Lets Talk About GIL

@aktech

Who am I?

Loves everything which is free and open!
including Software

@Work

Process

A process is an instance of a program in execution.

Thread

A thread is a single sequence stream within in a process.

Threads

light weight and share memory.

Strength of Threads

shared state

“Everyone has everything”

Weakness of Threads

shared state

"Everyone can access everything"

Weakness of Threads

shared state

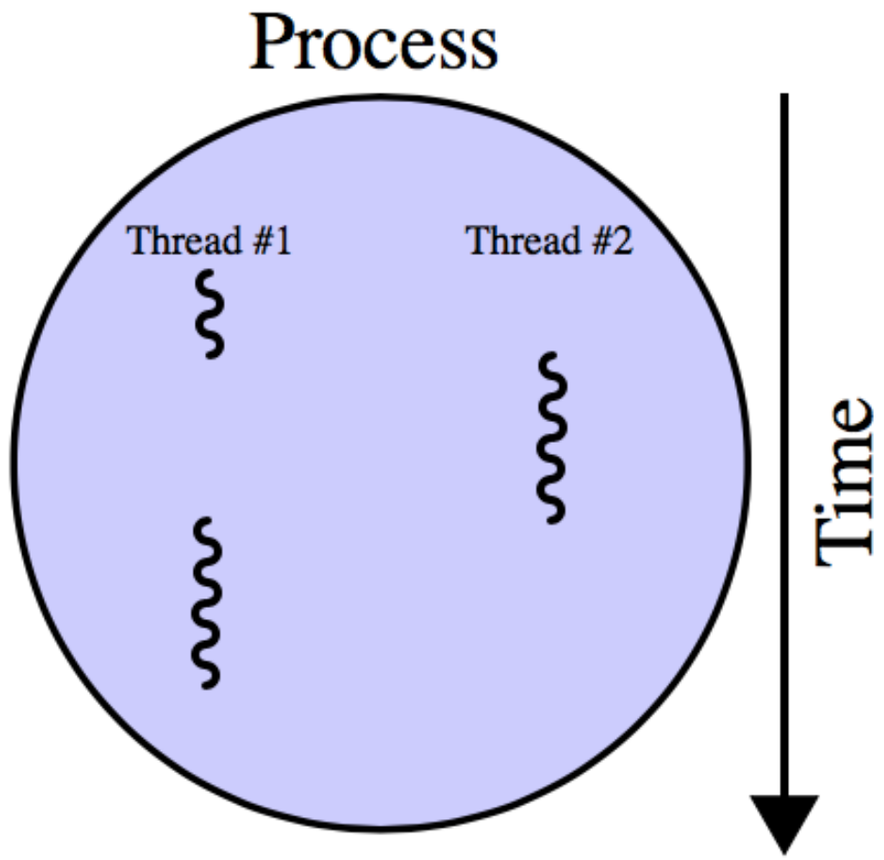
"Everyone can access everything"

Simultaneously!

Multithreading

The ability of a central processing unit (CPU) or a single core in a multi-core processor to execute **multiple processes** or **threads concurrently**, appropriately supported by the operating system.

Multithreading



Why Multithreading?

Why Multithreading?

- Keep a Process Responsive

Why Multithreading?

- Keep a Process Responsive
- Keep a Processor Busy

Why Multithreading?

- Keep a Process Responsive
- Keep a Processor Busy
- Keep Multiple Processors Busy

Why Multithreading?

- Keep a Process Responsive
- Keep a Processor Busy
- Keep Multiple Processors Busy
- Save Time

Multithreading in Python

Threading module in Python

Defining by Class

```
import time
import threading

class CustomThread(threading.Thread):
    def __init__(self, **kwargs):
        threading.Thread.__init__(self)
        self.param = kwargs.get('foo')

    def run(self):
        # This code executes in the Thread
```

Threading module in Python

functions as threads

```
def countdown(count):
      while count > 0:
          count -= 1
          time.sleep(5)

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
```

Lets See An Example

A Trivial Example

Lets do all the work **without** Threading

```
TOTAL_WORK = 10000000

def countdown(count):
    while count > 0:
        count -= 1

start = time.time()
countdown(TOTAL_WORK)  # Single Thread Execution
print(end - start)
```

A Trivial Example

Lets do all the work **with** Threading

```
TOTAL_WORK = 10000000

def countdown(count):
     while count > 0:
          count -= 1

thread1 = threading.Thread(target=countdown, args=(TOTAL_WORK/2,))
thread2 = threading.Thread(target=countdown, args=(TOTAL_WORK/2,))

start = time.time()
thread1.start(); thread2.start()
thread1.join(); thread2.join()
end = time.time()

print(end - start)
```

Which one would be Faster?

All the work done sequentially ?

or

All the work divided in Two Threads?

All the work done sequentially took:
0.632690191269

All the work divided in Two Threads took:
0.91114282608

If **two people** divide a work, shouldn't it be faster than a **single person** doing all the work?

Lets Talk About GIL Now!

GIL?

Global Interpreter Lock

Global Interpreter Lock

- The GIL ensures that only one thread runs in the interpreter at once.
- So, any time a thread is forced to wait, other "ready" threads get their chance to run.
- Whenever a thread runs, it holds the GIL

Processes

- I/O Bound:

  processes which are associated with input/output based activity like reading from files,etc.

- CPU Bound

  processes which spends the majority of its time simply using the CPU (doing calculations)

GIL Behaviour

For I/O Bound:

GIL is released on blocking I/O

GIL Behaviour

For CPU Bound:

Interpreter periodically performs a “check”,
every 100 interpreter "ticks"

Before Python 3.2

Tick?

- Roughly stated, a tick corresponds to a **Python bytecode operation**.
- For the most part that's true, however there are certain bytecode instructions that do not qualify as whole ticks.
- Ticks are **uninterruptible**. Example(x in range(10^6))
- The interpreter will not thread switch in the middle of a tick.

Why GIL?

Why GIL?

- Simplified implementation
- Easy to write C Extensions
- No Deadlocks!
- Good for I/O Bound processes

