Concurrency with Python

Rochester's Python User Group

June 18th, 2013 Meeting

Overview

Concurrency

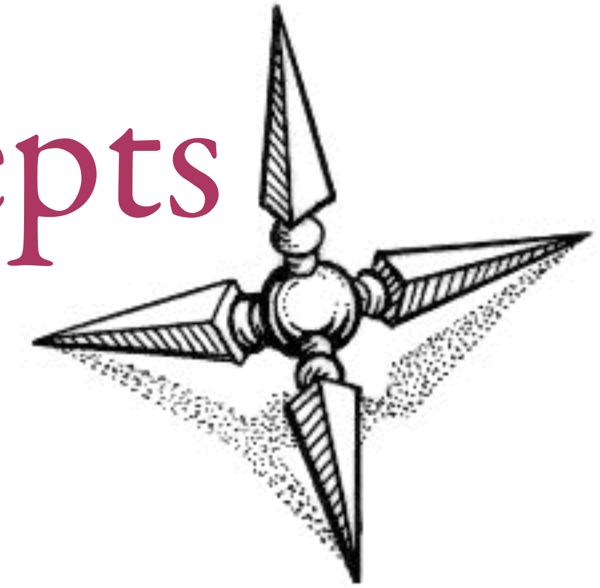
A property where multiple computations can be executed simultaneously.

- Code examples
  - https://github.com/RocPy/Topic-Concurrency

Expectations

- Concurrent techniques used with Python
  - We'll touch on CS topics of concurrency
  - Learn some Standard Library tools
  - A grand tour of all your options, with advantages and pitfalls.
- This is not proper instruction on concurrent programming and parallel computing.

Part 1: Concepts



Concurrency

- A Computer Science term, a property where multiple computations are executing simultaneously.
- There is potential each independent execution to interact with each other.
- Execution units can be multiple cores on a chip, multiple chips in a machine, or physically separated processes on different computer nodes.

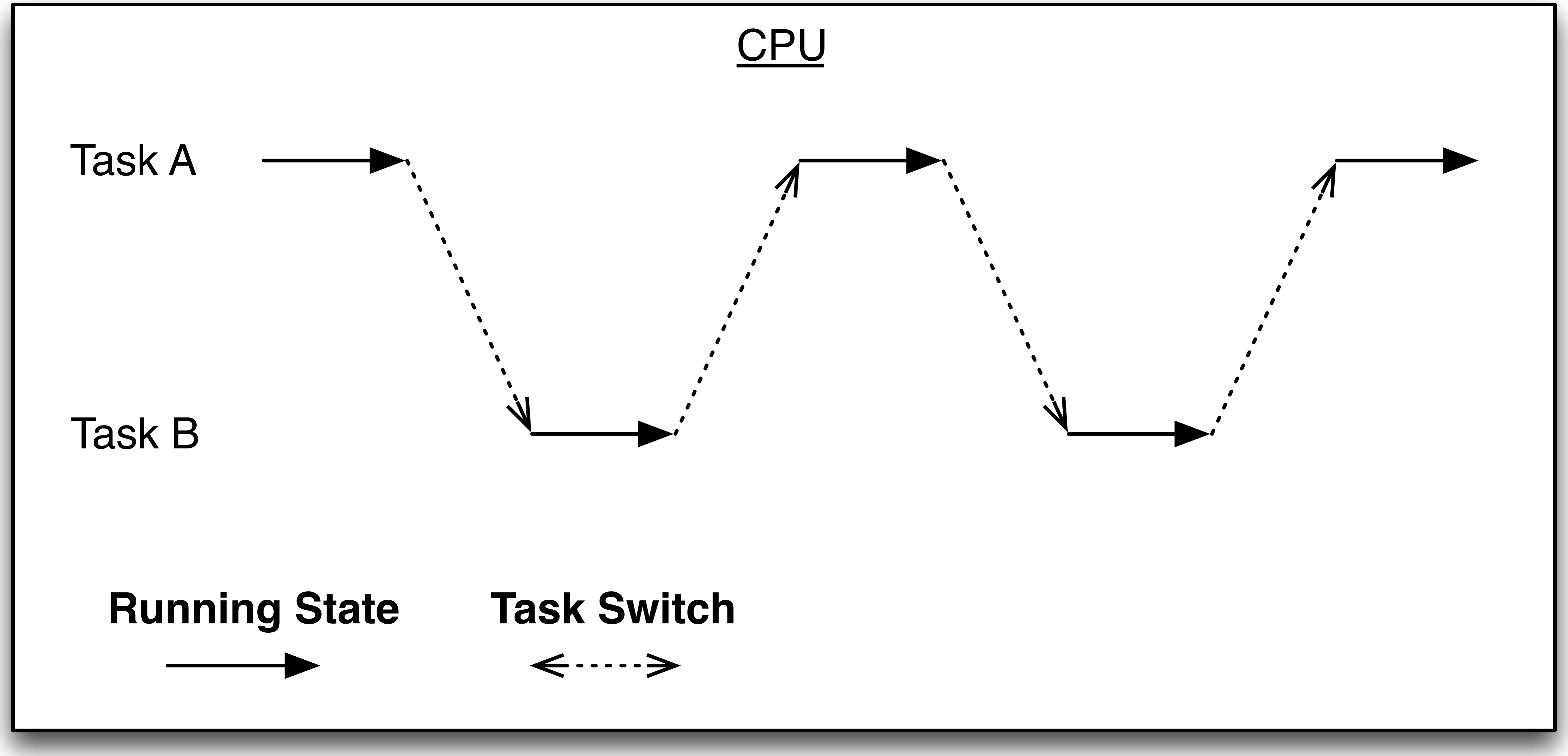
Task

- A set of program instructions loaded into an address space (memory) is a **Task**.
  - It can define processes, threads, kernels, etc.

