how's it going?

pycon 2017 edition

larry hastings

preface

exceed*ingly* technical!

multithreading

cpython internals

cf. “Python's Infamous GIL”

“The Gilectomy: Remvoing Python's GIL”

goal-ectomy

run e*xisting* multithreaded python programs

on multiple cores *simultaneously*

with as little C API breakage *as possible*

faster than CPython with a GIL does *by wall time*

approach

atomic incr/decr

fast internal locks on mutable objects

fast locks around C data structures

- obmalloc
- freelists

disable gc

profile and experiment!

gilectomy's official benchmark

```
def fib(n):
    if n < 2: return 1
    return fib(n-1) + fib(n-2)
```

overview

"atomic" (june)

buffered refcounts (october)

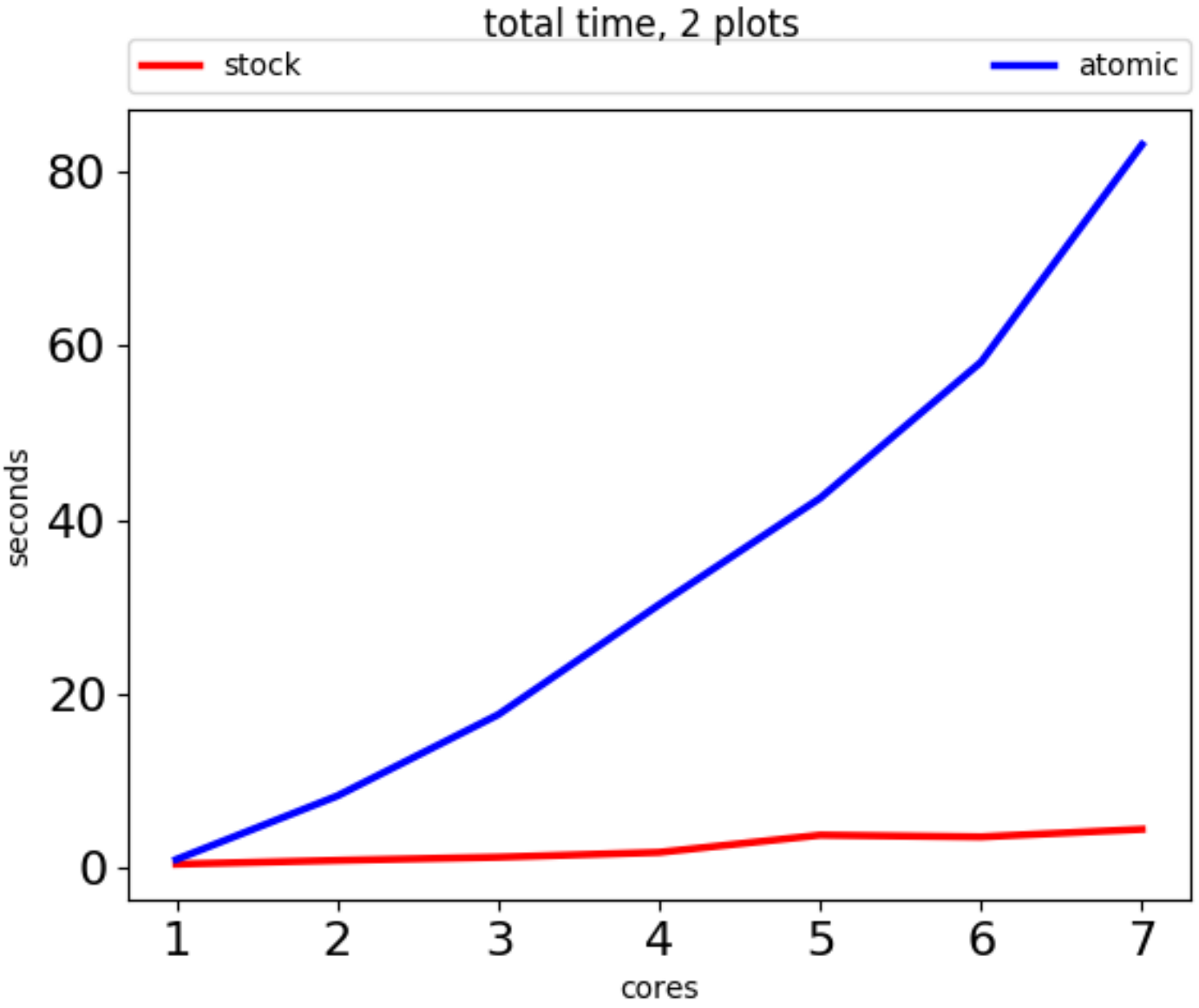
obmalloc (april)

no tls (may)