Memory Management in Python

Reference Counting

Reference Counting

`Py_INCREF()`

`Py_DECREF()`

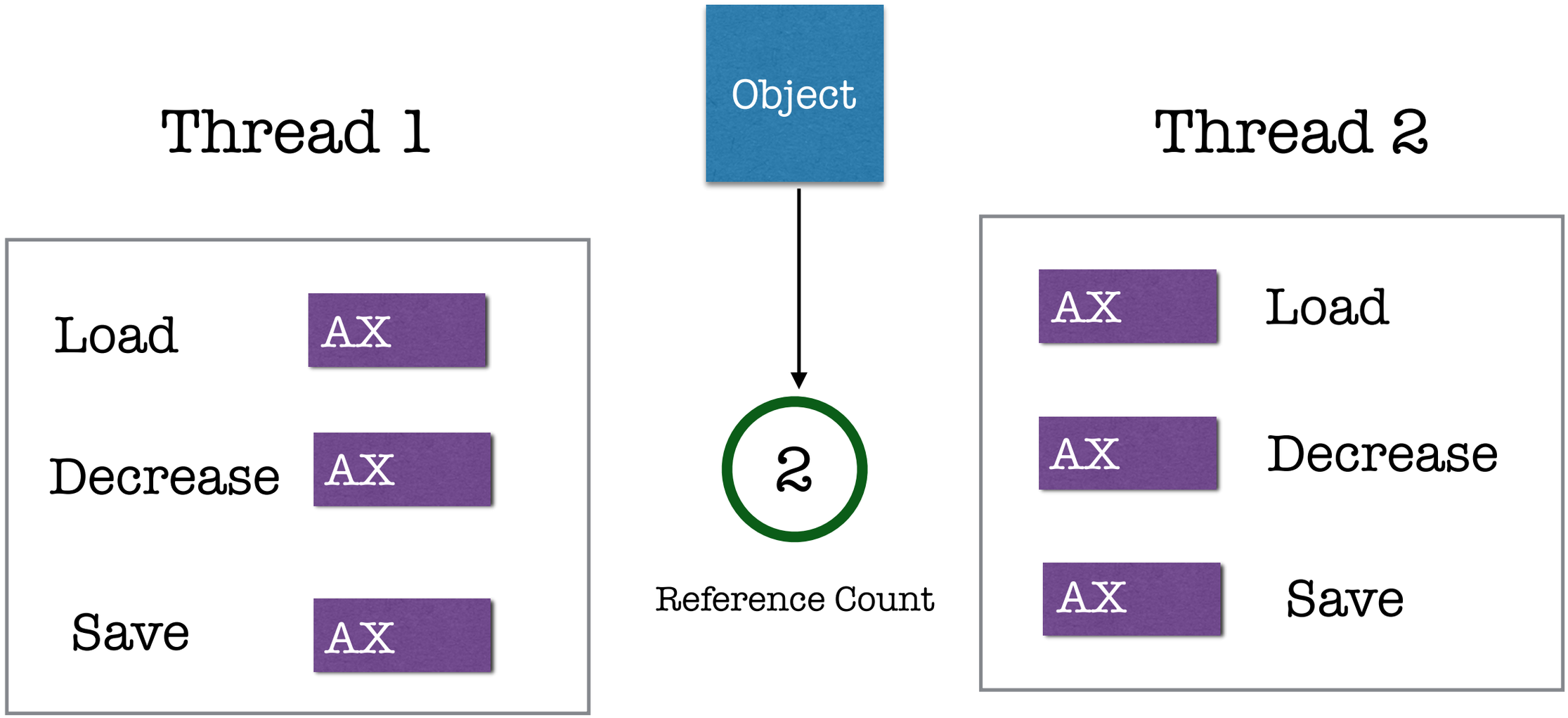
Methods in Python/C API

Reference Counting With Threads

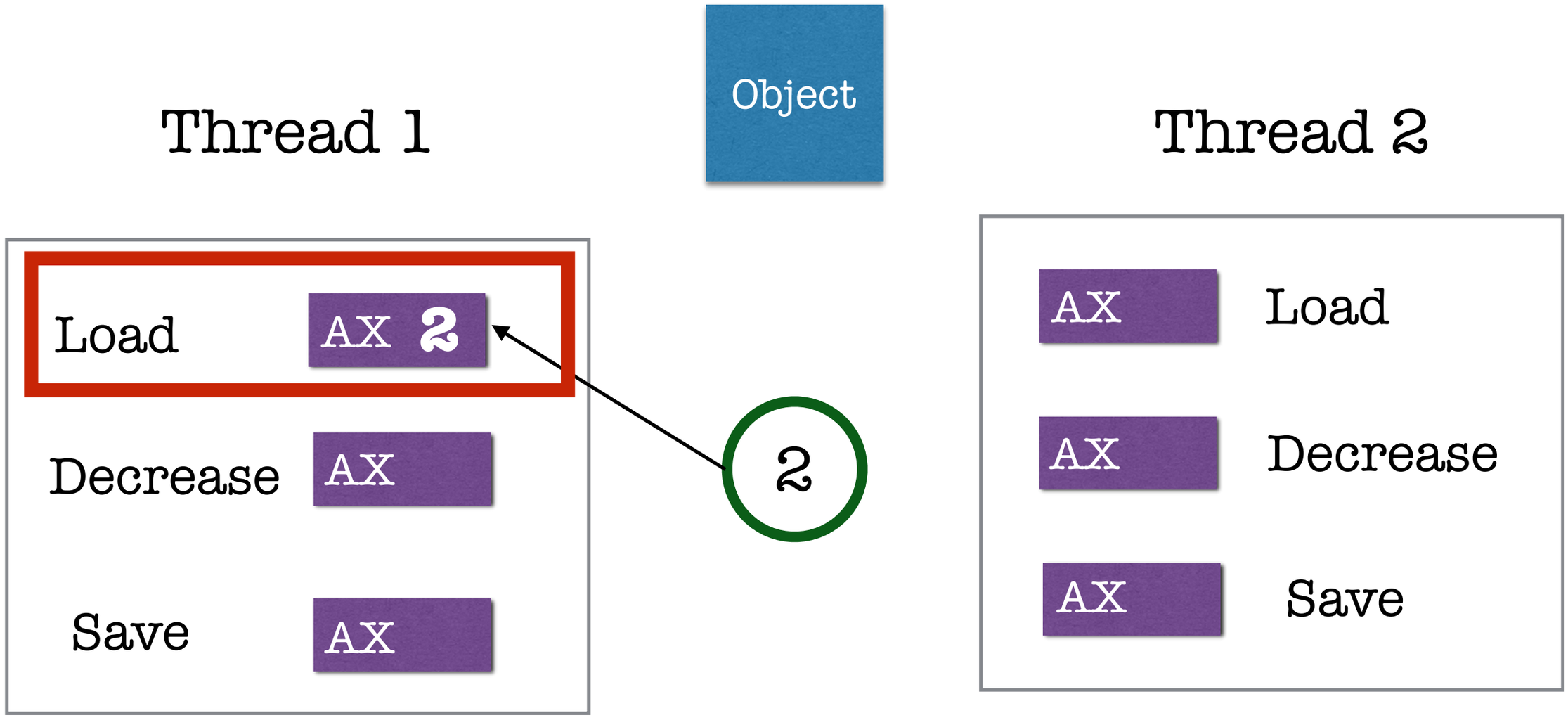
Lets See an Example!

Py_DECREF()