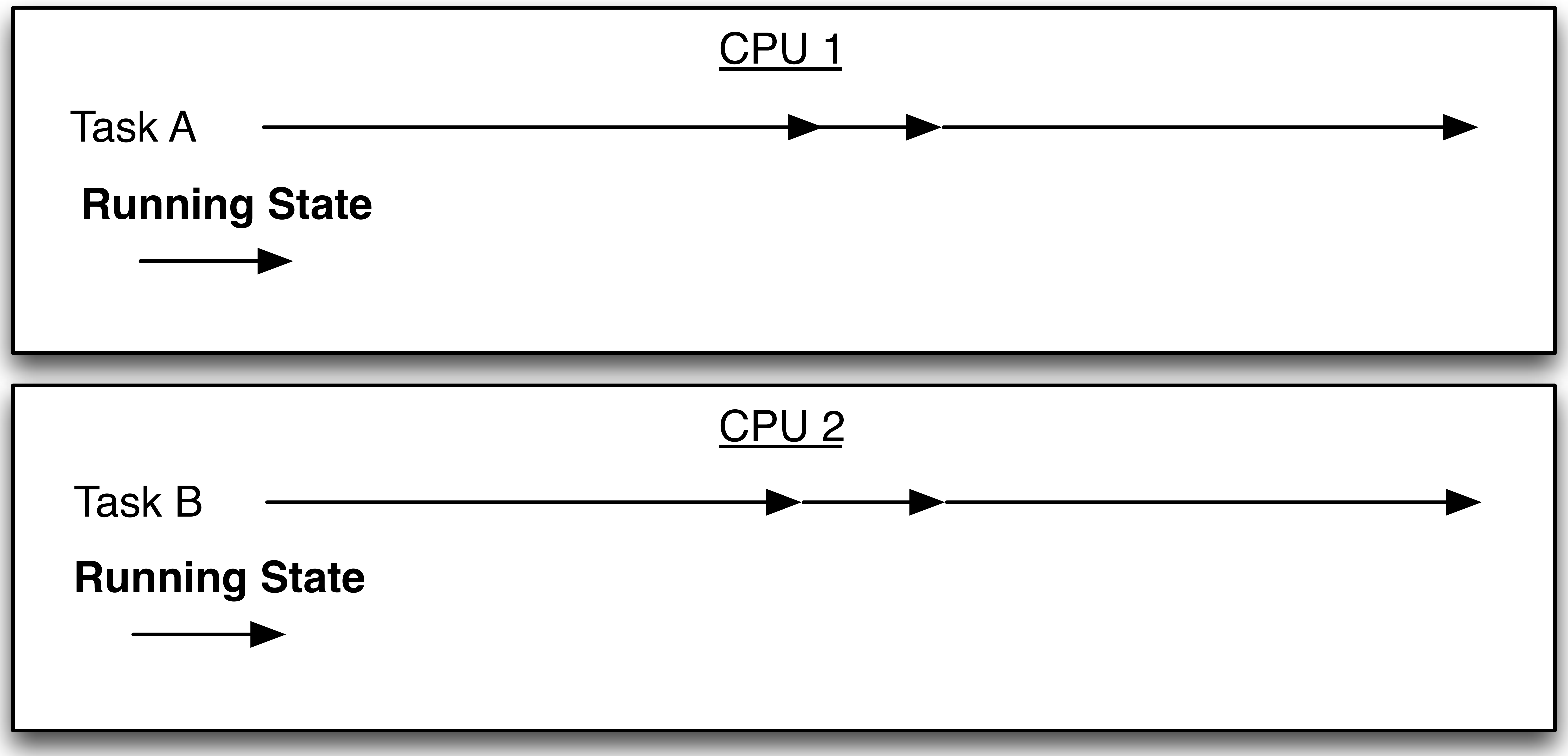
Concurrent Use Cases

- Concurrency: Many units of computation that are fairly independent of each other
  - e.g. A web server handling thousands of connected clients
- Parallelism: Breaking down one large computation into smaller units of computation
  - e.g. Image analysis