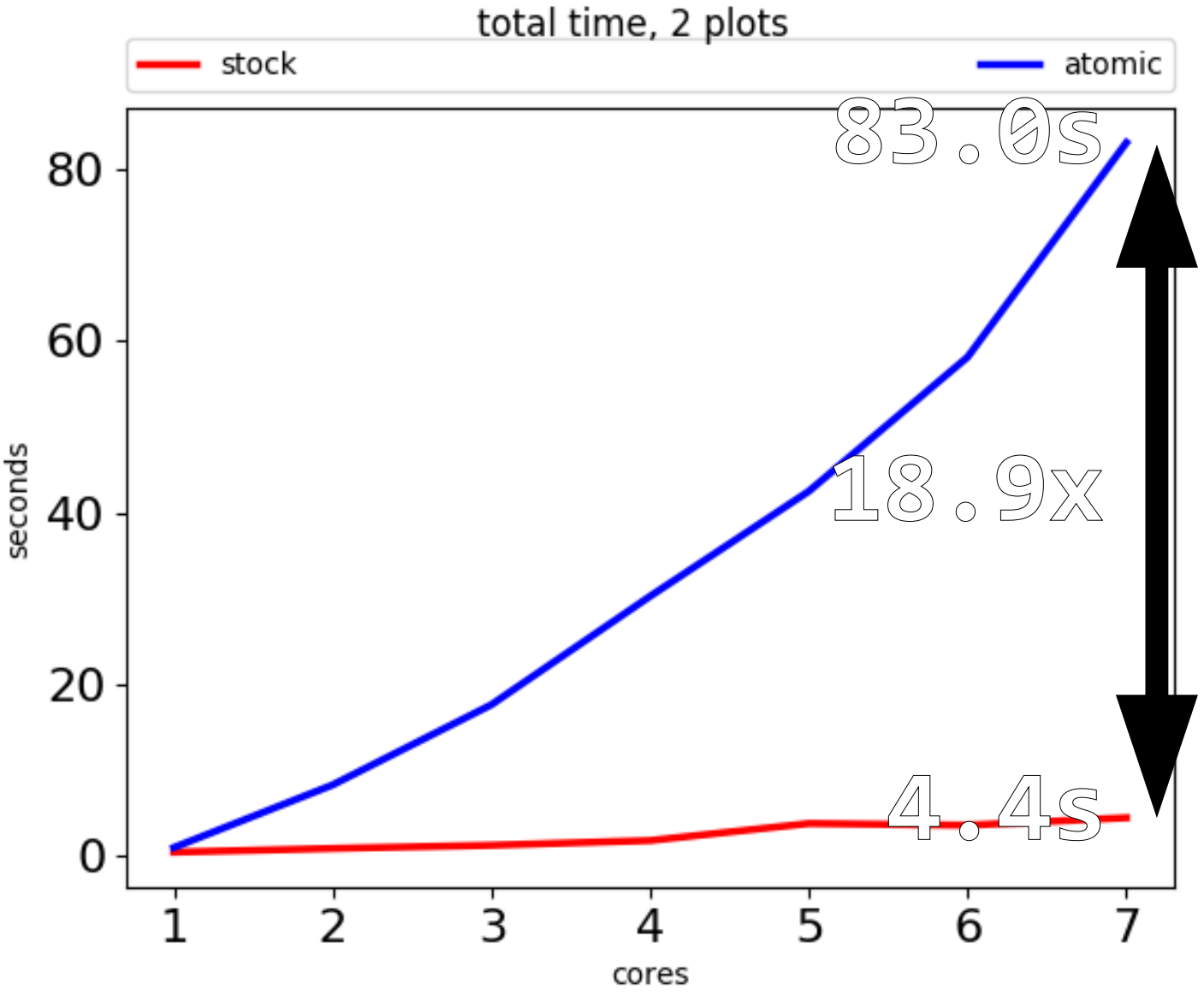
benchmarks are impossible

cpu MHz : 1200.024
cpu MHz : 1200.024
cpu MHz : 1200.024
cpu MHz : 1200.024
cpu MHz : 1201.135
cpu MHz : 1201.611
cpu MHz : 1203.039
cpu MHz : 1205.419
cpu MHz : 1207.165
cpu MHz : 1212.243
cpu MHz : 1215.734
cpu MHz : 1219.543
cpu MHz : 1220.178
cpu MHz : 1220.336
cpu MHz : 1224.621
cpu MHz : 1227.160
cpu MHz : 1230.493
cpu MHz : 1233.984
cpu MHz : 1242.712
cpu MHz : 1245.727
cpu MHz : 1247.631
cpu MHz : 1252.075
cpu MHz : 1252.868
cpu MHz : 1259.533
cpu MHz : 1271.435
cpu MHz : 1280.163
cpu MHz : 1289.050
cpu MHz : 1326.342
cpu MHz : 1350.781
cpu MHz : 1384.265
cpu MHz : 1395.214
cpu MHz : 1397.912
cpu MHz : 1496.936
cpu MHz : 1578.027
cpu MHz : 1697.998
cpu MHz : 2599.841
cpu MHz : 2947.692
cpu MHz : 3099.877
cpu MHz : 3099.877
cpu MHz : 3099.877
cpu MHz : 3099.877
cpu MHz : 3099.877
cpu MHz : 3100.036
cpu MHz : 3100.036
cpu MHz : 3100.036
cpu MHz : 3101.623
cpu MHz : 3103.051