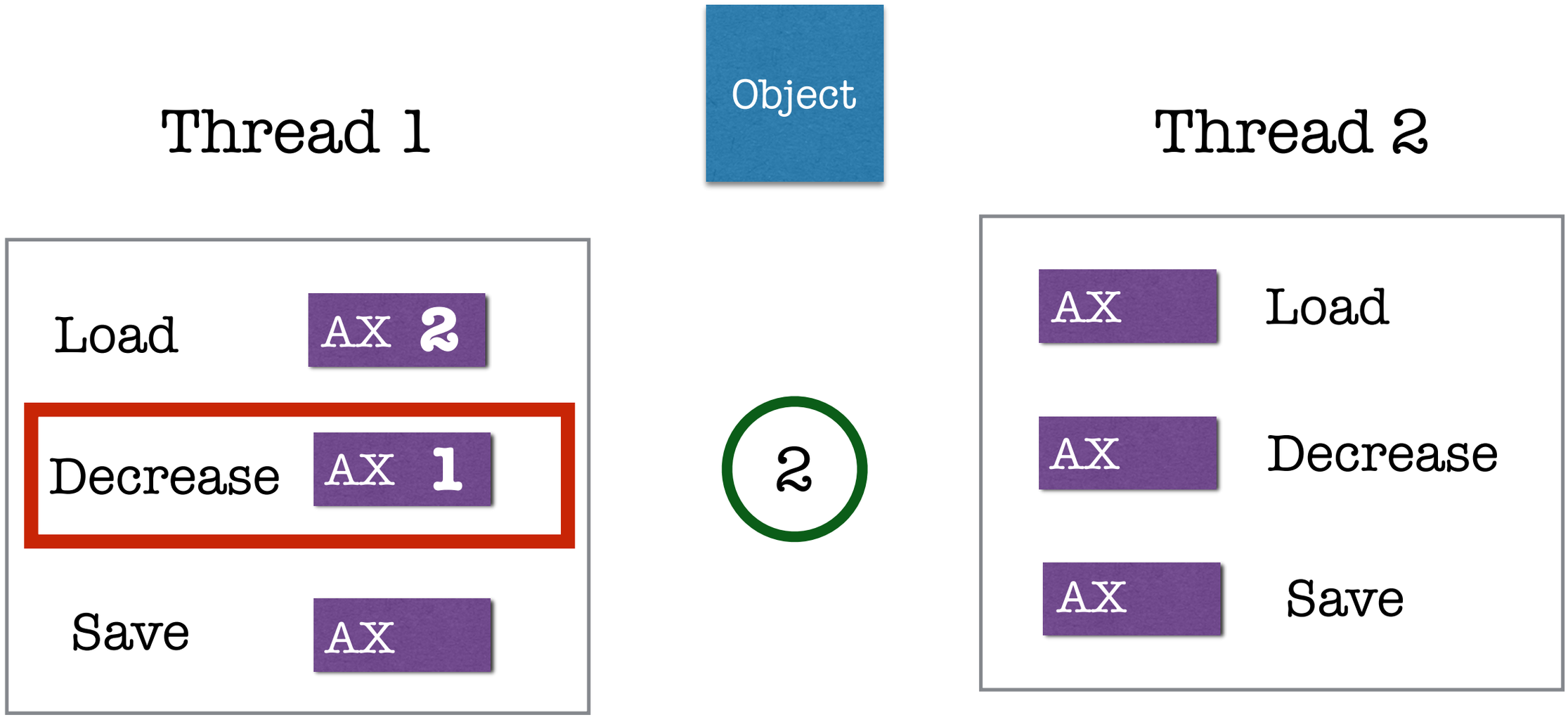
Reference Counting With Threads



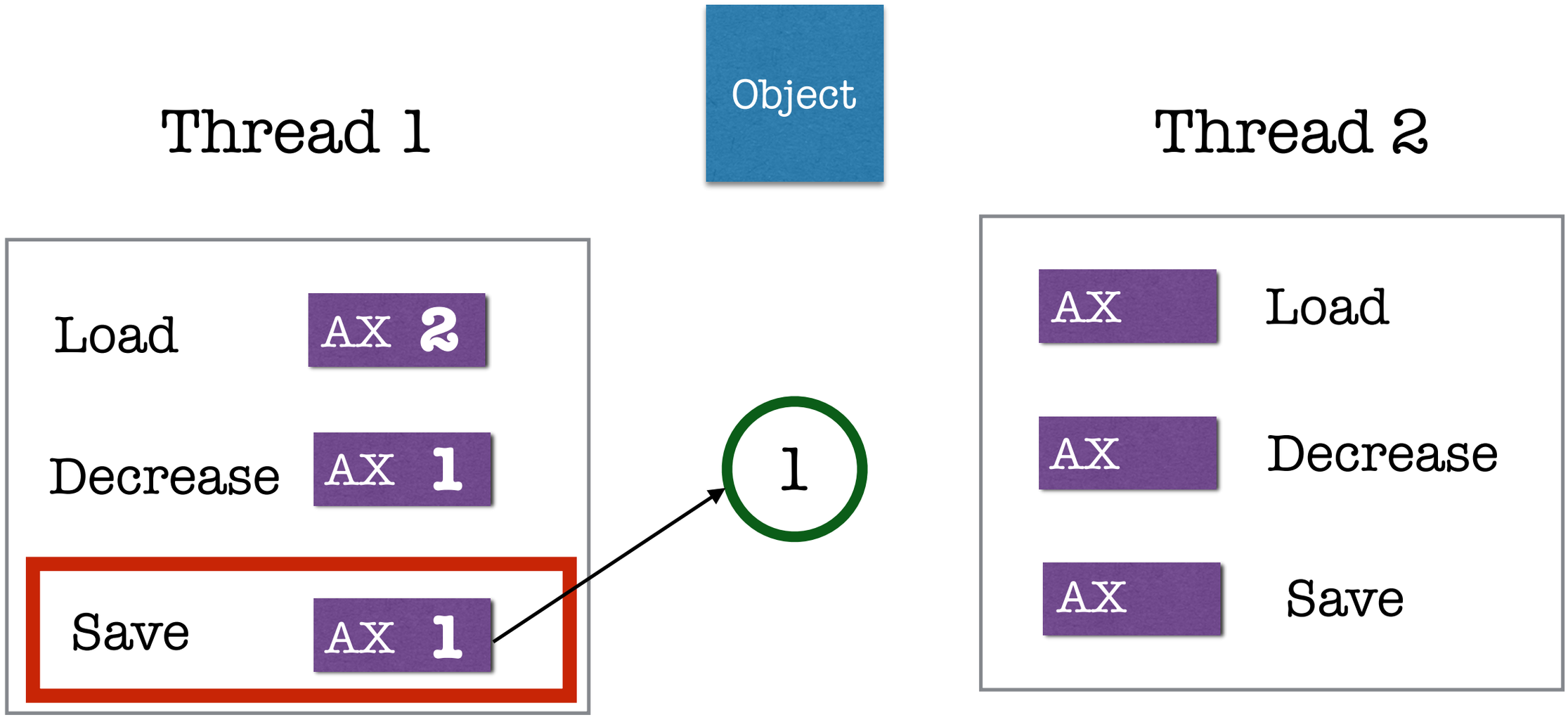
Reference Counting With Threads



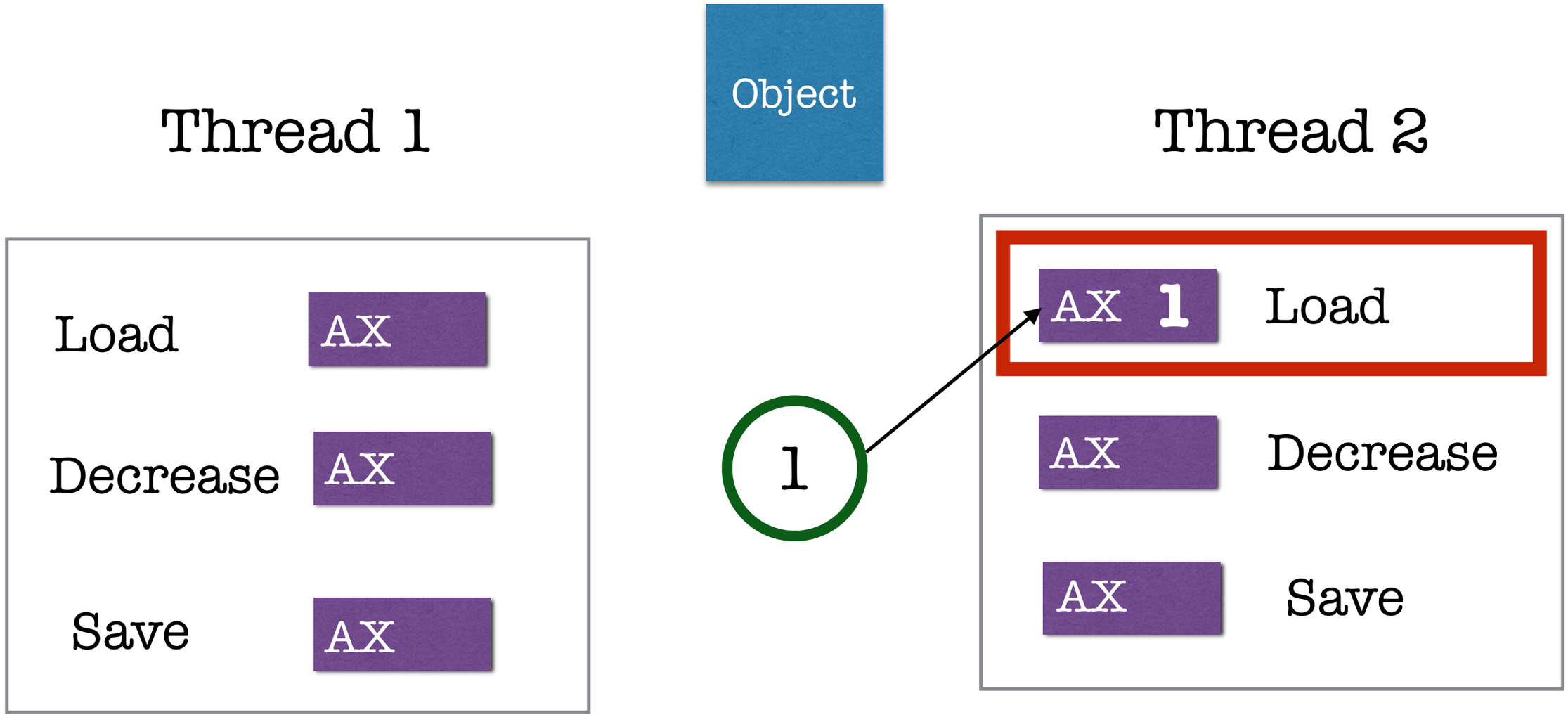
Reference Counting With Threads