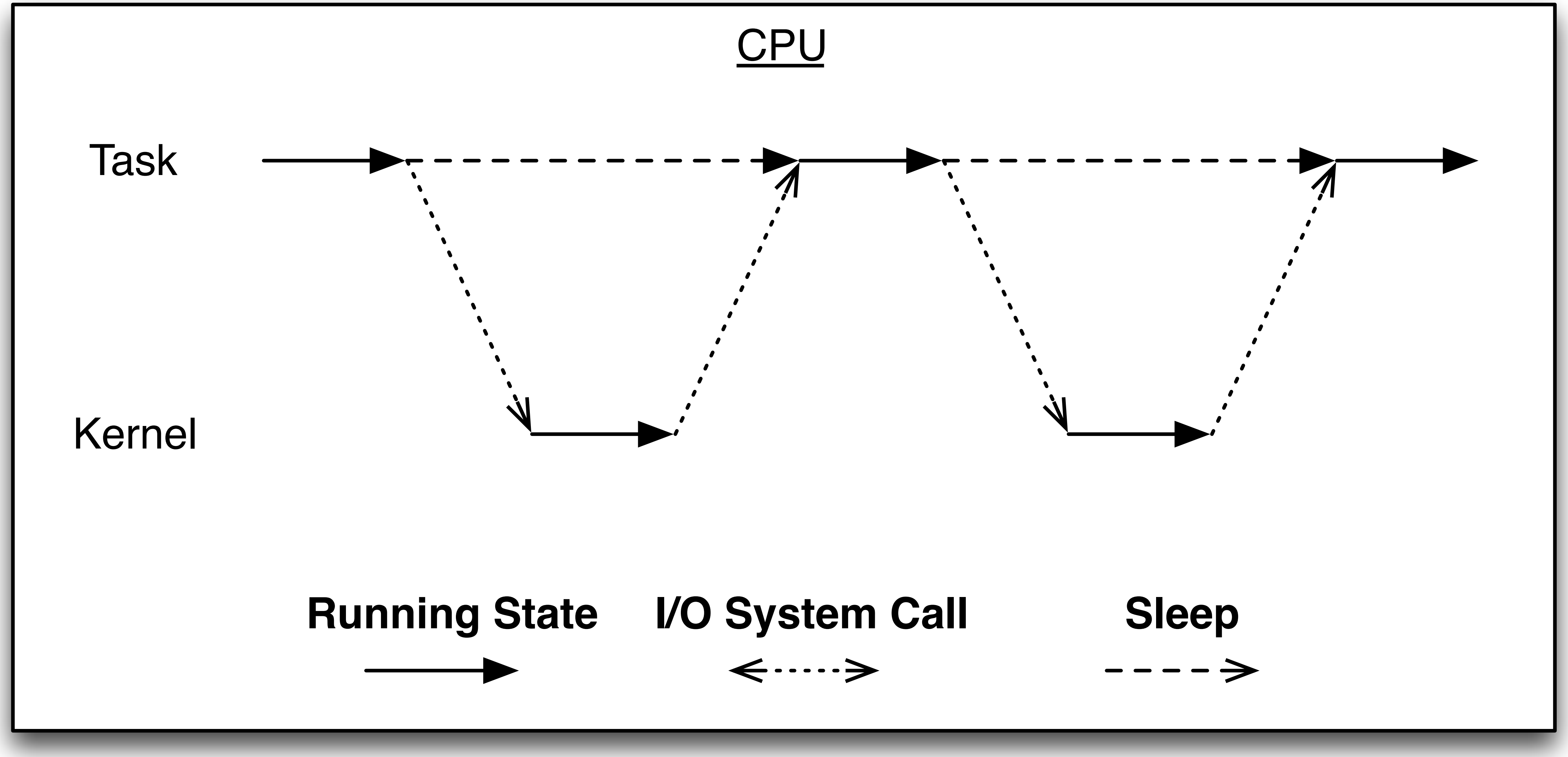
Multitasking



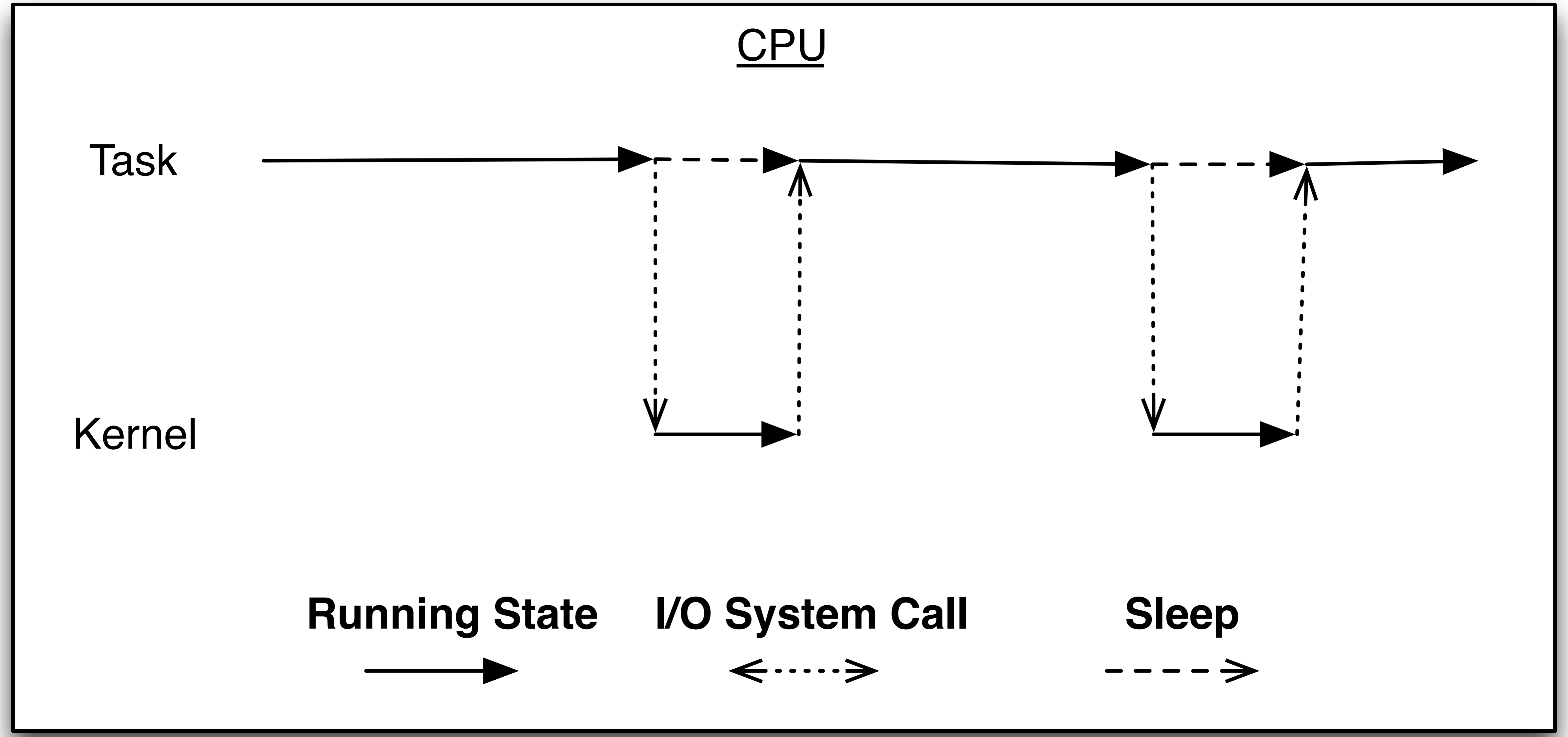
Parallelism



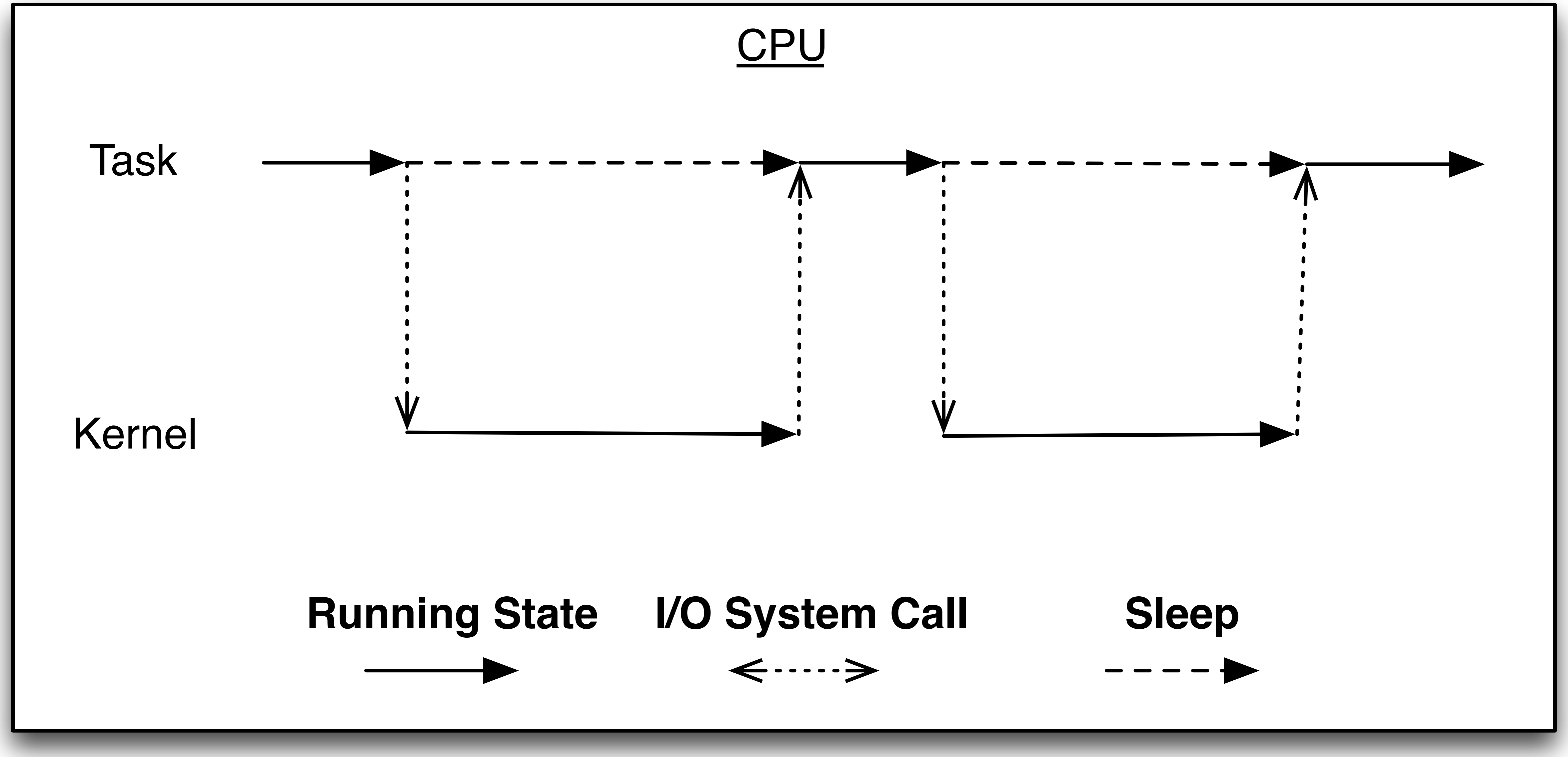
Task Execution



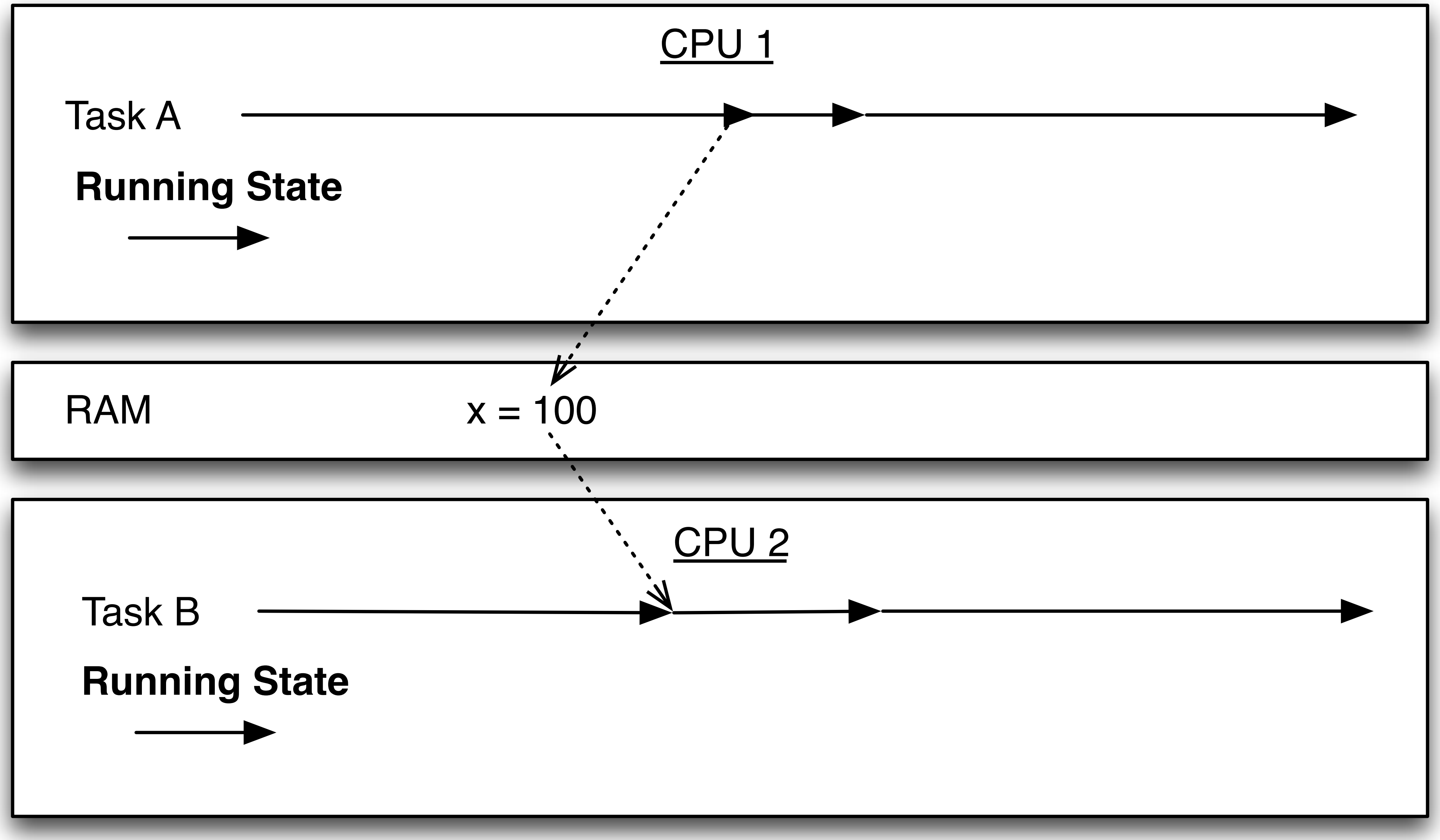
CPU Bound



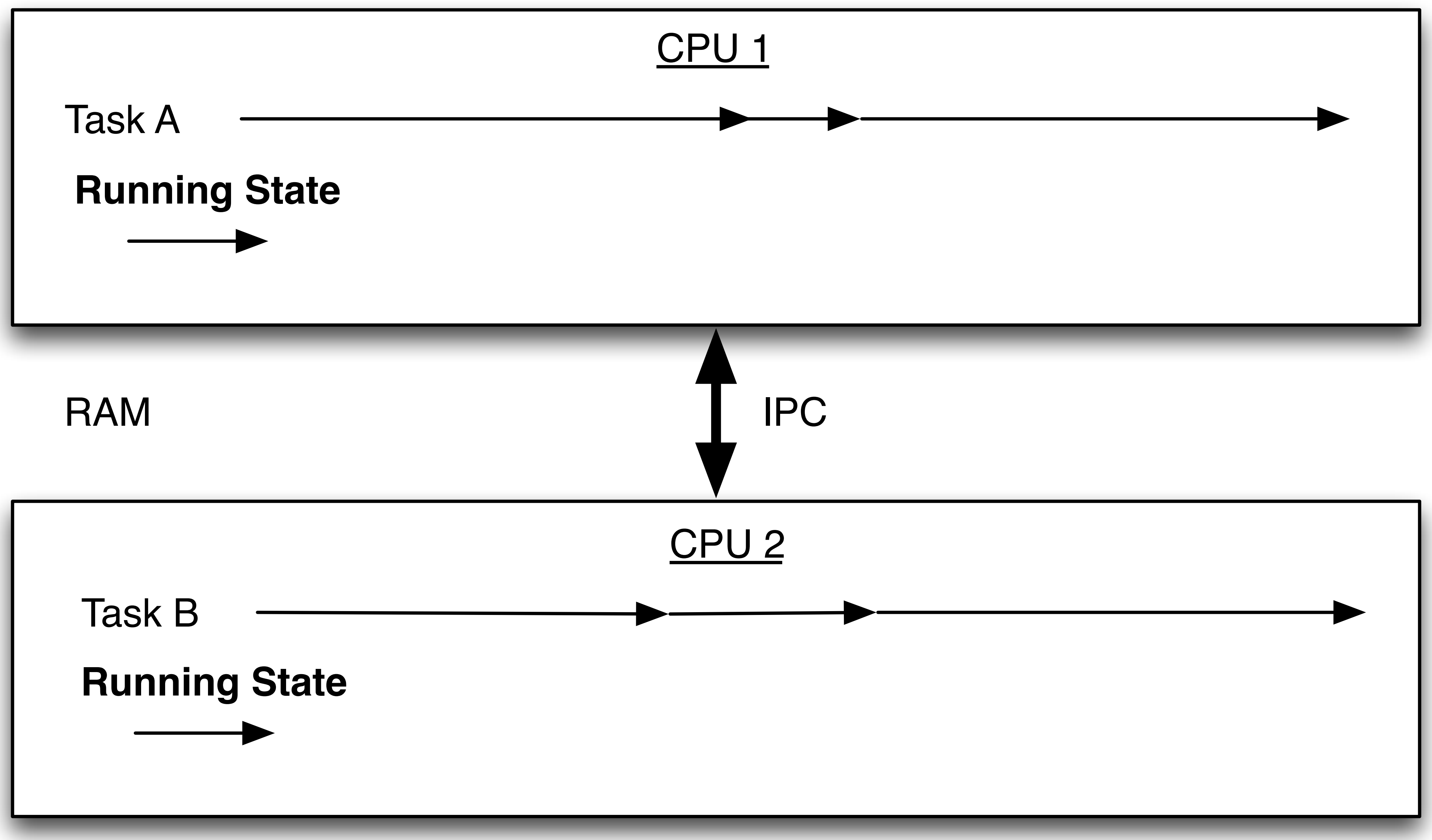
I/O Bound



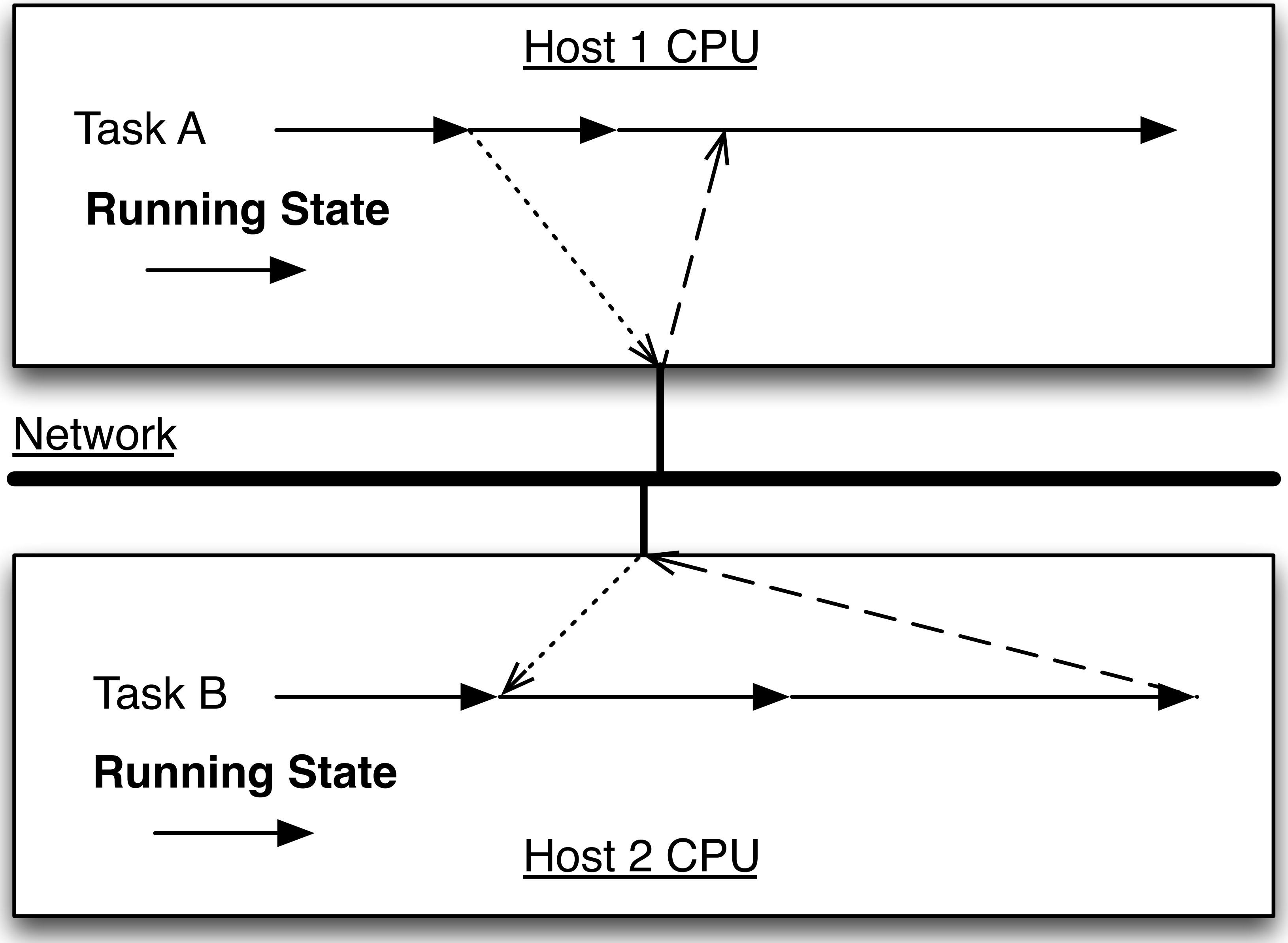
Shared Memory



Processes



Distributed Computing



Part 2: Concurrency

Why Python?

- Sadly, Python and “High Performance” seem orthogonal.