cpu time
june 2017



cpu time
june 2017

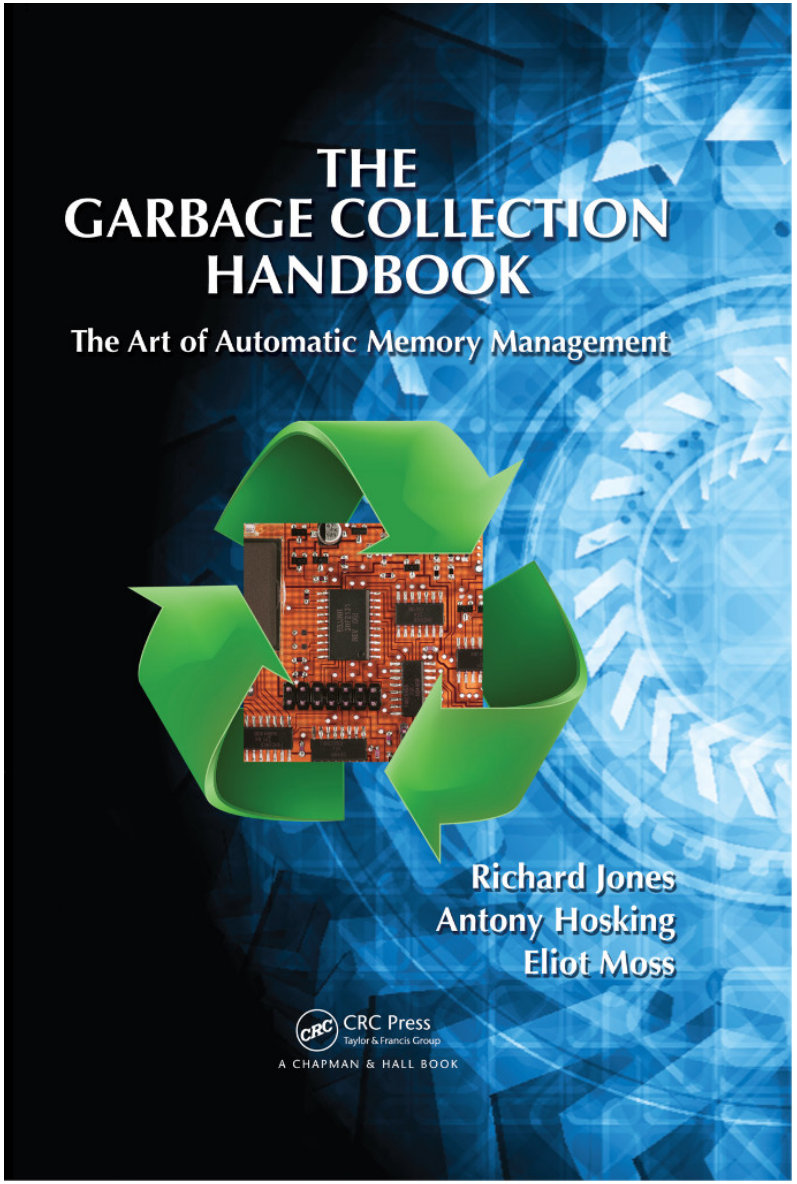


atomic incr/decr

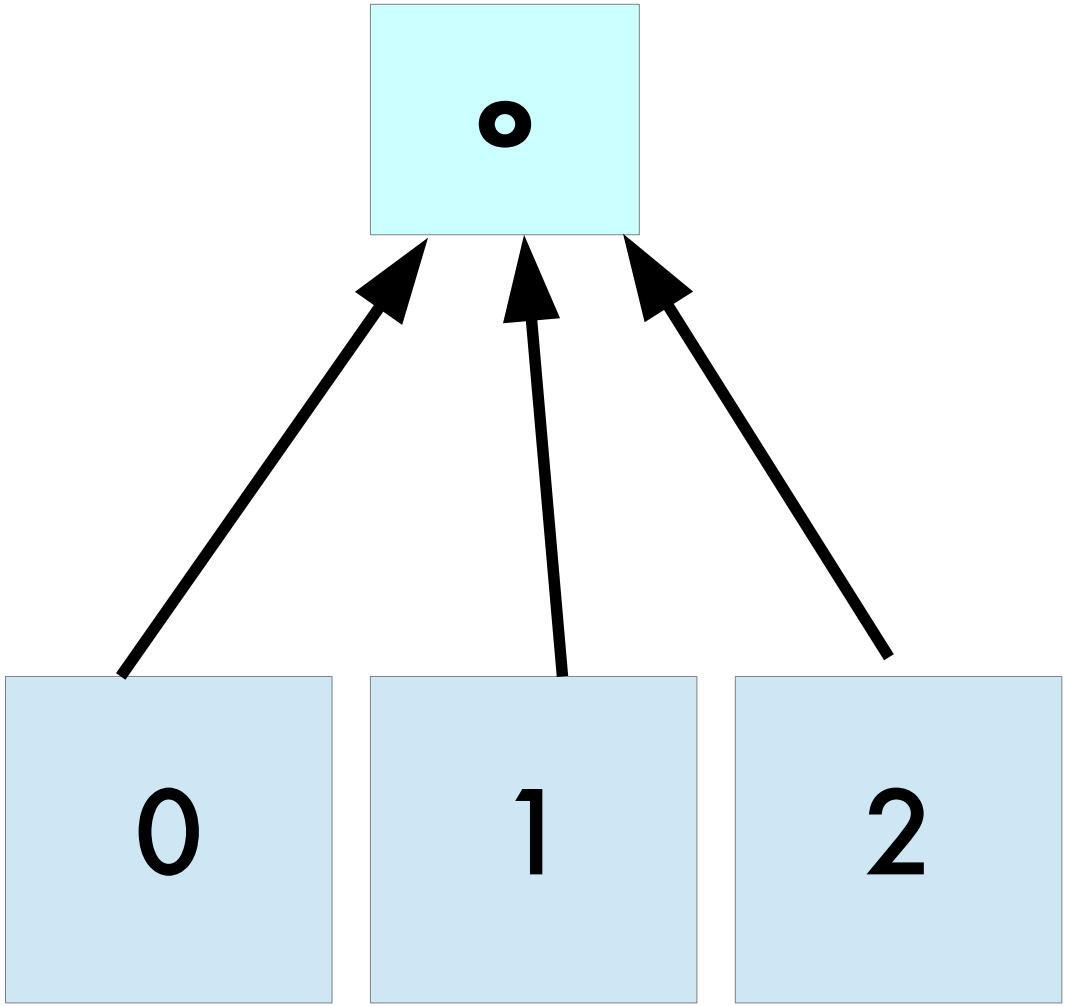
30% at 2 threads

rising overhead per thread

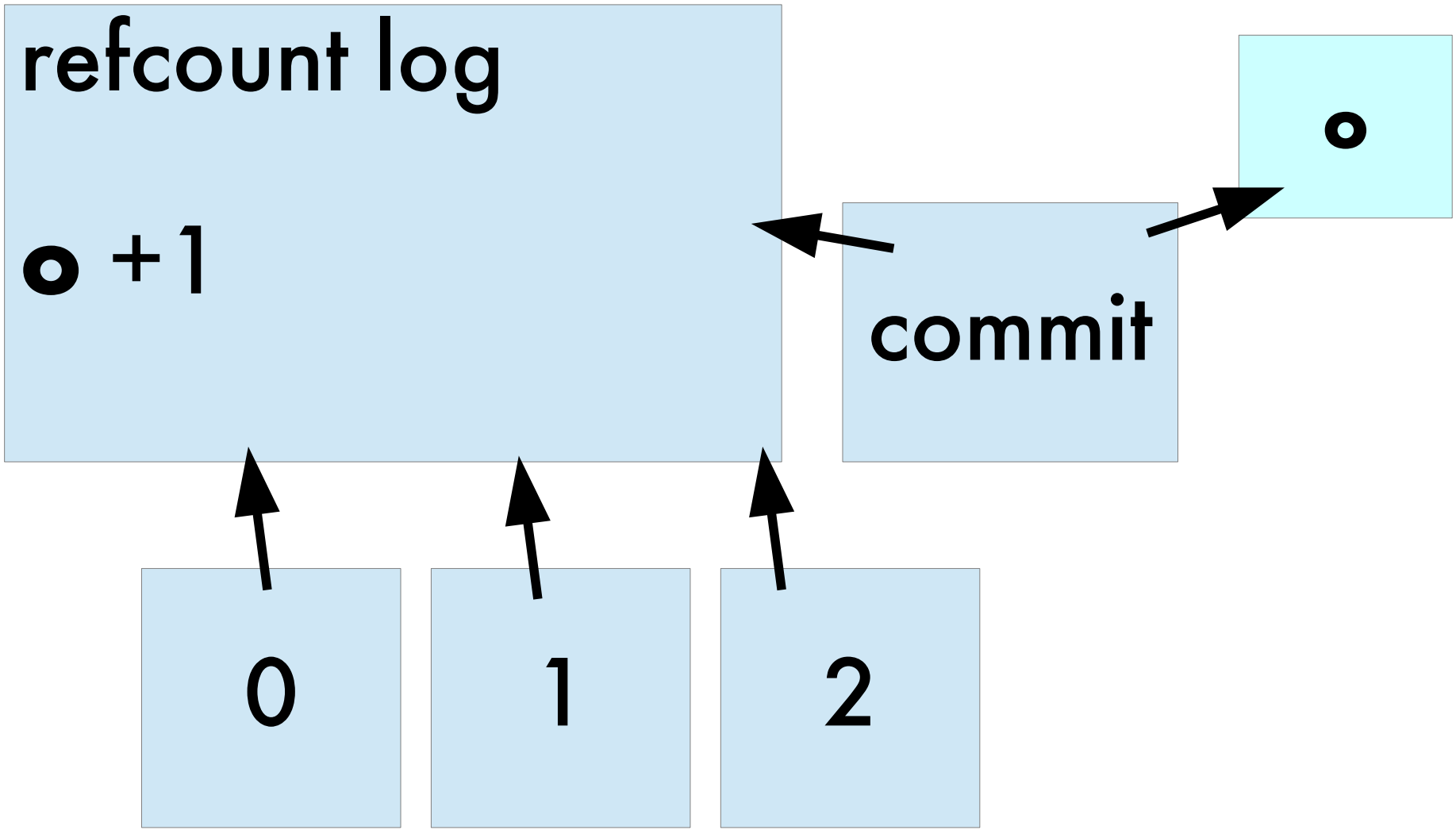
the garbage collection handbook



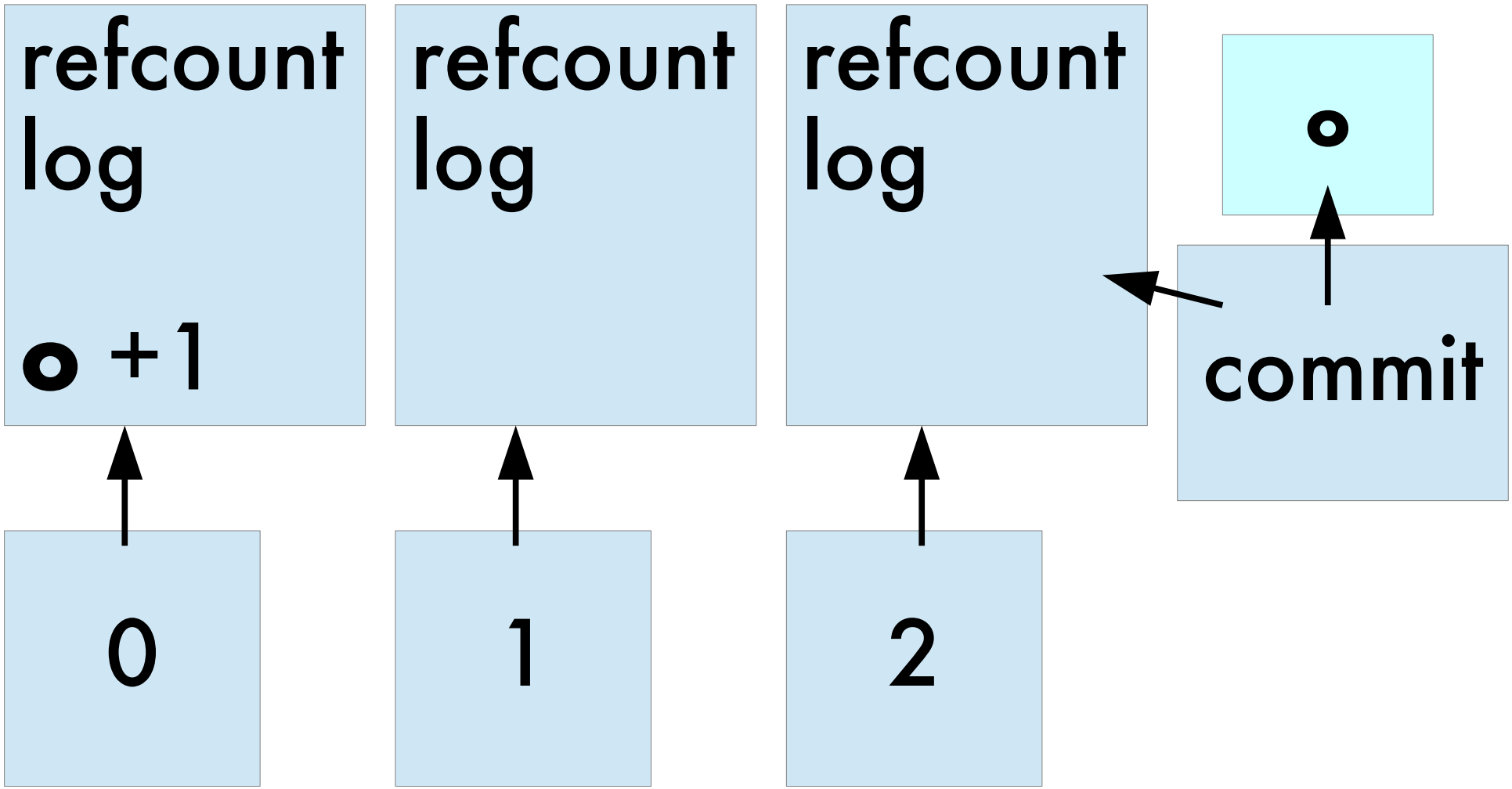
buffered reference counting



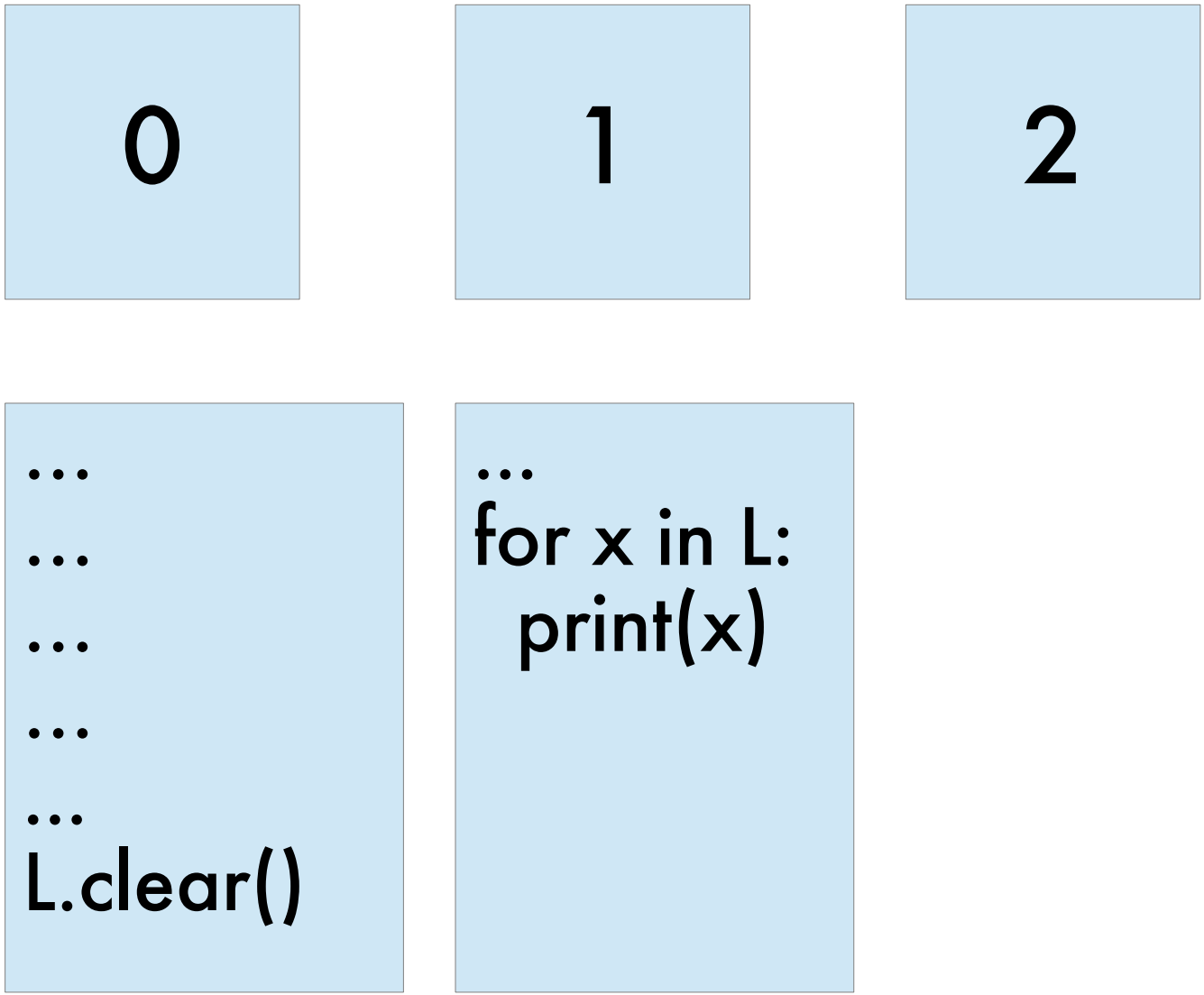
buffered reference counting



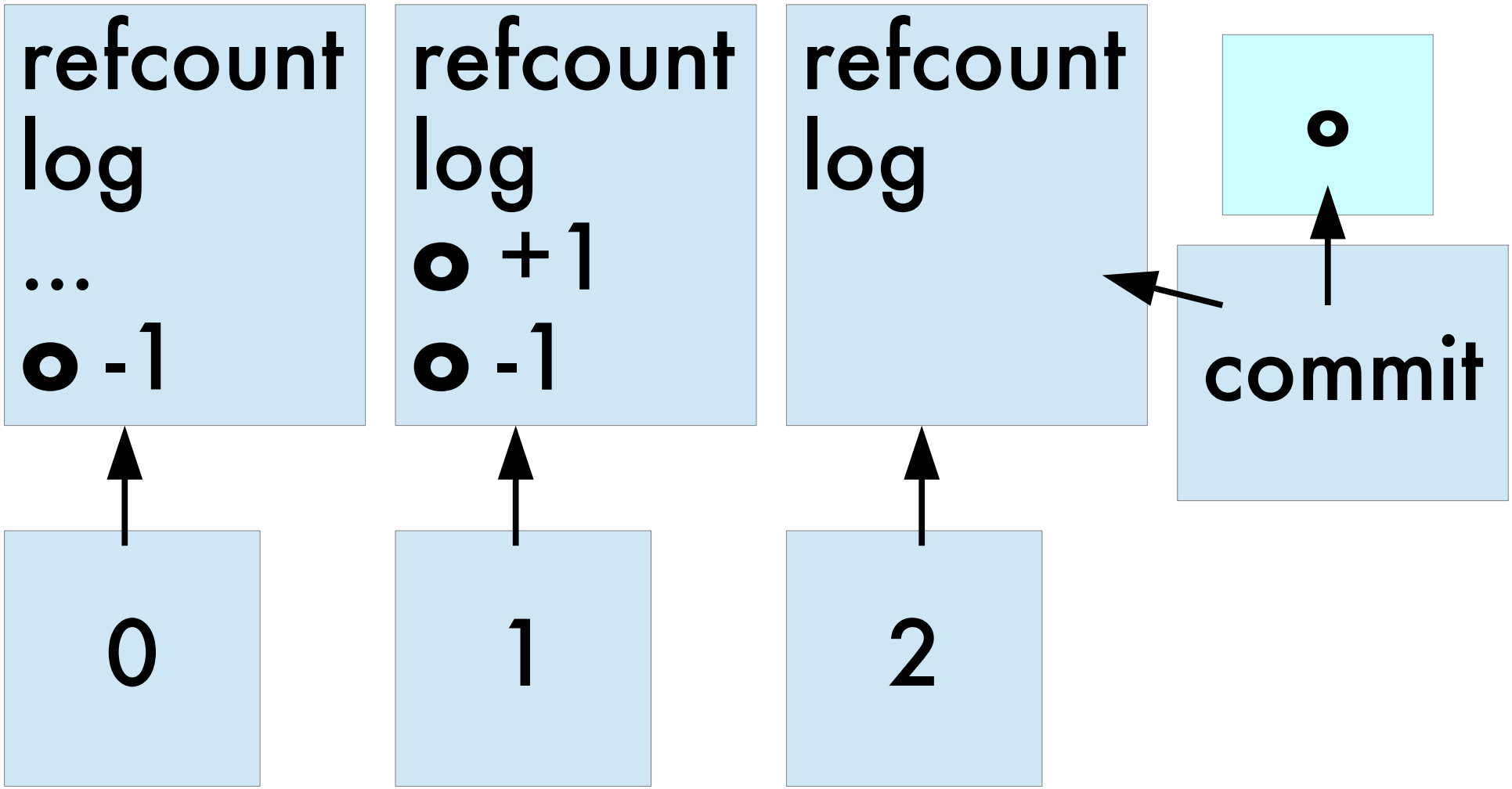
buffered reference counting



buffered reference counting



buffered reference counting



buffered reference counting

0

1

2

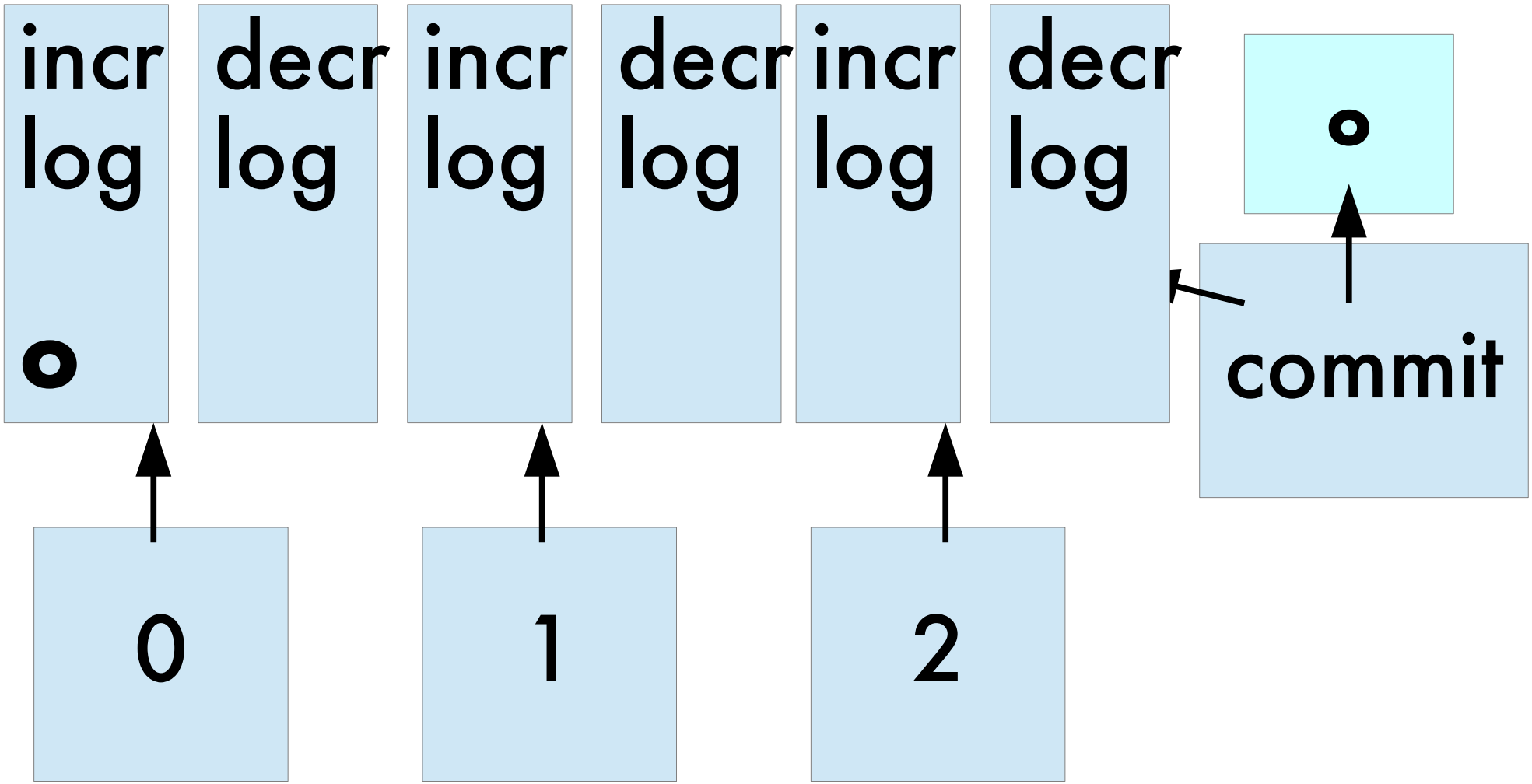
```
...
for x in L2:
  print(x)
...
...
L.clear()
```

```
...
for x in L:
  print(x)
...
...
L2.clear()
```

buffered reference counting

	2: incr	2: decr
1: incr		
1: decr		