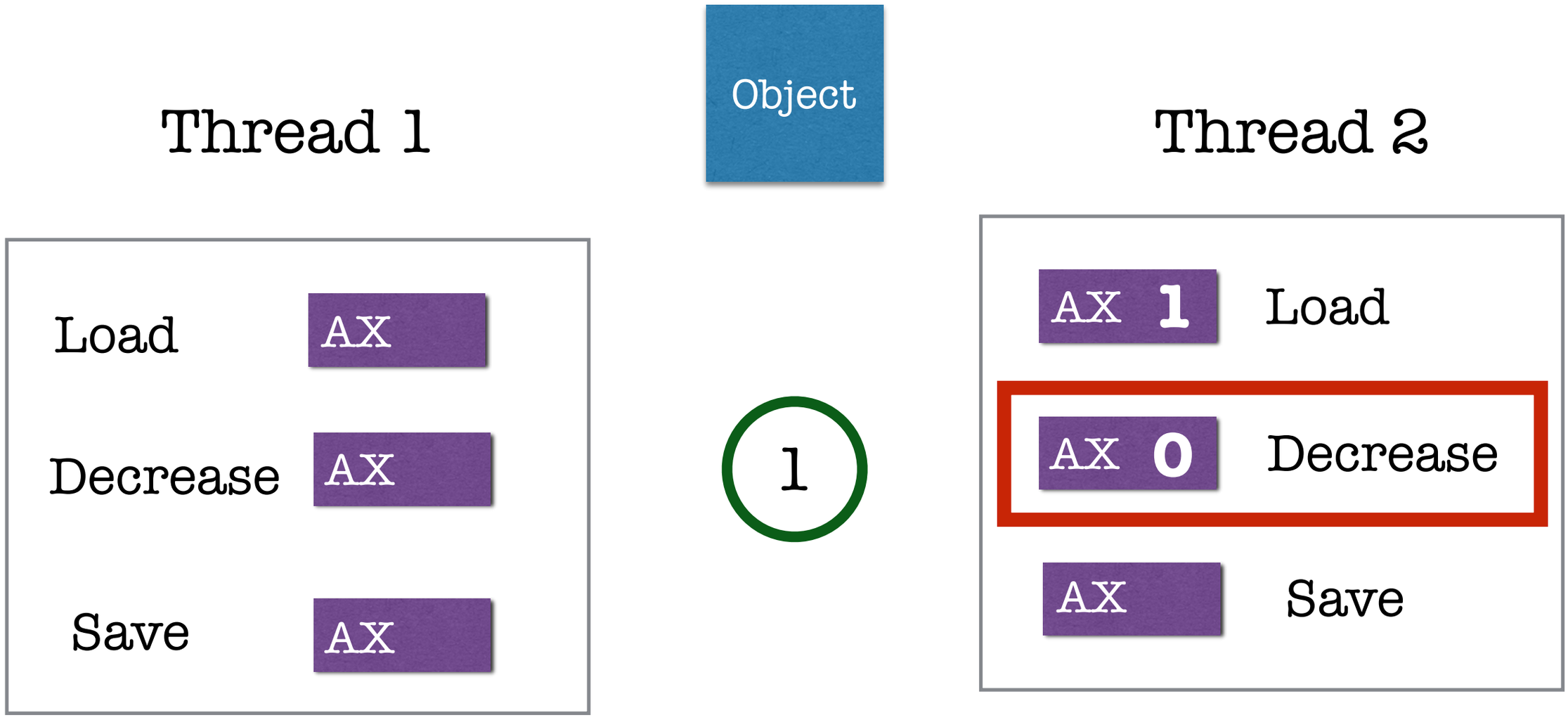
Reference Counting With Threads



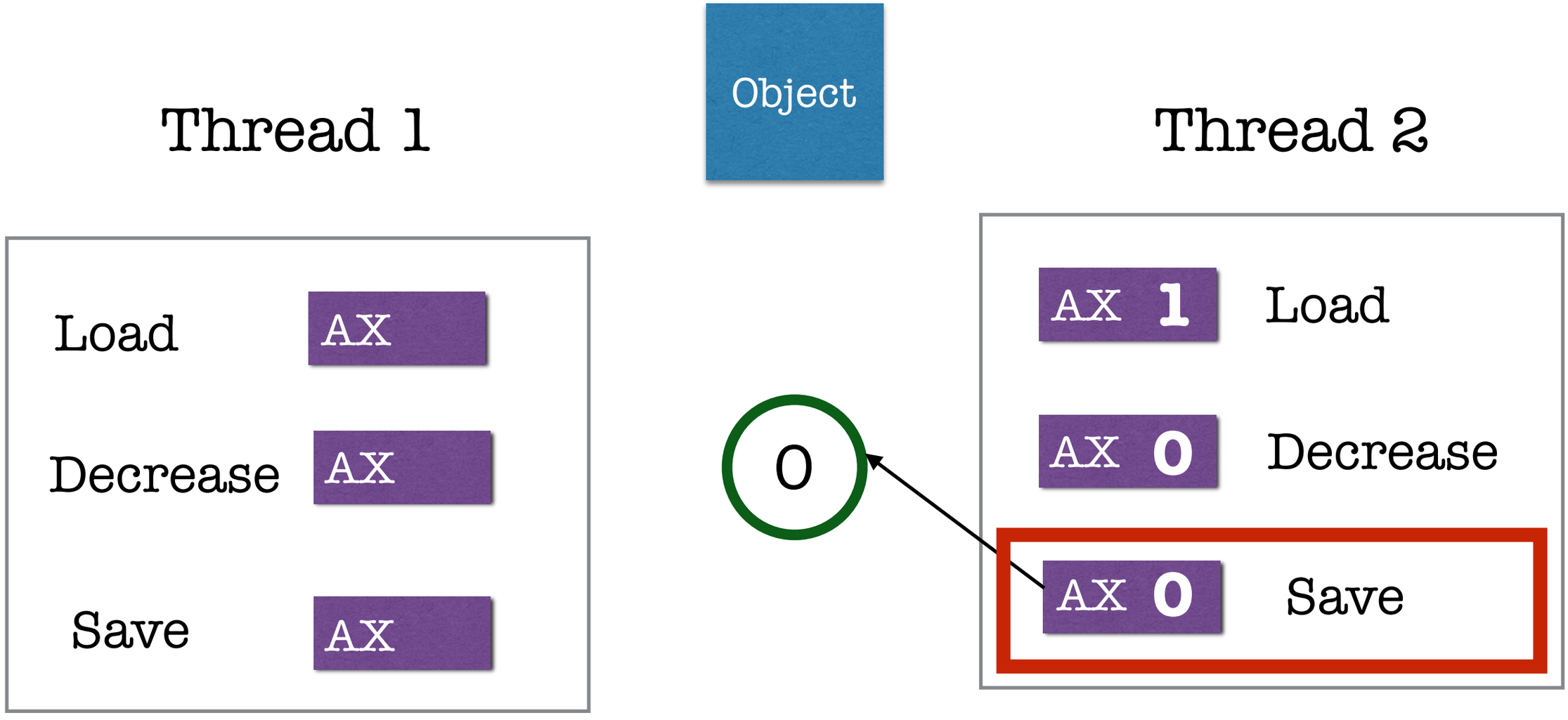
Reference Counting With Threads



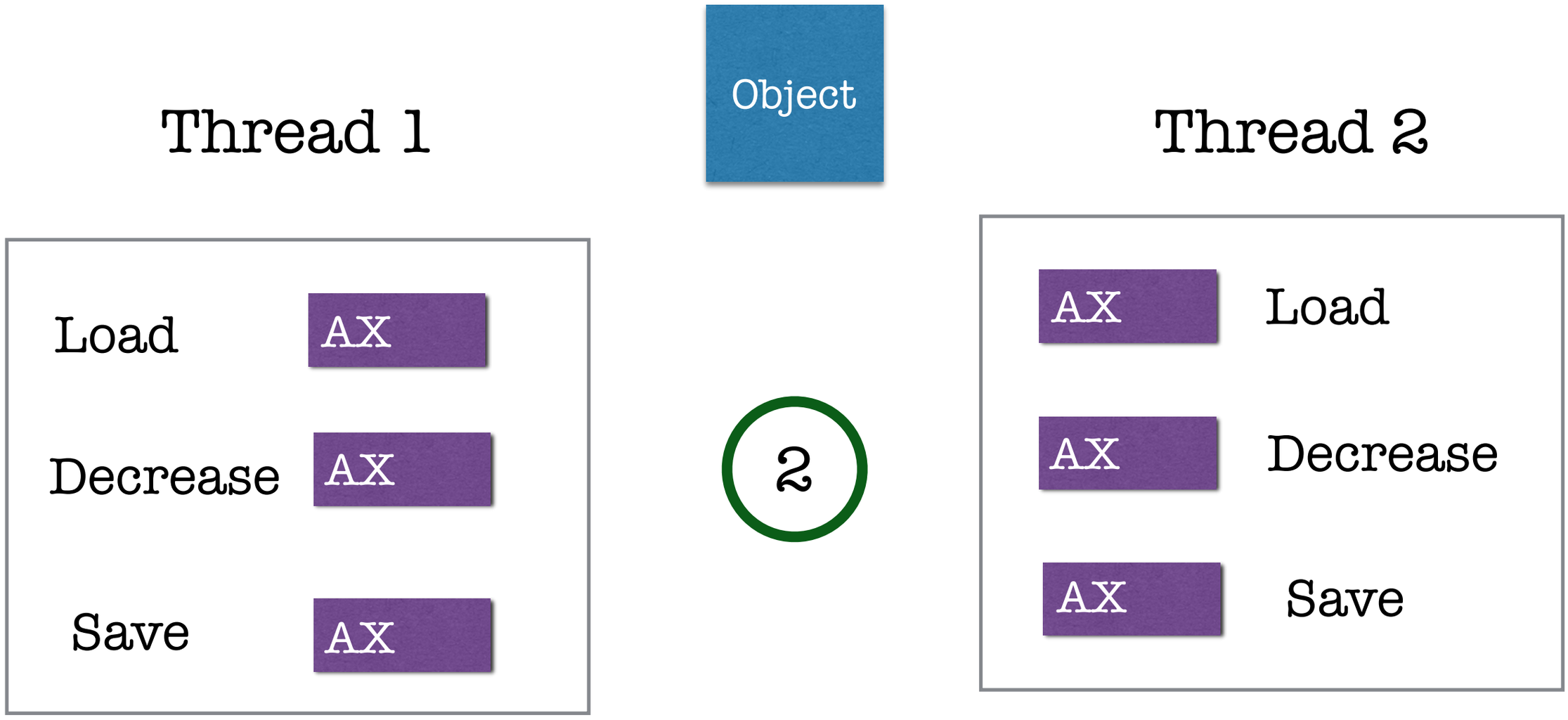
Reference Counting With Threads



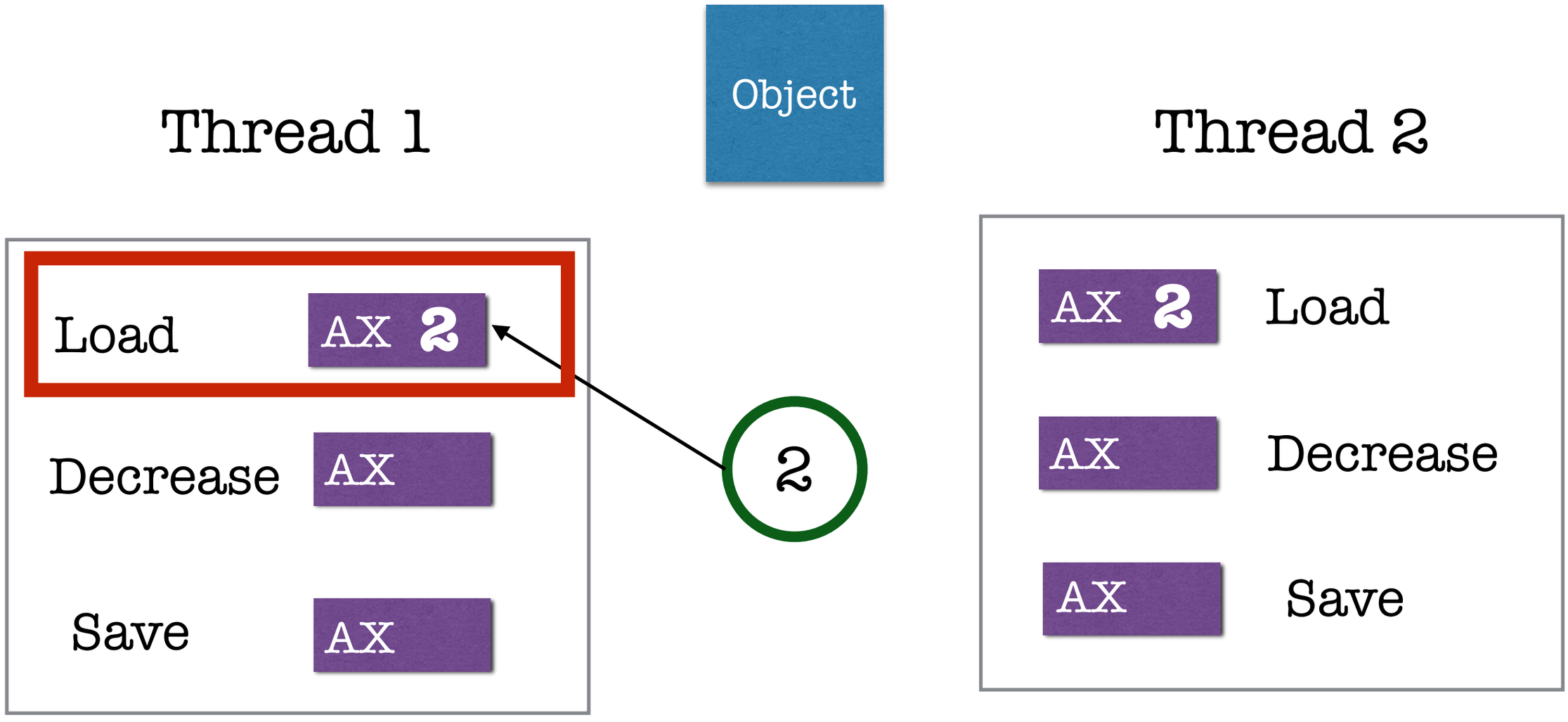
