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

```python
big = alloc(100000)
report('After alloc')
small = alloc(1)
del big
report('After del')
```
$ python frag.py
After alloc: 502244 kB used
After del: 501484 kB used
Problem hammering

```python
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
- Documentation is… not it’s greatest point
- Still better than others
Debugging with Guppy

```python
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])
```
$ python frag-debug.py
After alloc: 502448 kB used
Partition of a set of 116311 objects.
Total size = 506138848 bytes.

<table>
<thead>
<tr>
<th>Index</th>
<th>Count</th>
<th>%</th>
<th>Size</th>
<th>% Cumulative</th>
<th>% Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>110222</td>
<td>95</td>
<td>504818568</td>
<td>100</td>
<td>100 str</td>
</tr>
<tr>
<td>1</td>
<td>179</td>
<td>0</td>
<td>844888</td>
<td>0</td>
<td>505663456 100 list</td>
</tr>
<tr>
<td>2</td>
<td>5910</td>
<td>5</td>
<td>475392</td>
<td>0</td>
<td>506138848 100 tuple</td>
</tr>
</tbody>
</table>

After del: 511676 kB used
Partition of a set of 16028 objects.
Total size = 1510312 bytes.

<table>
<thead>
<tr>
<th>Index</th>
<th>Count</th>
<th>%</th>
<th>Size</th>
<th>% Cumulative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10061</td>
<td>63</td>
<td>814552</td>
<td>54</td>
</tr>
<tr>
<td>1</td>
<td>5894</td>
<td>37</td>
<td>474104</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>73</td>
<td>0</td>
<td>221656</td>
<td>15</td>
</tr>
</tbody>
</table>

| % Kind | | |
|--------|---|---|---|
| str    | | | |
| list   | | | |
| tuple  | | | |
| dict of module | | | |
Diagnose:
Memory Fragmentation

- big

- big small

- small
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

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

deleted big
report('After del')
```
$ 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?