solution is to rebuild CPython with `-g` or `-g2` (`2` is the default value when `-g` is passed)

Gotchas: optimized out frames

- depending on the optimization level used in gcc when building CPython or the exact compiler version used, it's possible that information on local variables or function arguments will be lost at runtime (e.g. with aggressive optimizations enabled by `-O3`)

```
(gdb) py-bt
Traceback (most recent call first):
  File "test2.py", line 9, in g
    time.sleep(1000)
  File "test2.py", line 5, in f
    g()
  (frame information optimized out)
```

Gotchas: optimized out frames

- it's still possible to debug such builds of CPython, though it may be tricky

```
(gdb) disassemble
Dump of assembler code for function PyEval_EvalFrameEx:
...
  0x00007ffff7a04e88 <+8>:     mov    %rdi,%r12
...

(gdb) p ((PyObject*) $r12)->ob_type->tp_name
$97 = 0x7ffff7ab59f0 "frame"

(gdb) p (char*) (&((PyUnicodeObject*) ((PyFrameObject*) $r12)
                    ->f_code->co_name)->_base->_base + 1)
$98 = 0x7ffff6a8aca0 "g"
```

Gotchas: PyPy, Jython, etc

- the described debugging technique is only feasible for the CPython interpreter as is, as the gdb extension is specifically written to introspect the state of CPython internals (e.g. `PyEval_EvalFrameEx` calls)
- for PyPy there is an open issue on Bitbucket, where it was proposed to provide integration with gdb, but looks like the attached patches have not been merged yet and the person, who wrote those, lost interest in this
- for Jython you could probably use standard tools for debugging of JVM applications, e.g. VisualVM

Links

- gdb Debugging Full Example: http://brendangregg.com/blog/2016-08-09/gdb-example-ncurses.html
- Low-level Python debugging with gdb: http://grapsus.net/blog/post/Low-level-Python-debugging-with-GDB
- a blog post on CPython internals: https://tech.blog.aknin.name/category/my-projects/pythons-innards/
- pydevd: http://pydev.blogspot.com/2014/09/attaching-debugger-to-running-process.html
- pyringe: https://github.com/google/pyringe

Conclusion

- gdb is a powerful tool, that allows one to debug complex problems with crashing or hanging CPython processes, as well as Python code, that does calls to native libraries
- on modern Linux distros debugging CPython processes with gdb must be as simple as installing of debugging symbols for the interpreter build, although there are a few known gotchas, especially when virtual environments are used

Questions?

Your feedback is very appreciated!

twitter: @rpodoliaka
blog: http://podoliaka.org
slides: http://podoliaka.org/talks/