Reference Counting With Threads



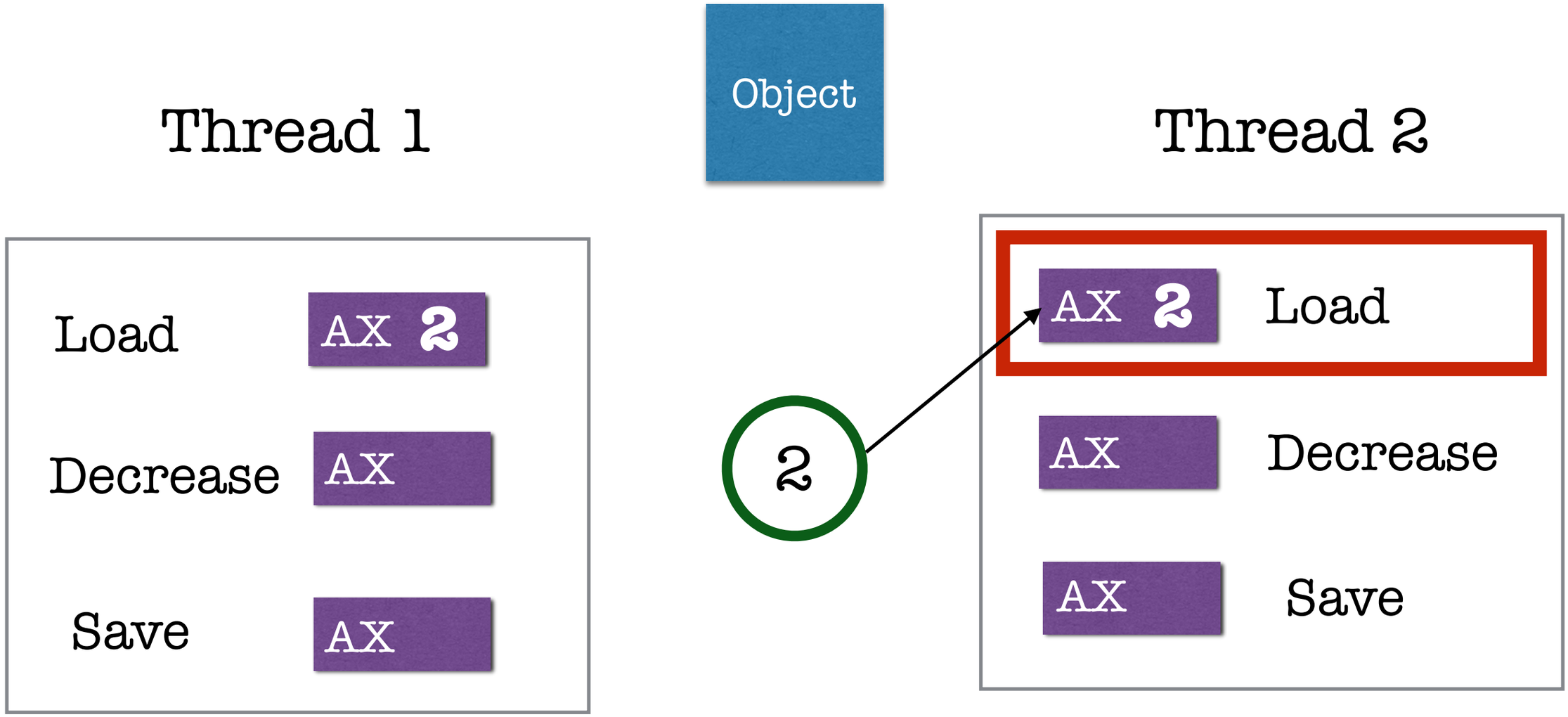
Reference Counting With Threads



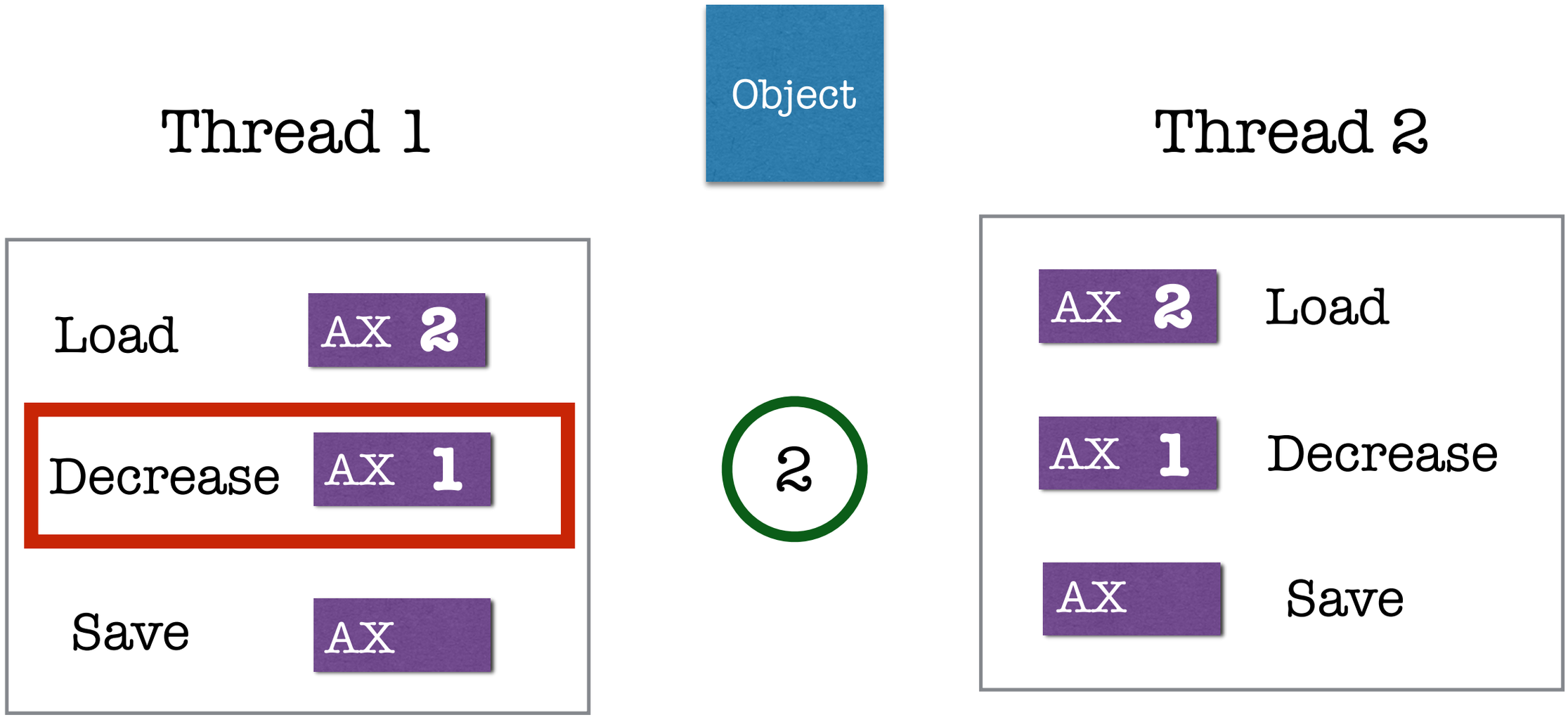
Reference Counting With Threads



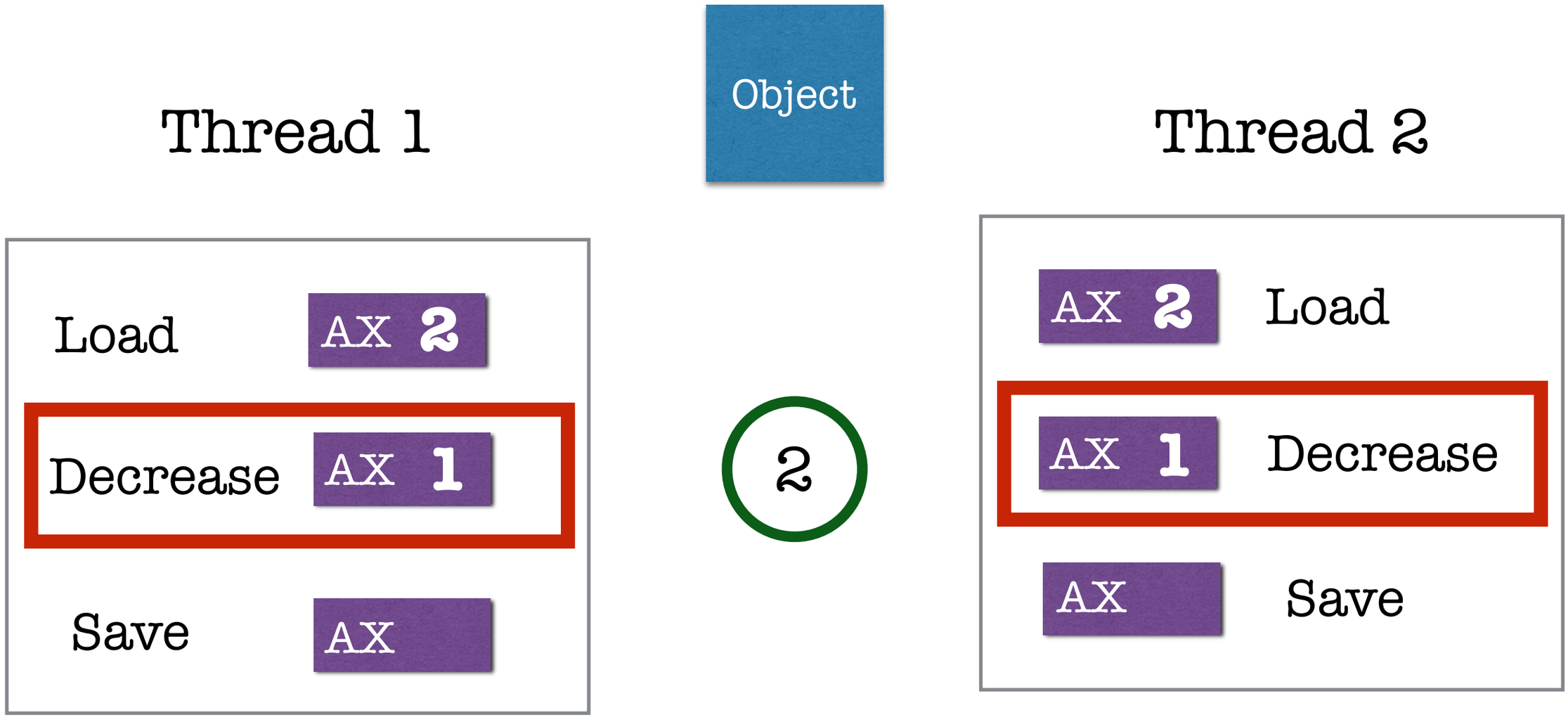