  - Isn’t that what concurrent programming is all about?
- Python is interpreted
  - *Hardware giveth. Software taketh away.*

Why Python?

- High Level
- Large Library
  - “The library makes the language”
- We have our reasons.

As a glue language

- A high-level framework
- A mix of Python, C, C++, Fortran

Programmer Performance

- Programmers revere high-level languages like Python for it’s ability to just work, instead of hacking C code all day

Performance is Misunderstood

- Most programs are I/O bound
  - They’re mainly idle!
- If I/O is the bottleneck, the the overhead of an interpreter is less meaningful.

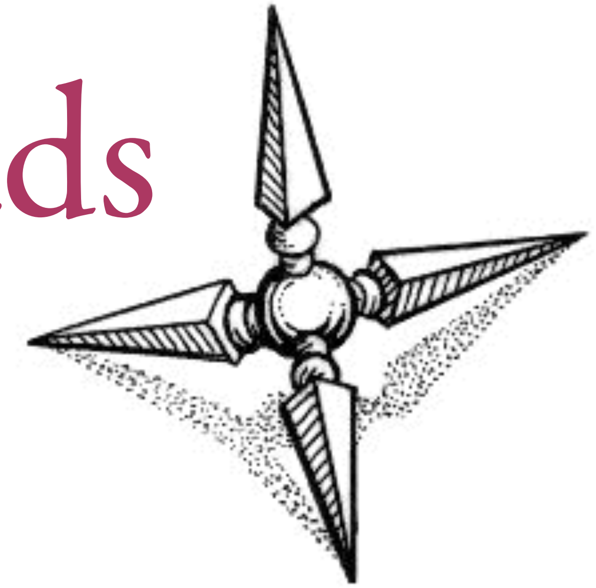
Unless you're CPU bound

- If you need CPU power, then extending with C code can be useful
  - High performance in Python really comes down to using programming in C
- There's no shame in using the right tool for the right job.

