llnode for Node.js Memory Leak Analysis

13 Jul 2016

The memory of your Node.js process is growing endlessly, what do you do?

I start with a page fault flame graph using Linux perf, which can be generated on the live running process with low overhead. That only solves some issues, though. Other approaches include analyzing heap snapshots with heapdump, or taking a core dump and analyzing it with mdb findjsobjects on an old Solaris image.

Fortunately, findjsobjects has just been made available for Linux in llnode, an lldb plugin, which I'll write about here. Thanks to Fedor Indutny for creating llnode, Howard Hellyer for contributing findjsobjects support, and thanks to Dave Pacheco for first developing this kind of analysis.

llnode is not yet well known or documented. To help out, I'll share the commands and screenshots I used for taking a fresh Ubuntu Xenial server with Node v4.4.7 and running some memory object analysis. NOTE: It's 13-Jul-2016, and I'd expect this to be simplified in future versions. See the llnode repository for the latest.

1. Installing llnode

Just the commands:

```
sudo bash
apt-get update
apt-get install -y lldb-3.8 lldb-3.8-dev gcc g++ make gdb lldb
apt-get install -y python-pip; pip install six
git clone https://github.com/nodejs/llnode
cd llnode
git clone https://chromium.googlesource.com/external/gyp.git tools/gyp
./gyp_llnode -Dlldb_dir=/usr/lib/llvm-3.8/
make -C out/ -j2
make install-linux
```

(Update: this used to be https://github.com/indutny/llnode .)

Perhaps is the future this will just be "apt-get install -y llnode" (update: there's now "npm install llnode"). llnode works for MacOS as well, and can be installed via brew.

2. Taking a Core Dump

You can use gcore, noting that it may pause node for some period (short or long). My Node.js PID is 30833.

```
# ulimit -c unlimited
# gcore 30833
[New LWP 30834]
[New LWP 30835]
[New LWP 30836]
[New LWP 30837]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
syscall () at ../sysdeps/unix/sysv/linux/x86_64/syscall.S:38
38  ../sysdeps/unix/sysv/linux/x86_64/syscall.S: No such file or directory.
warning: target file /proc/30833/cmdline contained unexpected null characters
Saved corefile core.30833
# ls -lh core.30833
-rw-r--r-- 1 root root 855M Jul 12 20:56 core.30833
```

This has hit a warning. It's likely that the gdb (which includes gcore) installed for Xenial isn't matching this kernel version. I thought I'd better note this in case you hit it. If you can't get gcore to work at all, maybe lldb and "process save-core"?

Another problem you might hit is that the core file hangs when lldb tries to read it:

```
# lldb -f /usr/bin/node -c /var/cores/core.node.30833
(lldb) target create "/usr/bin/node" --core "/var/cores/core.node.30833"
[...hang...]
```

This time I suspect it's the known lldb bug 26322 Core load hangs, which was fixed in late 2016, but you might have an older lldb version that misses the fix. This only affects core files taken from live processes, rather than those from a crash.

Since I was hitting that lldb bug, I tried crashing the process instead. WARNING, this will crash the process! To start with I disabled Ubuntu's appreport, and setup traditional core dumps, then sent a SIGBUS (why not):

```
# mkdir /var/cores
# echo "/var/cores/core.%e.%p.%t" > /proc/sys/kernel/core_pattern
# kill -BUS 30833
# ls -lh /var/cores/core.node.30833.1468367170
-rw------- 1 root root 65M Jul 12 23:46 /var/cores/core.node.30833.1468367170
```

For production use I'll go fix gcore (although, we are in a fault-tolerant environment).

3. Memory Ranges File

On lldb 3.8 and earlier, an extra step was needed for findjsobjects to work (this was fixed lldb 3.9). This is where a memory ranges file must be created and environment variable must be set. There are scripts to generate this file for Mac OS and Linux in llnode. While in the llnode directory:

```
# ./scripts/readelf2segments.py /var/cores/core.node.30833.1468367170 > core.30833.ranges
# export LLNODE_RANGESFILE=core.30833.ranges
```

Again, this step vanishes on lldb 3.9.

4. Starting llnode

Starting up lldb (as root):

```
# lldb -f /usr/bin/node -c /var/cores/core.node.30833.1468367170
(lldb) target create "/usr/bin/node" --core "/var/cores/core.node.30833.1468367170"
Core file '/var/cores/core.node.30833.1468367170' (x86_64) was loaded.
```

Your /usr/bin/node should exist, and match the node version that the core dump is from. If you are debugging multiple node versions, you can keep the binaries under different paths and specify them in `-f` as needed. You can also omit the `-f /usr/bin/node` option.