Reference Counting With Threads



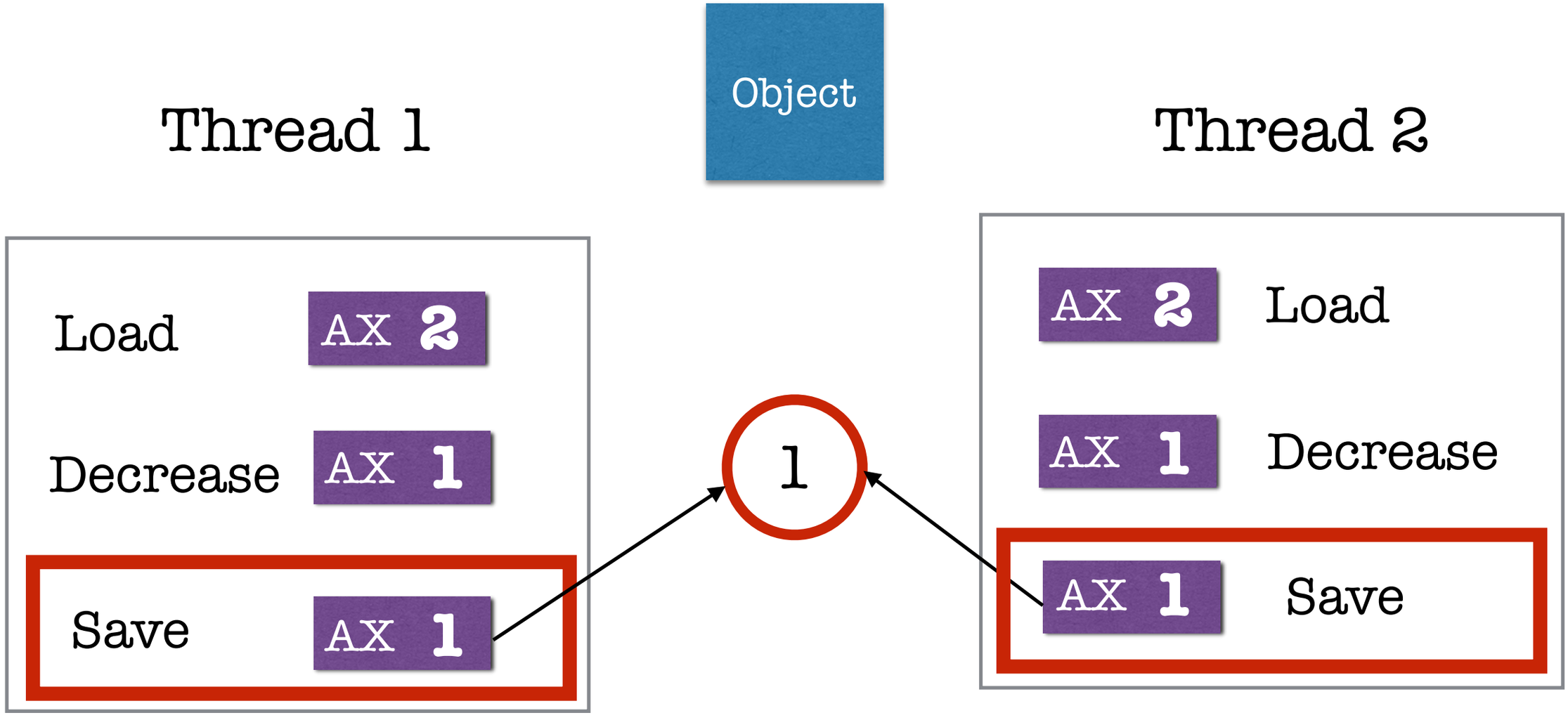
Reference Counting With Threads



Reference Counting With Threads

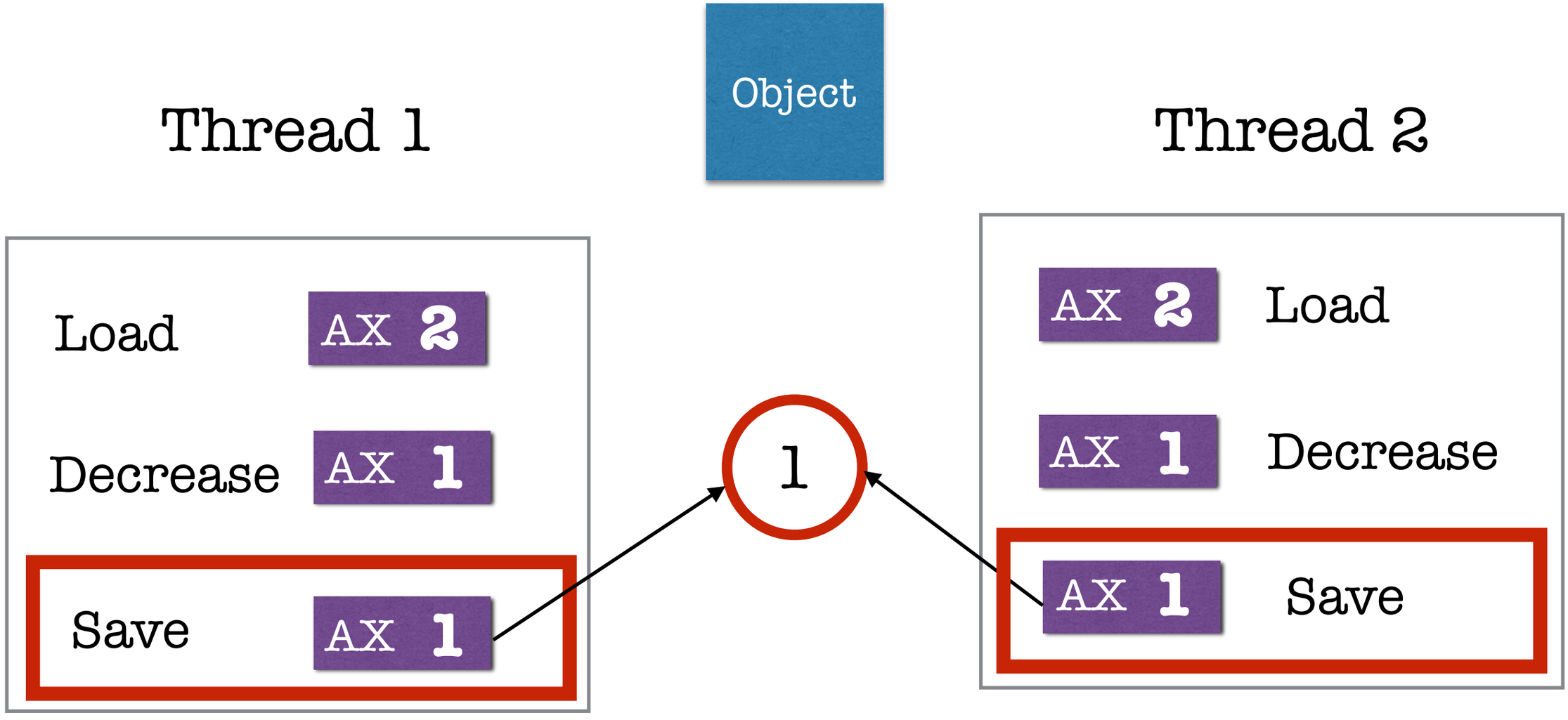


Reference Counting With Threads



Should have been 0, isn't it?