buffered reference counting



undodb

http://undo.io/

incref and decref were simple...

```
#define Py_INCREF(op)                           \
    (((PyObject *)(op))->ob_refcnt++)
```

incref1 and decref1 are complex

```
#define Py_REFLOG \
    ((PyRefLog *)PyThread_get_key_value(PyRefLogTLSKey))
#define Py_REF_CACHE                            \
    PyRefLog *__py_reflog = Py_REFLOG           \

#define Py_INCREF1(o)                               \
    do {                                        \
        Py_REF_CACHE;                           \
        Py_INCREF2((o));                        \
    } while (0)                                 \
#define Py_INCREF Py_INCREF1
```

incref2 and decref2 are complex

```
#define Py_INCREF2(o)                                   \
    PyRefLog_Incref(__py_reflog, (PyObject *)(o))   \

#define PyRefLog_Incref(_rl, _o)      do {   \
        PyRefLog *rl = (_rl);               \
        PyObject *logged = (_o);            \
        if (PyRefPad_IsFull(rl->incref))   \
            PyRefLog_Rotate(rl);            \
        PyRefLog_UnsafeIncref(_rl, logged);\
    } while (0)                             \
```

incref3 and decref3 are complex

```
#define Py_INCREF3(o)                                   \
 PyRefLog_UnsafeIncref(__py_reflog, (PyObject *)(o)) \


#define PyRefLog_UnsafeIncref(_rl, _o)     \
    do {                                    \
        PyRefLog *rl2 = (_rl);              \
        PyObject *logged2 = (_o);           \
        PyRefPad_Write(rl2->incref, logged2);\
    } while (0)                             \
```