The help message:

```
(lldb) v8
The following subcommands are supported:

      bt              -- Show a backtrace with node.js JavaScript functions and their args. An op
                         the number of frames to display. Otherwise all frames will be dumped.

                         Syntax: v8 bt [number]
      findjsinstances -- List all objects which share the specified map.
                         Accepts the same options as `v8 inspect`
      findjsobjects   -- List all object types and instance counts grouped by map and sorted by i
                         Requires `LLNODE_RANGESFILE` environment variable to be set to a file co
                         There are scripts for generating this file on Linux and Mac in the scrip
      inspect         -- Print detailed description and contents of the JavaScript value.

                         Possible flags (all optional):

                          * -F, --full-string    - print whole string without adding ellipsis
                          * -m, --print-map      - print object's map address
                          * --string-length num  - print maximum of `num` characters in string

                         Syntax: v8 inspect [flags] expr
      print           -- Print short description of the JavaScript value.

                         Syntax: v8 print expr
      source          -- Source code information

For more help on any particular subcommand, type 'help  '.
```

Run this yourself to see the latest.

5. Stack Trace

The bt command will print the stack backtrace (not interesting in this case, just showing it works):

```
(lldb) v8 bt
 * thread #1: tid = 0, 0x00007f75719a7c19 libc.so.6`syscall + 25 at syscall.S:38, name = 'node',
  * frame #0: 0x00007f75719a7c19 libc.so.6`syscall + 25 at syscall.S:38
    frame #1: 0x0000000000fc375a node`uv__epoll_wait(epfd=, events=, nevents=, timeout=) + 26 at
    frame #2: 0x0000000000fc1838 node`uv__io_poll(loop=0x00000000019aa080, timeout=-1) + 424 at l
    frame #3: 0x0000000000fb2506 node`uv_run(loop=0x00000000019aa080, mode=UV_RUN_ONCE) + 342 at
    frame #4: 0x0000000000dfedf8 node`node::Start(int, char**) + 1080
    frame #5: 0x00007f75718c7830 libc.so.6`__libc_start_main(main=(node`main), argc=2, argv=0x000
    frame #6: 0x0000000000722f1d node`_start + 41
```

Remember to run `v8 bt` and not just `bt`, which I do out of habit.

6. llnode Object Analysis

Finding all JavaScript objects (reminds me of Java's jmap -histo):

```
(lldb) v8 findjsobjects
 Instances  Total Size Name
 ---------- ---------- ----
          1         24 (anonymous)
          1         24 JSON
          1         24 process
          1         32 Arguments
          1         32 HTTPParser
          1         32 Signal
          1         32 Timer
          1         32 WriteWrap
          1         56 DefineError.aV
          1         56 MathConstructor
          1         56 RangeError
          1         96 Console
          1        104 Agent
          1        112 Server
          1        120 exports.FreeList
          1        136 Module
          2         64 TTY
          2        208 EventEmitter
          2        208 WriteStream
          6        336 Error
          7        560 (ArrayBufferView)
         29        704 (Object)
         42       2352 NativeModule
       1219     146280 ServerResponse
       1219     263304 IncomingMessage
       1220      39040 TCP
       1859     416416 Socket
       1860     386880 WritableState
       2435     136360 WriteReq
       3080     591360 ReadableState
       3718     178528 CorkedRequest
       9708     543000 Object
       9747     467856 TickObject
      32585    1042720 (Array)
```

This is a simple app without a real issue, I'm just showing the commands and screenshots. At this point you'd be looking for large counts of an unexpected object to study further. You could also take two core dumps at different times, and then look at the difference between the findjsobject counts, to see which objects were growing.

Listing all (2) instances of WriteStream:

```
(lldb) v8 findjsinstances WriteStream
0x0000360583199f89:<Object: WriteStream>
0x0000360583199ff1:<Object: WriteStream>
```