Reference Counting With Threads



Memory Leak!

Utilising Multiple Cores

alternative approaches

Utilising Multiple Cores

- Process based concurrency

Utilising Multiple Cores

- Process based concurrency
- C Extensions

Utilising Multiple Cores

- Process based concurrency
- C Extensions
- Cython

C-Extensions

Extending Python with C or C++

C-Extensions

Releasing the GIL from extension code

Save the thread state in a local variable.
Release the GIL
... Do some blocking I/O operation ...
Reacquire the GIL
Restore the thread state from the local variable.

C-Extensions

`Py_BEGIN_ALLOW_THREADS`

..Don't Talk to CPython Interpreter..

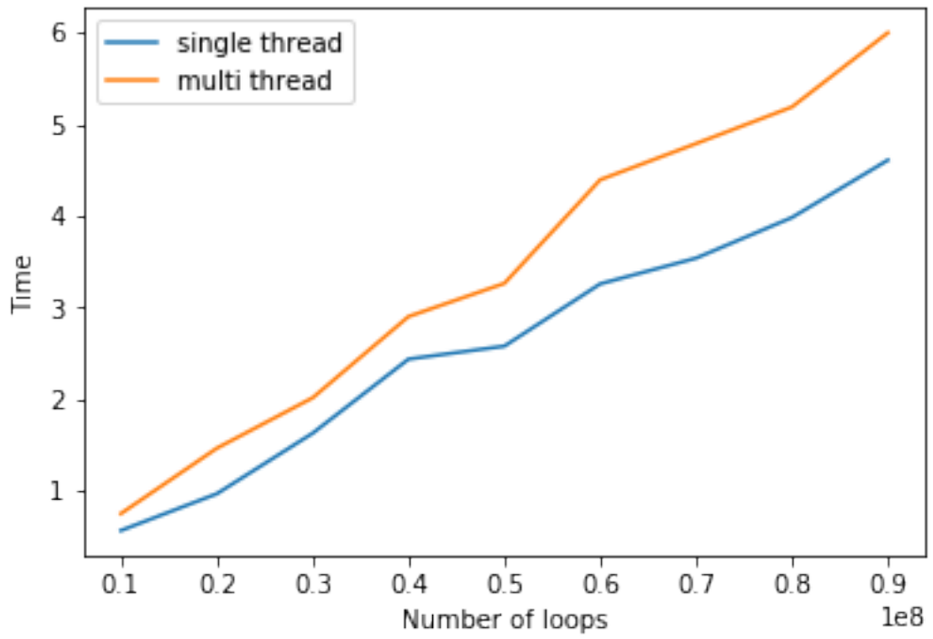
`Py_END_ALLOW_THREADS`

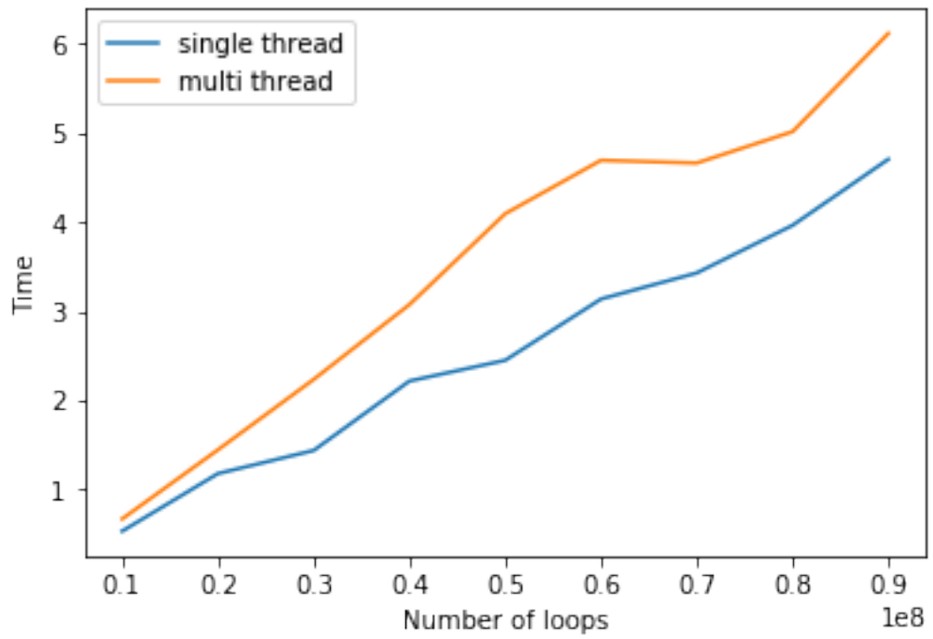
Threading in Python

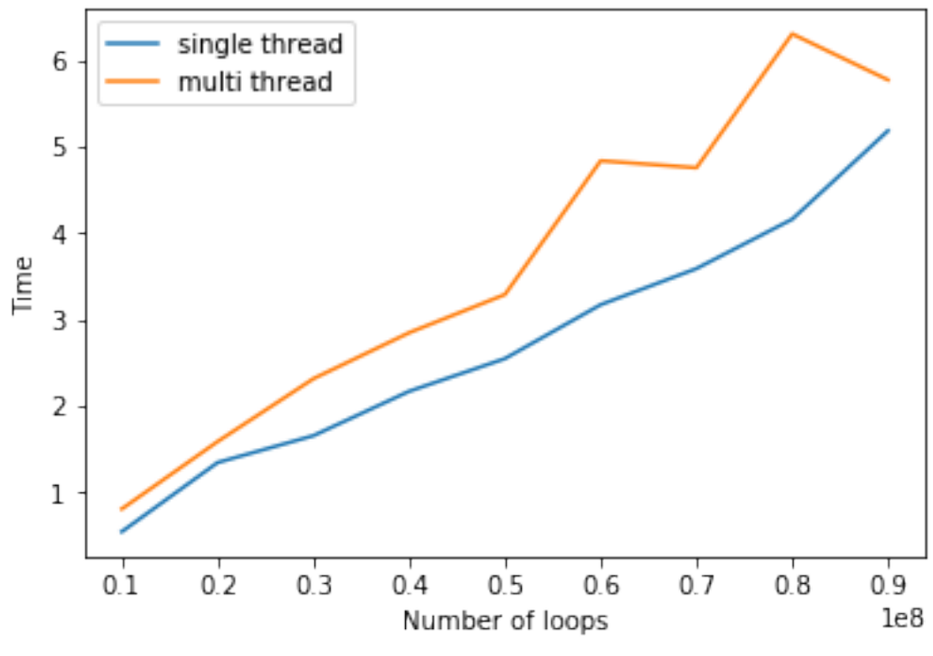
Lets see some Visualisations

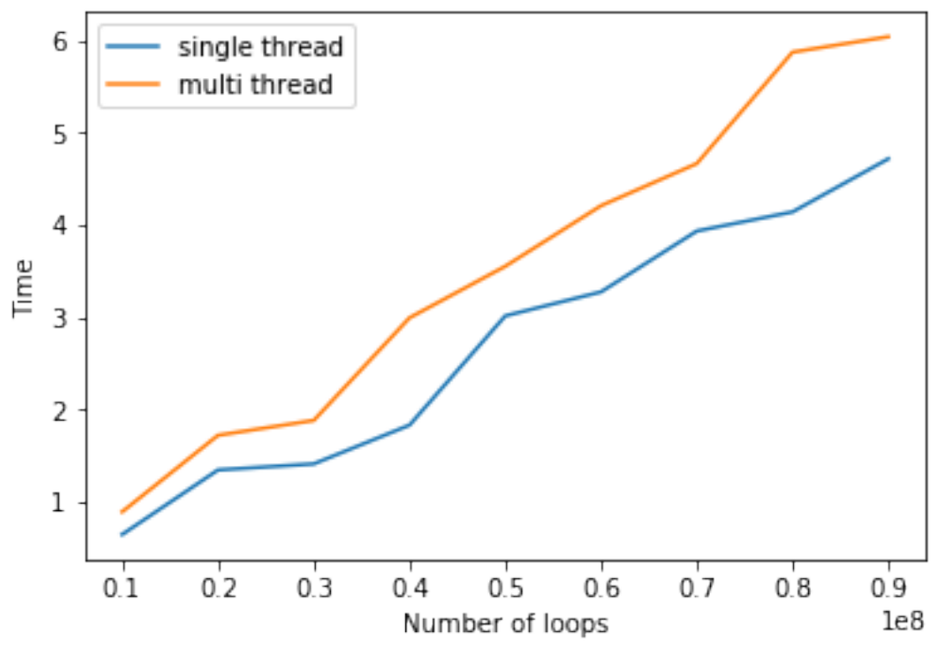
Benchmarked on:
MacBook Air (13-inch, Early 2015)
1.6 GHz Intel Core i5
4 GB 1600 MHz DDR3