No Concurrency

- Concurrency is not a solution around inefficient algorithms
  - Focus on rewriting with a better algorithm, or using a language like C
- A C extension might provide a Python script a 20x improvement in speed vs. a marginal speedup using parallelization

Part 3: Threads



Threads

- Threads are the most common concurrency idiom
  - An independent stream of execution
    - It's own stack, current instruction
  - Inside a parent process
  - Shares all resources with the main and accessory threads
    - memory, files, network connectioncs

Single Thread

- A Python program is started
  - Instructions are executed in a "main thread"

```
$ python program.py
        ↓
   <statement>
   <statement>
        …
        ↓
  <main thread>
```

Multi-threading

- A Python program is started
  - Instructions are executed in a "main thread"
  - A second thread is executed, running in parallel with the main thread.
    - Function `foo()` is executed

```
$ python program.py
          ↓
      statement
      statement
          …
          ↓
 create thread(foo) ⟶  def foo():
                          <statements>
```

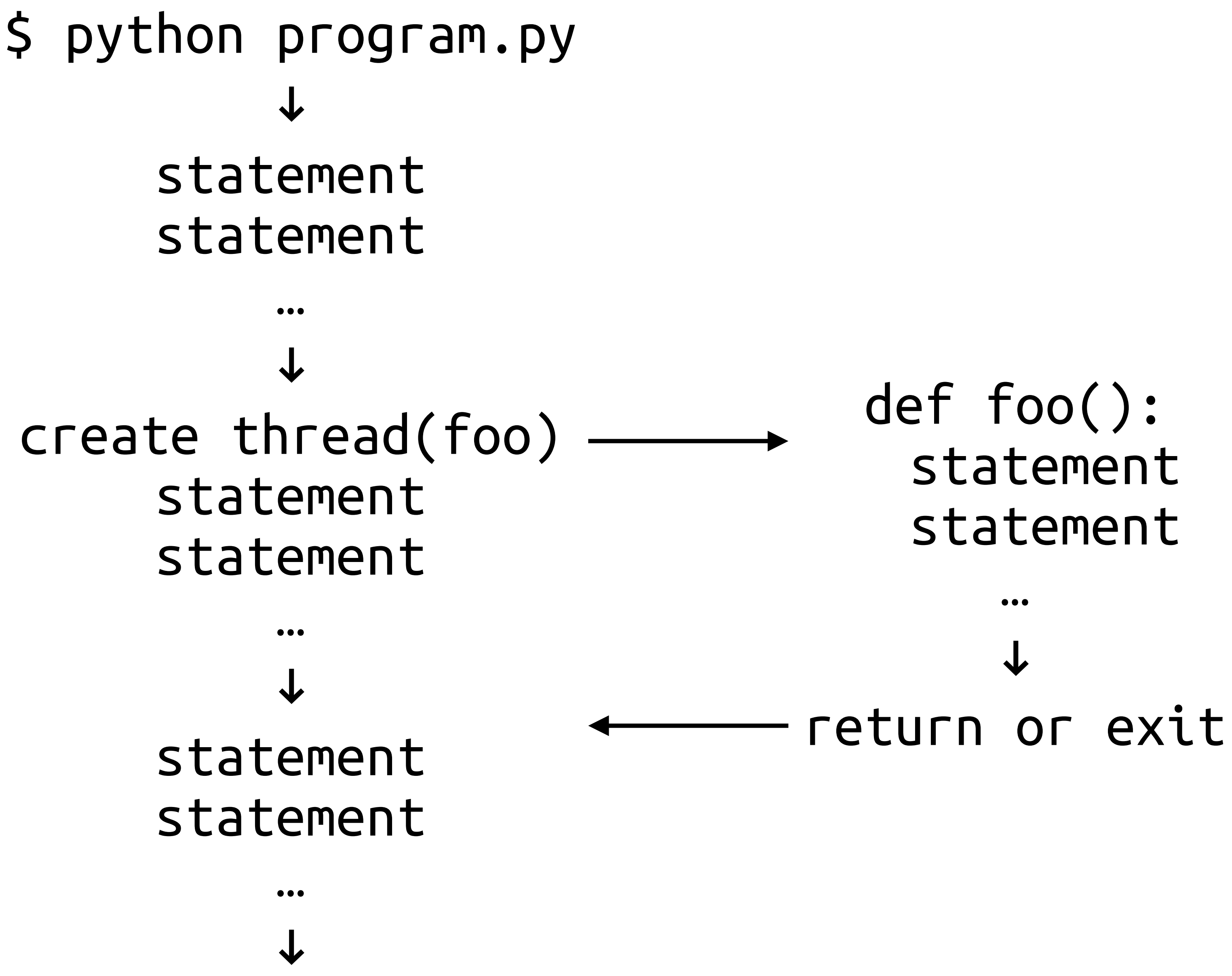
Multi-threading

- A Python program is started
  - Instructions are executed in a "main thread"
  - A second thread is executed, running in parallel with the main thread.
    - Function `foo()` is executed

```
$ python program.py
         ↓
     statement
     statement
         …
         ↓
create thread(foo) ———→ def foo():
     statement            statement
     statement            statement
         …                    …
         ↓                    ↓
```

Multi-threading

- A child thread terminates on `return` or `exit`
  - A thread is a "mini-process", a form of a task that runs independently inside your program



threading module

- The idiomatic method of accessing threads in Python.
- Inherit `threading.Thread` and override `run()`
- The content in `run()` executes in a thread

```
import time
import threading

class CountdownThread(threading.Thread):
    def __init__(self,count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

threading module

```
import time
import threading

class CountdownThread(threading.Thread):
    def __init__(self,count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

```
t1 = CountdownThread(10)  # Create the thread object
t1.start()                # Launch the thread
t2 = CountdownThread(20)  # Create another thread
t2.start()                # Launch
```

threading module

- Functions as threads is an alternative method
- The created Thread object assigns `run` to the given function passed in as the `target` named parameter.

```
import time
import threading

def countdown(count):
    while count > 0:
        print "Counting down", count
        count -= 1
        time.sleep(5)

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

Join The Club

- Threads run independently
- The `join()` method to wait for a thread to exit
- Joining can only happen from outside from outside threads, not the joining thread.

```
t.start() # Launch a thread ...
# Do other work
…

# Wait for thread to finish
t.join() # Waits for thread t to exit
```

What an excellent day for an exorcism.

- Threads run in a **daemon mode** will not prevent your program from hanging on exit
- Good for background utility tasks that require no cleanup — "Set it and forget it!"

```
t.daemon = True
t.setDaemon(True)
```

easy-peasy(-lemon-squeezy)

- Starting threads is easy
- Making many thousands of threads is easy
- The whole idea of threads sounds like a dream!
- But really, it’s a nightmare in disguise…
  - Keeping your program state coherent between many threads — that’s *really* hard

Q: Why did the multithreaded chicken cross the road?
A: to To other side. get the

- Jason Whittington

Shared Data Between Threads

- All threads in a process share access to that process' memory
- Non-deterministic
  - Thread scheduling
  - Access to shared data
- Most operations are non-atomic

Shared Data Conflicts

- Consider the shared memory address in variable **x**
- We have two threads that modify the value at that memory address
- Likely, we've corrupted that value in a non-deterministic way

```
x = 0

Thread-1   Thread-2
--------   --------
 ...        ...
 x = x+1    x = x-2
 ...        ...
```

Shared Data Conflicts