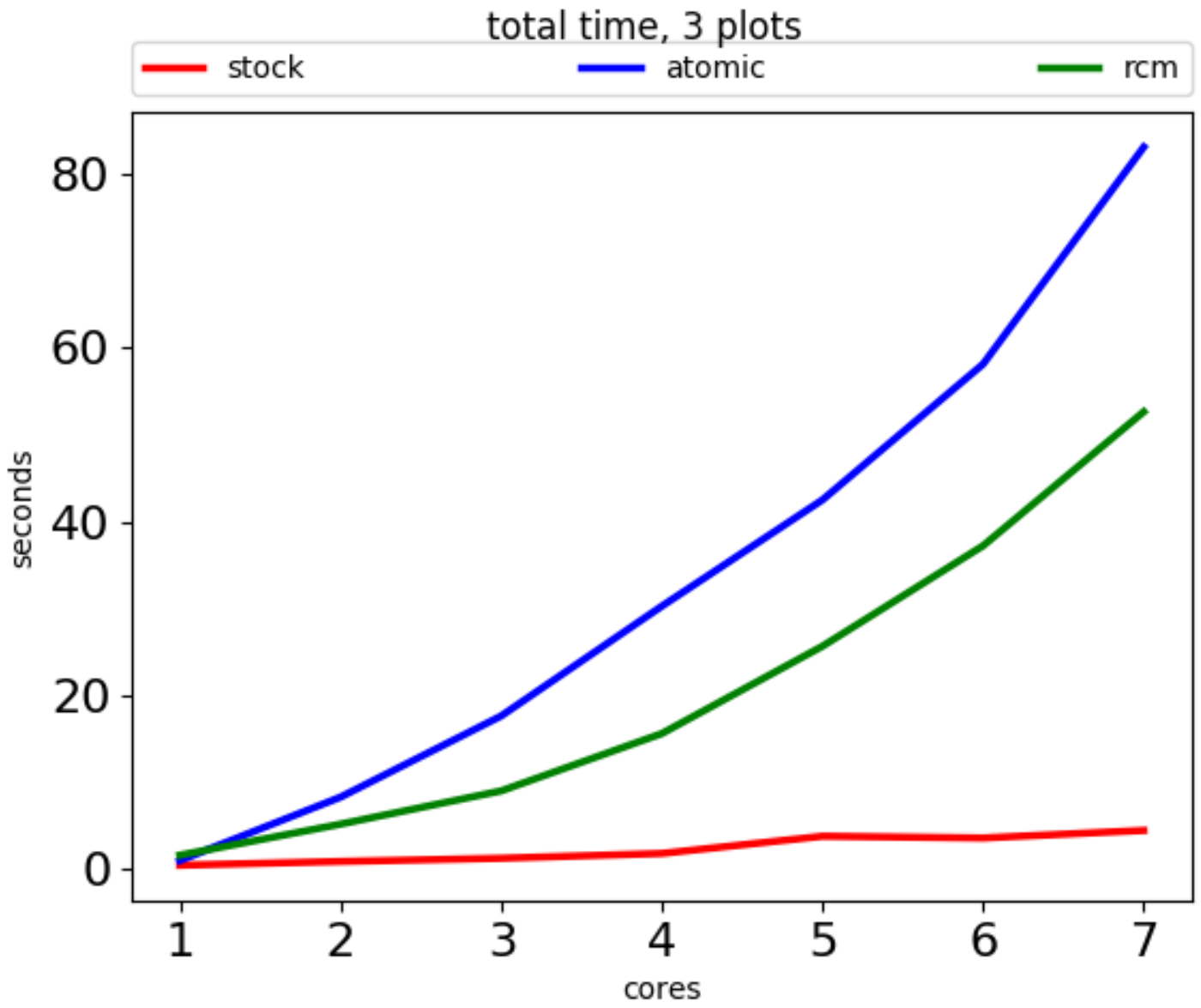
realtime reference counts

weakrefs

interned mortal strings

resurrecting objects (__del__)

cpu time
october 2016



obmalloc changes

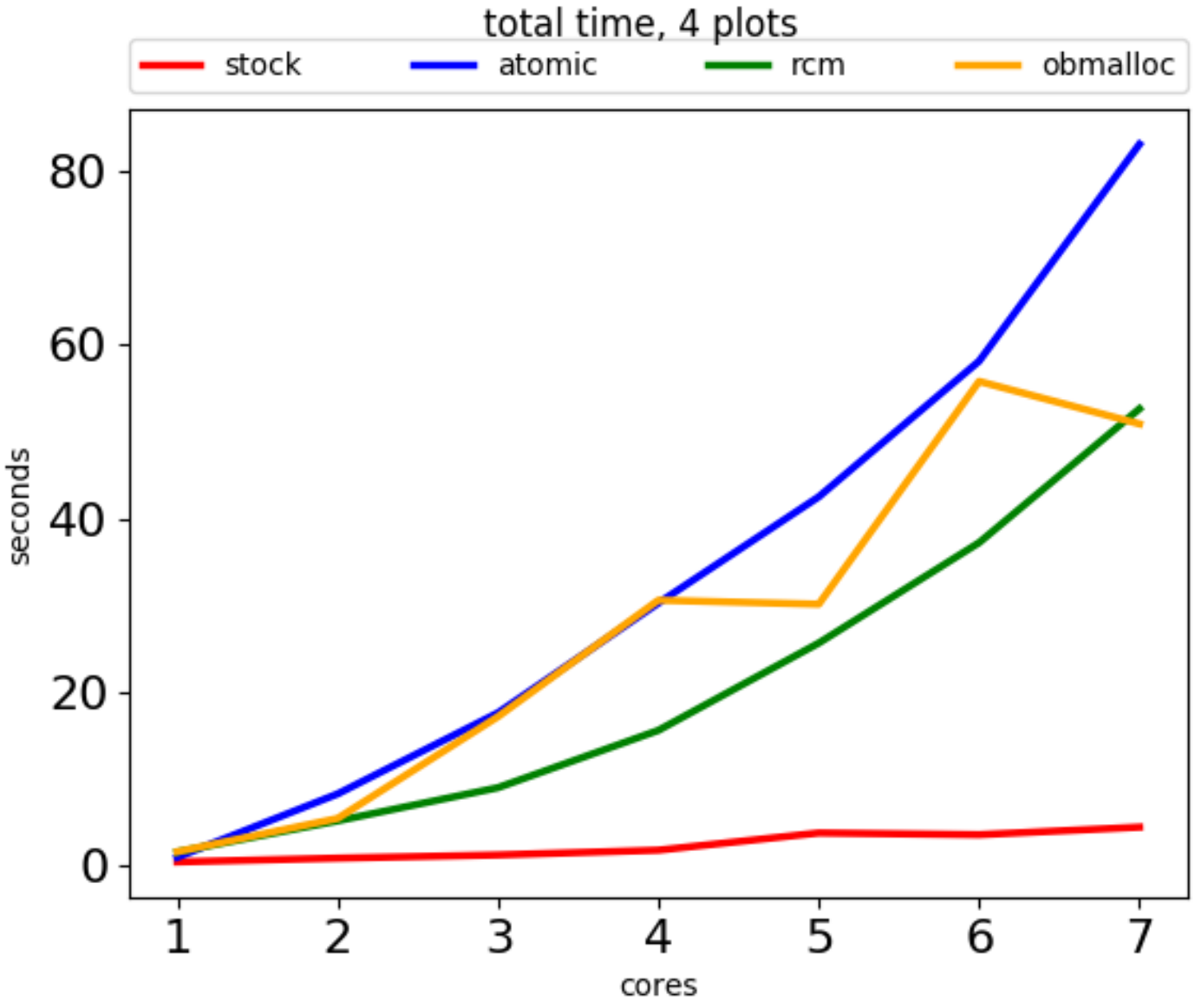
two-stage locking

- "fast" lock
- "heavy" lock

per-thread per-"class" freelist

remove **all** overhead from statistics

cpu time
april 2017



TLS calls 1

```
static PyObject *
PyEval_EvalFrameEx(…)
{
PyThreadState *tstate =
PyThreadState_GET();

…
res = call_function(…);
```

TLS calls 2

```
static PyObject *
call_function(...)
{
PyThreadState *tstate =
PyThreadState_GET();

…
x = fast_function(…);
```

TLS calls 3

```
static PyObject *
fast_function(…)
{
PyThreadState *tstate =
PyThreadState_GET();

…

retval = PyEval_EvalFrameEx(…);
```