Now inspecting one of those instances of WriteStream:

```
(lldb) v8 i 0x0000360583199f89
0x0000360583199f89:<Object: WriteStream properties {
    ._connecting=0x00001c604c904251:<false>,
    ._hadError=0x00001c604c904251:<false>,
    ._handle=0x0000360583119091:<Object: TTY>,
    ._parent=0x00001c604c904101:<null>,
    ._host=0x00001c604c904101:<null>,
    ._readableState=0x00003605831a8dd1:<Object: ReadableState>,
    .readable=0x00001c604c904251:<false>,
    .domain=0x00001c604c904101:<null>,
    ._events=0x00003605831a8e91:<Object: Object>,
    ._eventsCount=<Smi: 3>,
    ._maxListeners=0x00001c604c9041b9:<undefined>,
    ._writableState=0x00003605831a2609:<Object: WritableState>,
    .writable=0x00001c604c904211:<true>,
    .allowHalfOpen=0x00001c604c904251:<false>,
    .destroyed=0x00001c604c904251:<false>,
    ._bytesDispatched=<Smi: 36>,
    ._sockname=0x00001c604c904101:<null>,
    ._writev=0x00001c604c904101:<null>,
    ._pendingData=0x00001c604c904101:<null>,
    ._pendingEncoding=0x00001c604c904291:<String: "">,
    .server=0x00001c604c904101:<null>,
    ._server=0x00001c604c904101:<null>,
    .<non-string>=<Smi: 0>,
    .columns=<Smi: 172>,
    .rows=<Smi: 42>,
    ._type=0x00001c604c9bd571:<String: "tty">,
    .fd=<Smi: 1>,
    ._isStdio=0x00001c604c904211:<true>,
    .destroySoon=0x00001002183b4821:<function: stdout.destroy.stdout.destroySoon at node.js:633:5
    .destroy=0x00001002183b4821:<function: stdout.destroy.stdout.destroySoon at node.js:633:53>}>
```

Listing all instances of Socket:

```
(lldb) v8 findjsinstances Socket
0x00000c41530496f9:<Object: Socket>
0x00000c4153049909:<Object: Socket>
0x00000c4153049b19:<Object: Socket>
0x00000c4153049d29:<Object: Socket>
0x00000c4153049f39:<Object: Socket>
0x00000c415304a149:<Object: Socket>
0x00000c415304a359:<Object: Socket>
[...]
```

Inspecting an instance of Socket:

```
(lldb) v8 i 0x00000c41530496f9
0x00000c41530496f9:<Object: Socket properties {
    ._connecting=0x00001c604c904251:<false>,
    ._hadError=0x00001c604c904251:<false>,
    ._handle=0x00001c604c904101:<null>,
    ._parent=0x00001c604c904101:<null>,
    ._host=0x00001c604c904101:<null>,
    ._readableState=0x00000c4153087361:<Object: ReadableState>,
    .readable=0x00001c604c904251:<false>,
    .domain=0x00001c604c904101:<null>,
    ._events=0x00000c4153087421:<Object: Object>,
    ._eventsCount=<Smi: 10>,
    ._maxListeners=0x00001c604c9041b9:<undefined>,
    ._writableState=0x00000c41530497d9:<Object: WritableState>,
    .writable=0x00001c604c904251:<false>,
    .allowHalfOpen=0x00001c604c904211:<true>,
    .destroyed=0x00001c604c904211:<true>,
    ._bytesDispatched=<Smi: 145>,
    ._sockname=0x00001c604c904101:<null>,
    ._pendingData=0x00001c604c904101:<null>,
    ._pendingEncoding=0x00001c604c904291:<String: "">,
    .server=0x0000360583196da1:<Object: Server>,
    ._server=0x0000360583196da1:<Object: Server>,
    .<non-string>=<Smi: 59>,
    ._idleTimeout=<Smi: -1>,
    ._idleNext=0x00001c604c904101:<null>,
    ._idlePrev=0x00001c604c904101:<null>,
    ._idleStart=<Smi: 3883>,
    .parser=0x00001c604c904101:<null>,
    .on=0x0000360583196801:<function: socketOnWrap at _http_server.js:575:22>,
    ._paused=0x00001c604c904251:<false>,
    .read=0x00001002183260c9:<function: Readable.read at _stream_readable.js:258:35>,
    ._consuming=0x00001c604c904211:<true>,
    ._httpMessage=0x00001c604c904101:<null>}>
```

One of the largest Object counts for my process was TickObject:

```
(lldb) v8 findjsinstances TickObject
0x00000c41535a9221:<Object: TickObject>
0x00000c41535a9301:<Object: TickObject>
0x00000c41535a93e1:<Object: TickObject>
0x00000c41535a94c1:<Object: TickObject>
0x00000c41535aa241:<Object: TickObject>
0x00000c41535ab5a9:<Object: TickObject>
0x00000c41535ab7b1:<Object: TickObject>
[...]
(lldb) v8 i 0x00000c41535a9221
0x00000c41535a9221:<Object: TickObject properties {
    .callback=0x000036058319be09:<function: endReadableNT at _stream_readable.js:916:23>,
    .domain=0x00001c604c904101:<null>,
    .args=0x00000c41535a9171:<Array: length=2>}>
```

Fewer details for this object, but hopefully still enough to continue investigating. It's given me a callback function, plus I know the object name.

That's all for now. Hopefully these steps and screenshots are helpful, although do note that the steps are likely to be improved in future versions. Follow llnode for the latest.

For more reading:

- findrefs command for llnode.
- Advances in core-dump debugging for Node.js, including an mdb v8 vs llnode comparison.

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).