Threading in Python







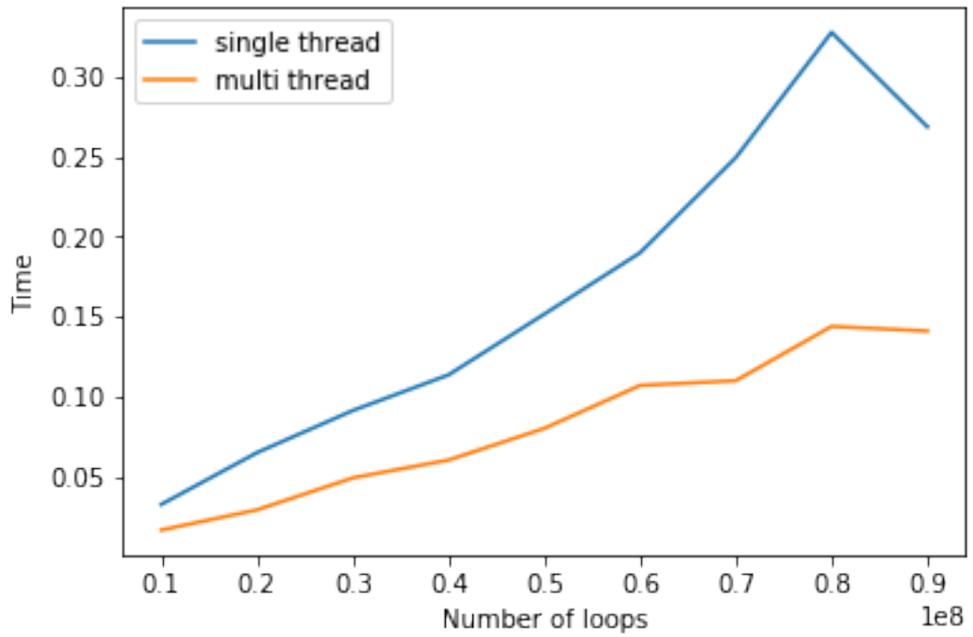


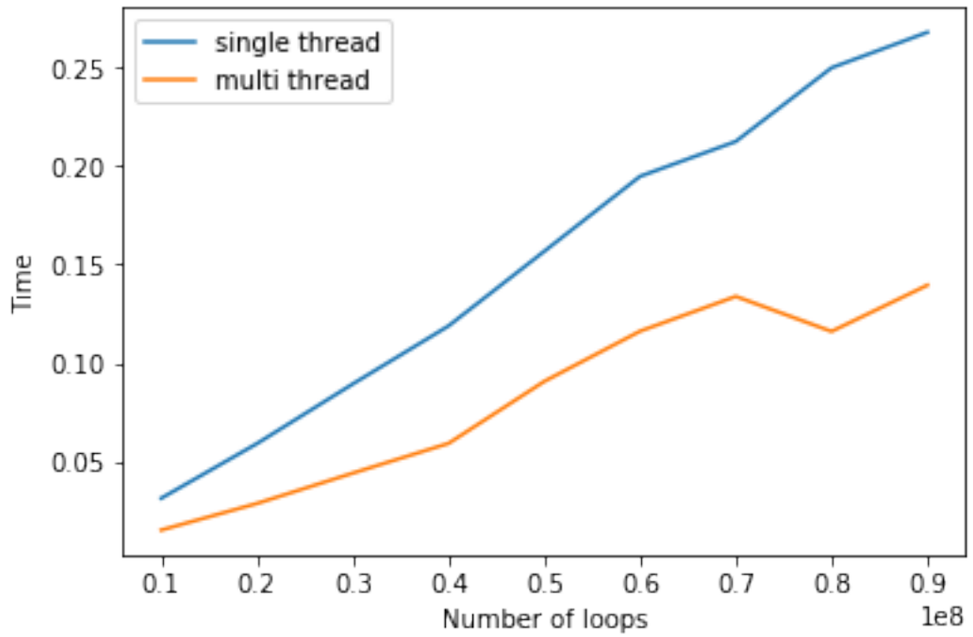
Threading with C-Extensions

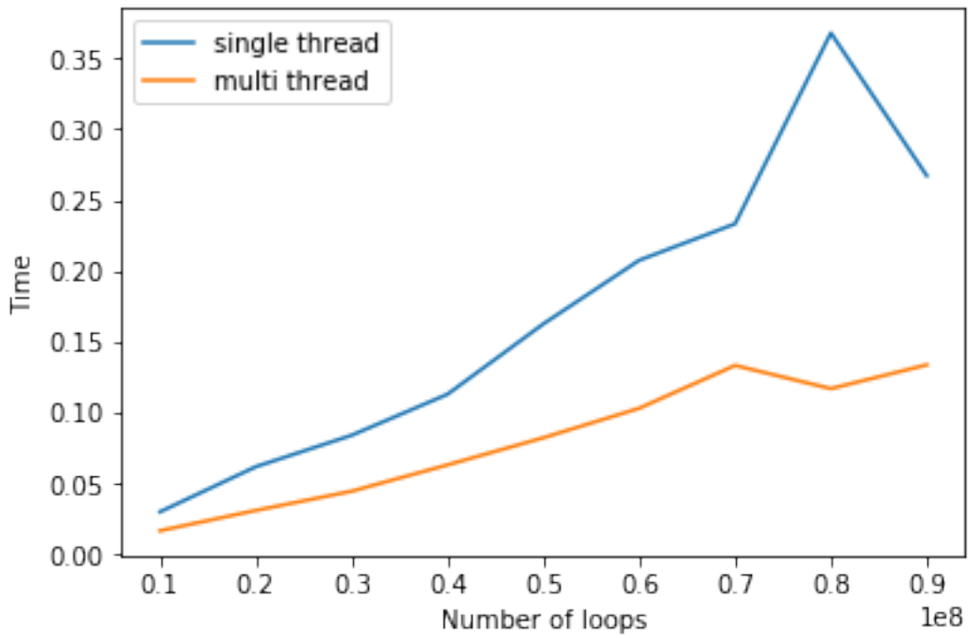
Lets see some Visualisations

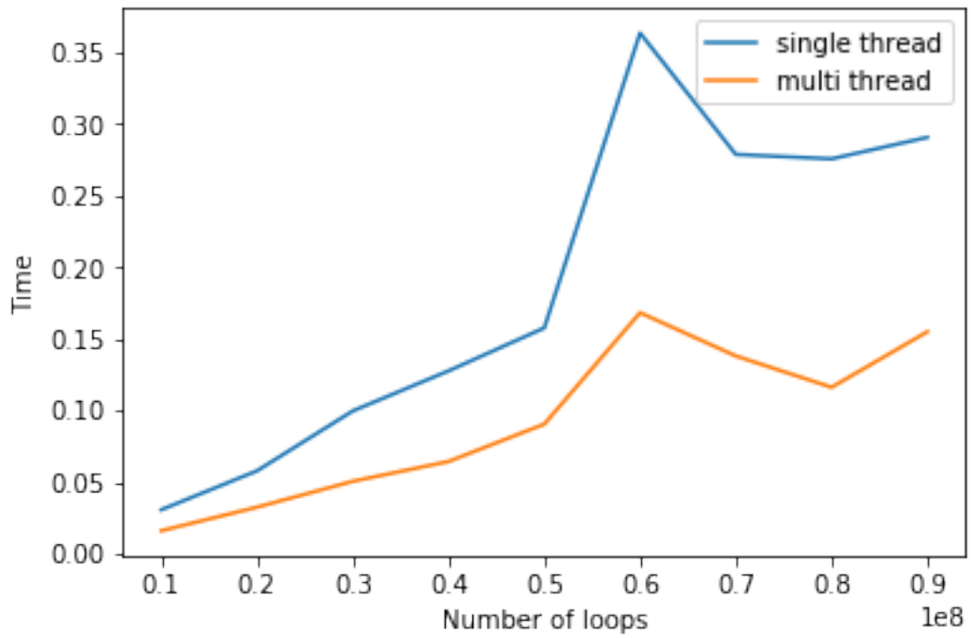
Benchmarked on:
MacBook Air (13-inch, Early 2015)
1.6 GHz Intel Core i5
4 GB 1600 MHz DDR3

Threading with C-Extensions









Guido on GIL

I'd welcome a set of patches into Py3k only if:

- performance for a **single-threaded** program and
- for a multi-threaded but **I/O-bound** program

does NOT decrease.

The Famous GIL Removal Patch

The Famous GIL Patch of Greg

Idea: Each thread has to isolate its interpreter state and not rely on C global variables.

- moved into a **per-thread data structure**.
- interpreter builds a **linked-list of all active threads.**
- patch introduces a **global reference-counting mutex lock**
- Mutable builtins such as lists and dicts need their **own locking** to synchronise modifications. T

The Famous GIL Patch of Greg

Idea: Each thread has to isolate its interpreter state and not rely on C global variables.

- moved into a **per-thread data structure**.
- interpreter builds a **linked-list of all active threads.**
- patch introduces a **global reference-counting mutex lock**
- Mutable builtins such as lists and dicts need their **own locking** to synchronise modifications. T

The Famous GIL Patch of Greg

patch made the performance of single-threaded applications much worse

The Famous GIL Patch of Greg

patch made the performance of single-threaded applications much worse

so much so that the patch couldn't be adopted.

The New Gil

The New Gil

by Antoine Pitrou

Since Python 3.2

The New Gil

Earlier:“ticks" based

Now: time based

The New Gil

Benefits:

- new GIL allows a thread to run for 5ms regardless of other threads
- Eliminates the Battle for GIL
- Eliminates Excessive Thrashing/Context Swithing

Questions?

Thank You!

@aktech