```
Thread-1    Thread-2
--------    --------
 ...         ...
 x = x+1     x = x-2
 ...         ...
```

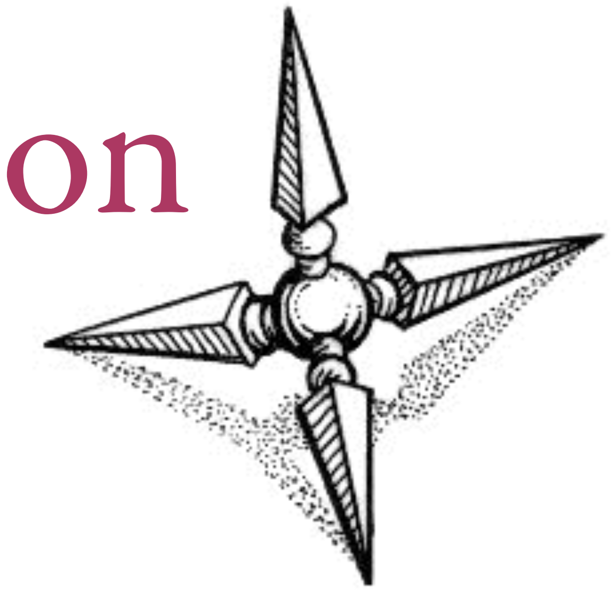
```
Thread-1                          Thread-2
--------                          --------
LOAD_GLOBAL  1 (x)
LOAD_CONST   2 (1)   thread switch
                     ---------->
                                  LOAD_GLOBAL  1 (x)
                                  LOAD_CONST   2 (1)
                                  BINARY_SUB
                     thread switch STORE_GLOBAL 1 (x)
                     <----------
BINARY_ADD
STORE_GLOBAL 1 (x)
```

Example

```
x = 0     # A shared value

COUNT = 10000000
def foo():
    global x
    for i in xrange(COUNT):
        x += 1

def bar():
    global x
    for i in xrange(COUNT):
        x -= 1

t1 = threading.Thread(target=foo)
t2 = threading.Thread(target=bar)
t1.start(); t2.start()
t1.join();  t2.join()
print x     # Expect result = 0
```

Data corruption due to thread scheduling is called a Race Condition

Part 4: Synchronization



Gentleman, synchronize your Swatches

- Parker Lewis

Thread Synchronization

- Avoid race conditions (and losing chunks of your life trying to find them) by using thread synchronization techniques and primitives

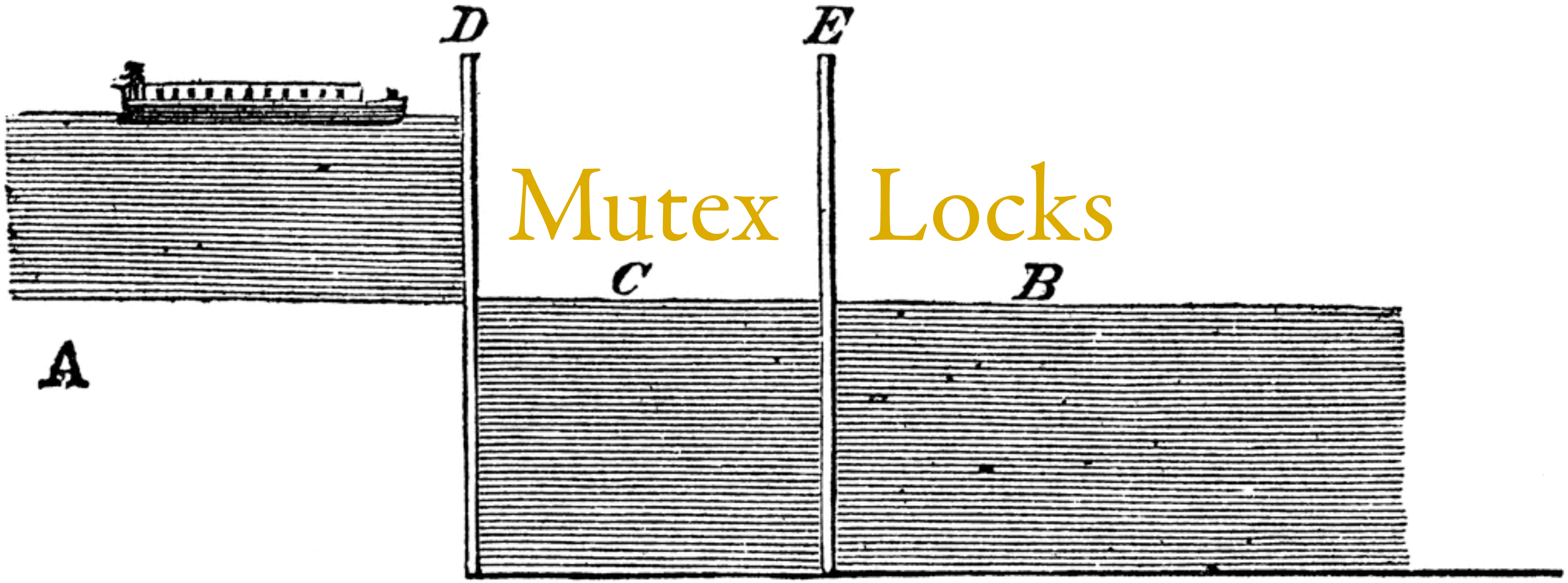
Thread Synchronization

- The threading library has the following options for thread synchronization
  - `threading.Lock()`
  - `threading.RLock()`
  - `threading.Semaphore()`
  - `threading.BoundedSemaphore()`
  - `threading.Event()`
  - `threading.Condition()`

A Tour

- There are many options to choose from with subtleties that may make it difficult to choose the right one for synchronization

Mutex Locks



Mutual Exclusion Locks

- The most commonly used synchronization primitive

```
m = threading.Lock()
```

- Used to synchronize threads as to allow only one thread permission to modify shared data at a given moment

Mutual Exclusion Locks

- Basic Usage

```
m = threading.Lock()
m.acquire()
m.release()
```

- Only one thread can acquire a lock at a time
- Attempts to acquire by a second (or more) threads results in a blocking action until the lock is released

Using Mutex Locks

```
x = 0
x_lock = threading.Lock()

Thread-1                    Thread-2
--------                    --------
 ...                        ...
 x_lock.acquire()           x_lock.acquire()
 x = x+1                    x = x-2
 x_lock.release()           x_lock.release()
 ...                        ...
```

Critical Section

- Used for creating a **critical section** block
- Only one thread can execute in a critical section at a time (i.e. lock gives exclusive access)

Lock Management

- Always release your locks
- Non-linear flow-control can add pain and suffering
- A Pythonic template for a critical section should ~~be used~~

```
x=0
x_lock = threading.Lock()

# Example critical section
x_lock.acquire()
try:
    statements using x
finally:
    x_lock.release()
```

Lock Management

- Python 2.6 and 3.0 improves the semantics for dealing with locks and critical sections
- The lock is acquired automatically, and released when the block exits

```
x=0
x_lock = threading.Lock()