TLS calls 4

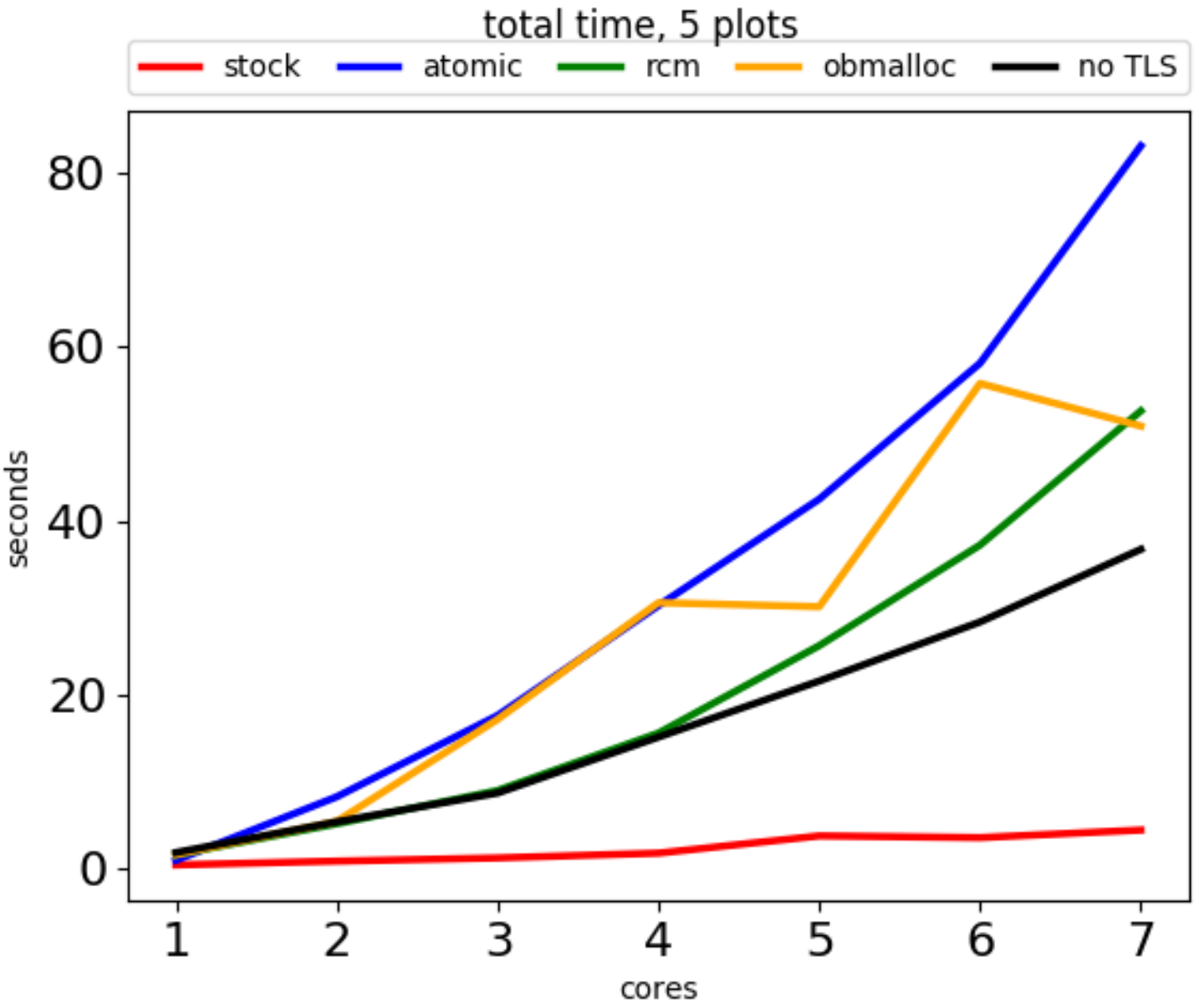
370m calls to pthread_getspecific

minimize TLS calls 3

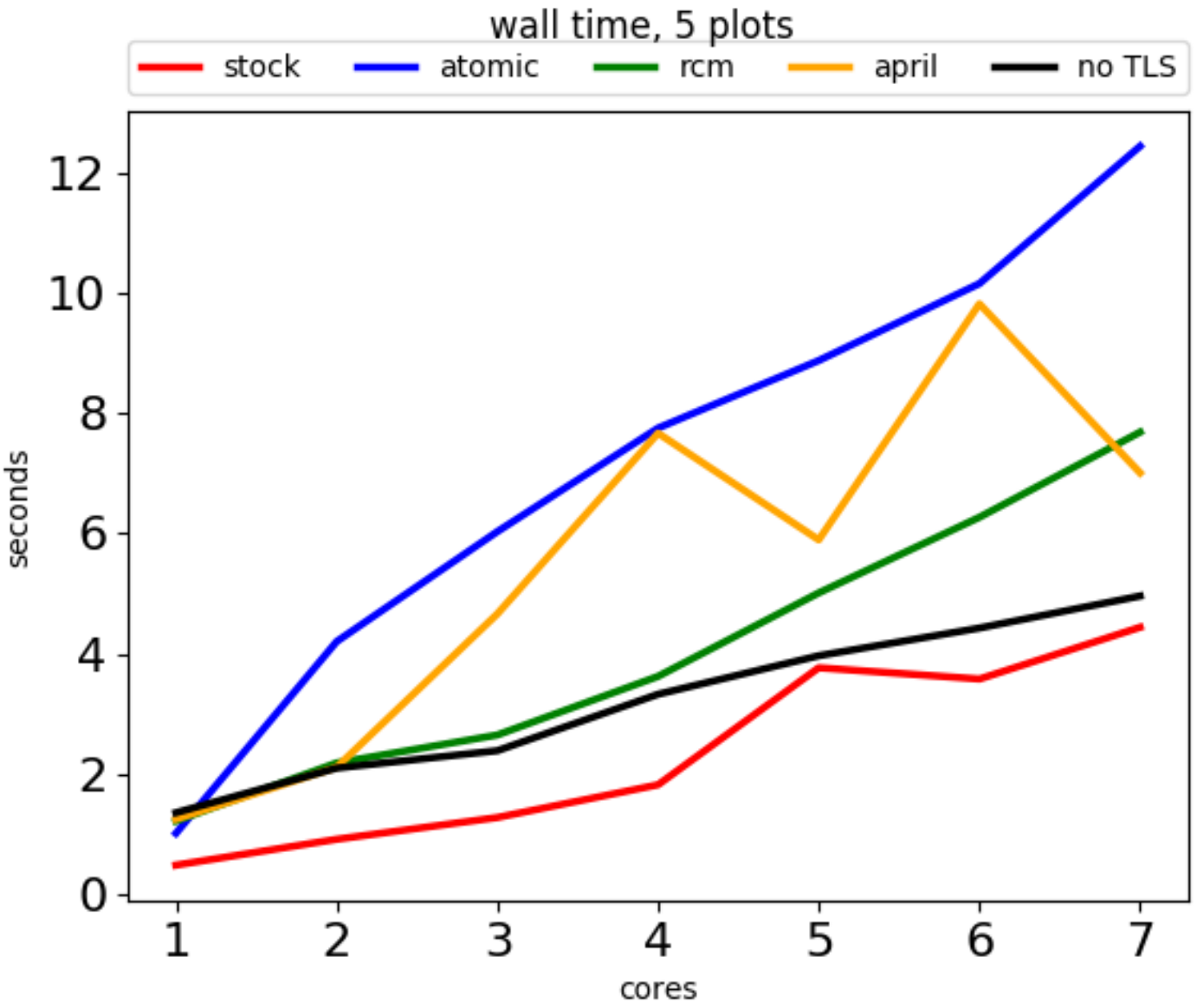
```
static PyObject *
PyEval_EvalFrameEx2(tstate, …)
{
…
}

static PyObject *
PyEval_EvalFrameEx(…)
{
return PyEval_EvalFrameEx2(PyThreadState_GET(),
…);
}
```

cpu time
may 2017



wall time
may 2017



next?

per-thread obmalloc (usedpools)

