Python & Memory

Tomasz Paczkowski
@oinopion

PyWaw, 14.07.2014

Disclaimer

- Code was executed on Ubuntu 12.04 x64 and cPython 2.7.3
- I’m not an expert in cPython
- It’s much more complicated than it looks like
- I’m not even sure anything here is true

Case Study

- Long lived web process
- Periodically allocates boatloads of memory
- For some reason, it's never released

Distilled code

```
big = alloc(100000)
report('After alloc')
small = alloc(1)
del big
report('After del')
```

Output

```
$ python frag.py
After alloc: 502244 kB used
After del: 501484 kB used
```

Problem hammering

```
big = alloc(100000)
report('After alloc')
small = alloc(1)
del big
report('After del')
import gc; gc.collect(2)
report('After gc')
```

```
$ python frag.py
After alloc: 502216 kB used
After del: 501460 kB used
After gc: 501496 kB used
```

Enter our hero

- **Guppy** is the only tool I've found usable and useful
- http://guppy-pe.sourceforge.net
- Documentation is… not it's greatest point
- Still better than others

Debugging with Guppy

```
from guppy import hpy

big = alloc(100000)
report('After alloc')
print hpy().heap()[:3]
small = alloc(1)
del big
report('After del')
print hpy().heap()[:3]
```

Output

```
$ python frag-debug.py
After alloc: 502448 kB used
Partition of a set of 116311 objects.
Total size = 506138848 bytes.
 Index  Count   %     Size   % Cumulative  % Kind
     0 110222  95 504818568 100 504818568 100 str
     1    179   0    844888   0 505663456 100 list
     2   5910   5    475392   0 506138848 100 tuple

After del: 511676 kB used
Partition of a set of 16028 objects.
Total size = 1510312 bytes.
 Index  Count   %     Size   % Cumulative  % Kind
     0  10061  63   814552  54     814552  54 str
     1   5894  37   474104  31    1288656  85 tuple
     2     73   0   221656  15    1510312 100 dict of module
```

Diagnose:
Memory Fragmentation

However, removing all “small” allocations did not help in this case.

Fun with Python allocator

- Python does not use malloc directly — too costly for small objects

- Instead implements more sophisticated allocator on top of malloc

Free lists

- For handful of most common types Python keeps unused objects of similar size in so called free lists
- Those are most significantly: lists, dictionaries, frames
- Speeds up code execution immensely by not hitting malloc and saying in user space

Free list torture

```
big = []
for i in xrange(500):
    strings = alloc(i)
    big.extend(strings)
report('After work')

del big
report('After del')
```

Output

```
$ python lists.py
After work: 622172 kB used
After del: 621248 kB used
```

Solutions

- Make better use of memory
- Subprocess
- jemalloc* via `LD_PRELOAD`

Using jemalloc

```
$ python frag.py
After alloc: 502212 kB used
After del: 501456 kB used
After gc: 501492 kB used

$ export LD_PRELOAD=/usr/lib/libjemalloc.so.1
$ python frag.py
After alloc: 814084 kB used
After del: 11060 kB used
After gc: 6988 kB used
```

Conclusions

- Sometimes memory leak is not what it seems
- `malloc` from glibc is not the best of breed
- Do memory intensive work in subprocess
- Be mindful when using C extensions

Thanks. Questions?