other experiments to try

private locking

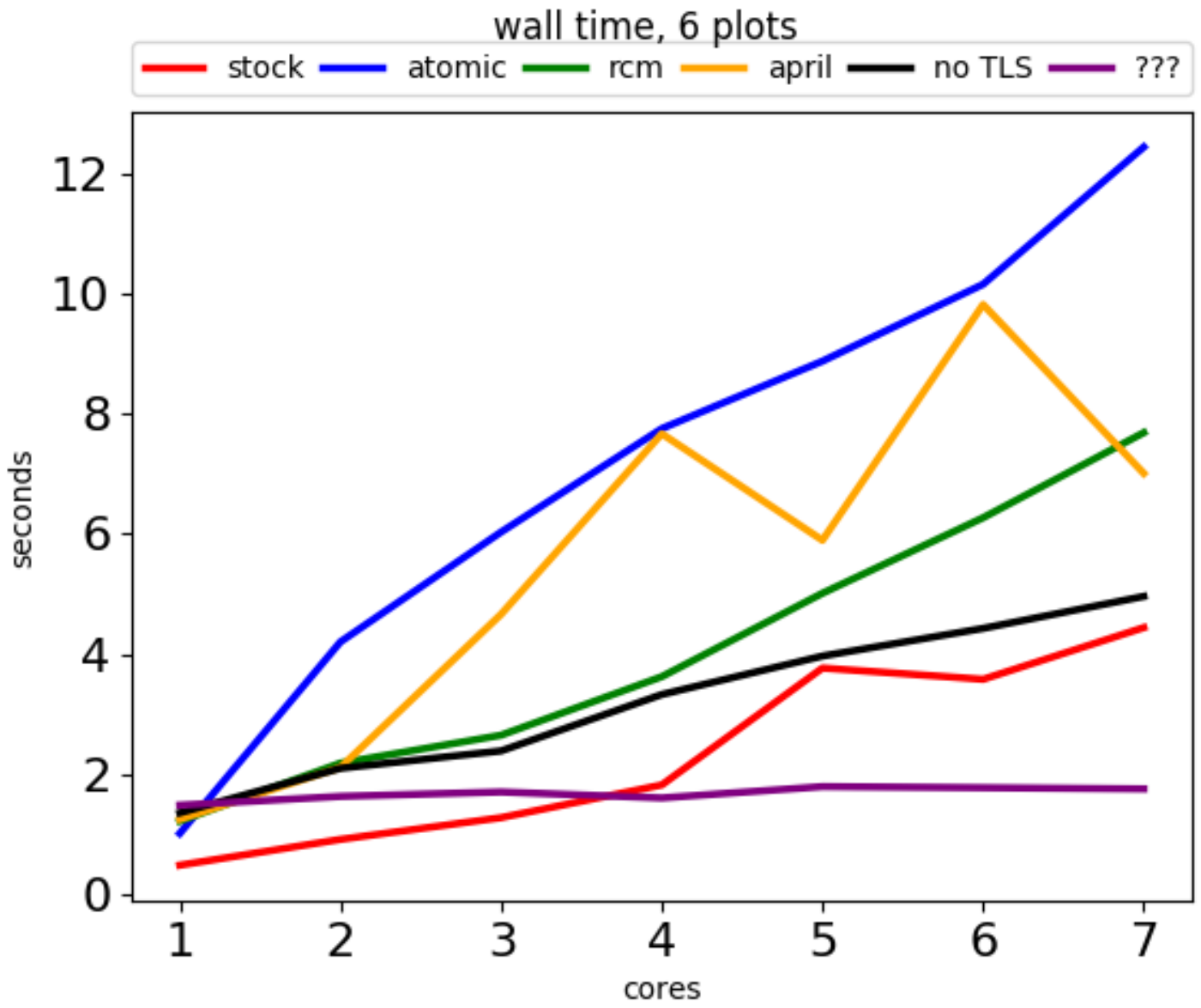
store refcnt outside object

crazy rewrite

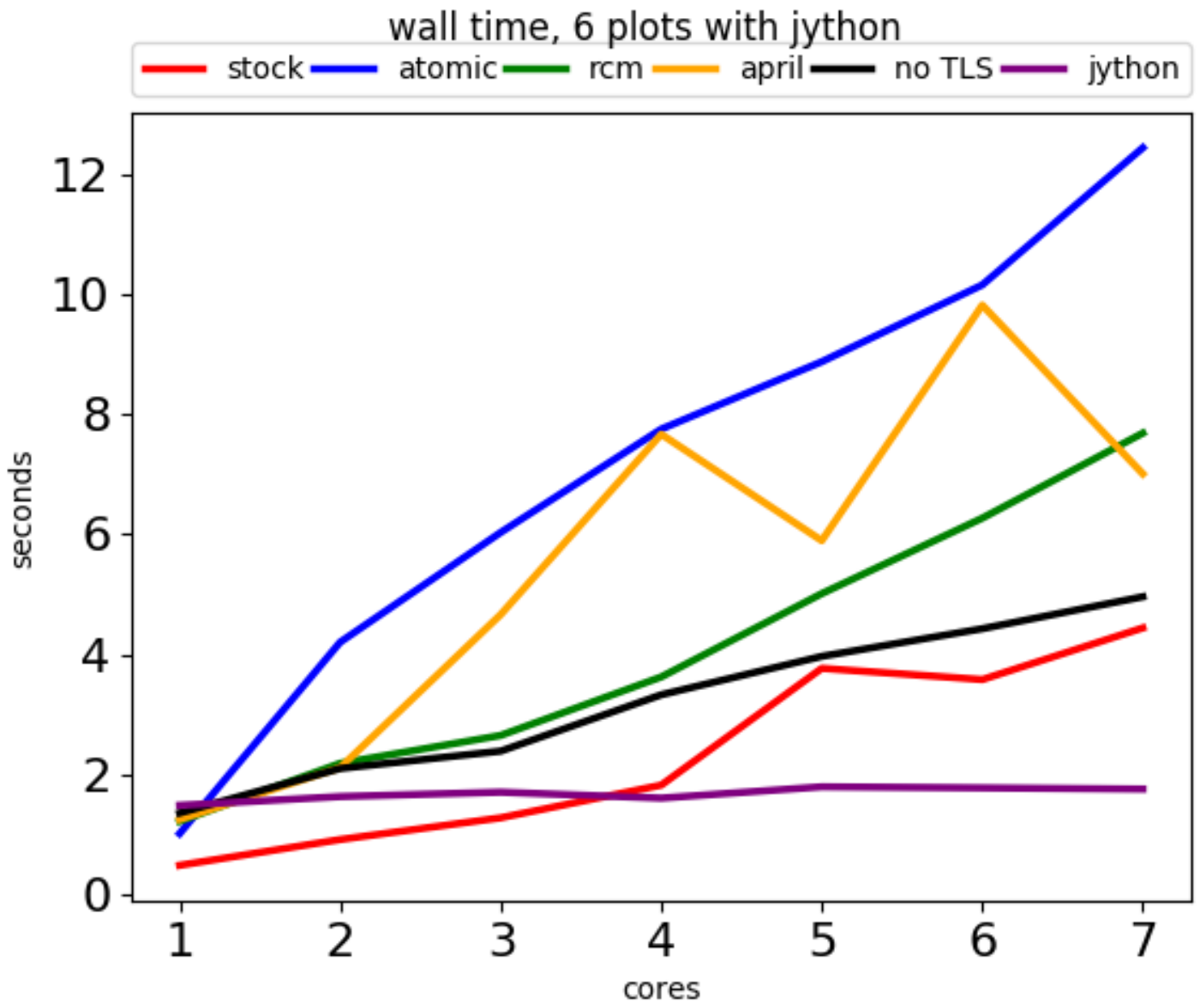
tracing garbage collection

cpyext for cpython

wall time
what we want



jython


wall time, 6 plots with jython
stock
atomic
rcm
april
no TLS
jython
seconds
cores
0
2
4
6
8
10
12
1
2
3
4
5
6
7

existence proofs

jython

ironpython

the question

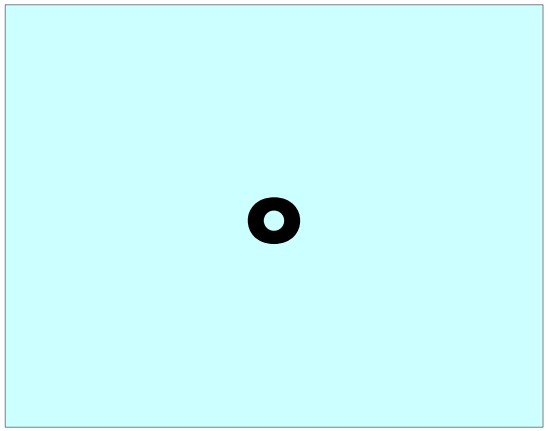
~~will it work?~~

how much does the c api have to break?

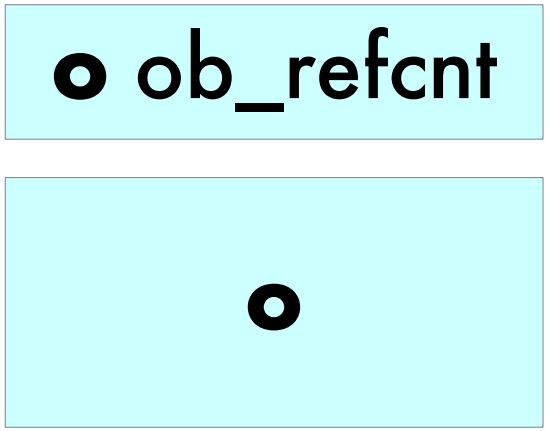
github.com/larryhastings/gilectomy

#gilectomy

remote object headers

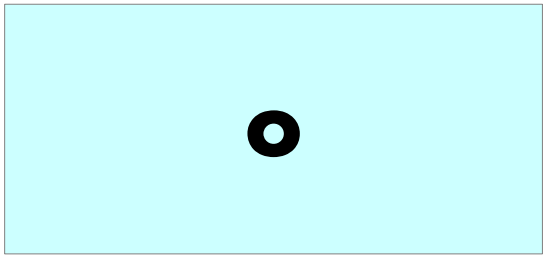


remote object headers



remote object headers

- ob_refcnt



github.com/larryhastings/gilectomy

#gilectomy