# Critical section
with x_lock:
    <statements using x>
```

Deadlocks

- Using nesting locks is a bad and confusing idea

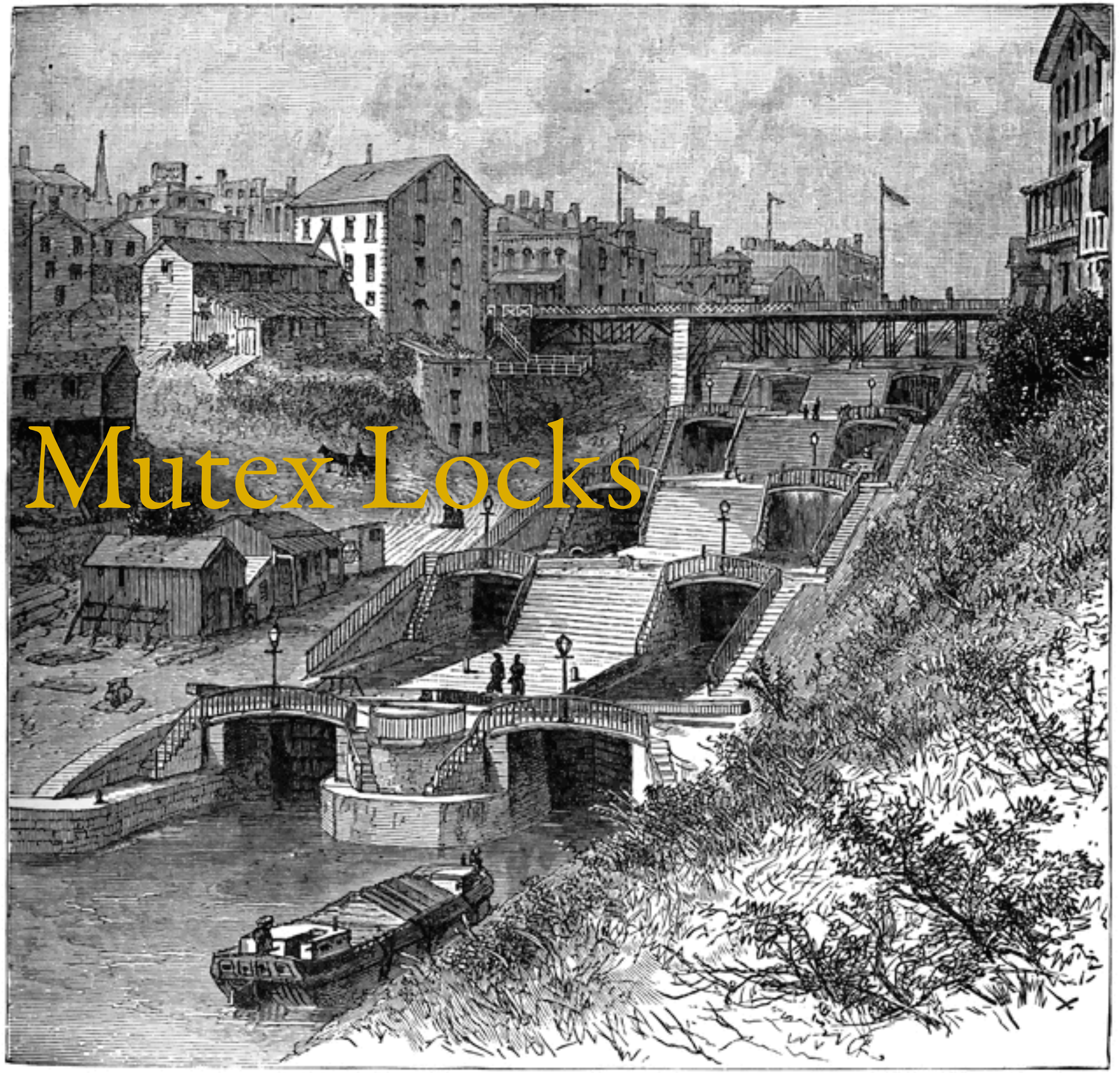
- Expect deadlocks in such situations!

```
x=0
y=0
x_lock = threading.Lock()
y_lock = threading.Lock()

with x_lock:
    <statements using x>
    ...
    with y_lock:
        <statements using x and y>
        ...
```

Mutex Conclusions

- Like threading, locking is easy to do
- That is, until you need to identify and lock all parts of your code that are critical for locking
- It’s another *really* tricky job

Reentrant Mutex Locks



Reentrant Mutex Lock

- RLock

```
m = threading.RLock() # Create a lock
m.acquire()           # Acquire the lock
m.release()           # Release the lock
```

  It extends the normal mutex lock by allowing the lock to be acquired multiple times by the same thread

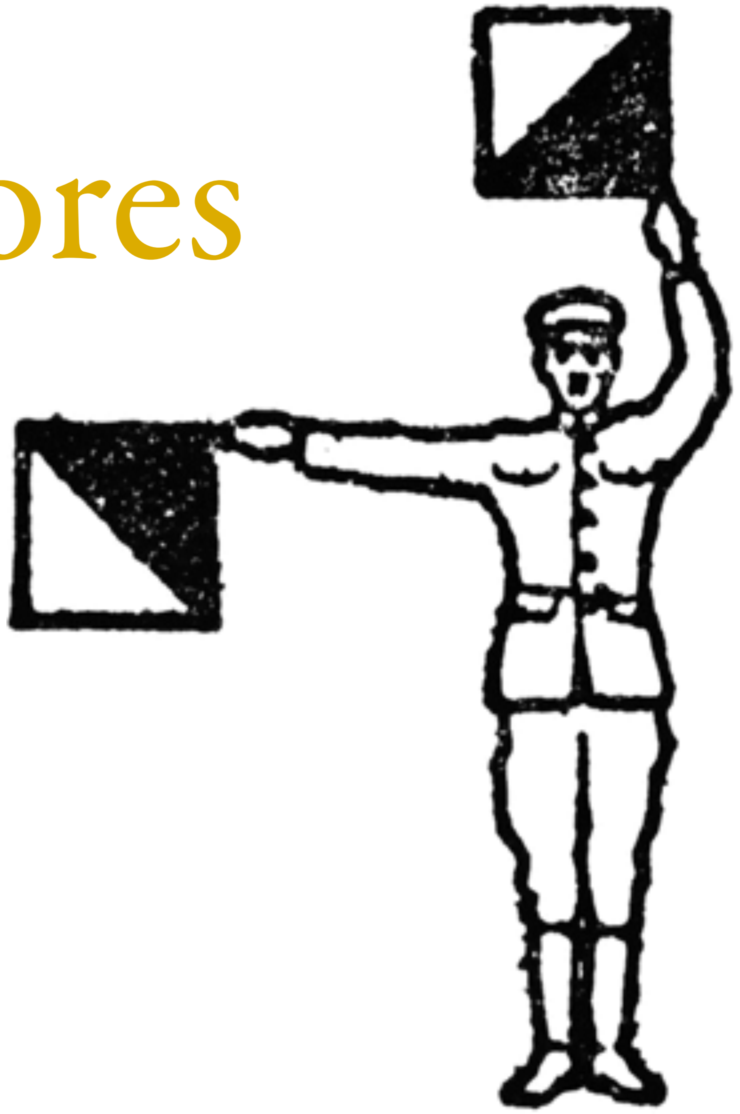
- Each `acquire()` must be balanced by a matching `release()`
- Used commonly for locking code execution, rather than data access

RLock Example

- A monitor object
- Allows only one thread to execute an method in a class at a time
- Methods can call other methods that are holding the lock in the same thread

```
class Foo(object):
    lock = threading.RLock()
    def bar(self):
        with Foo.lock:
            ...
    def spam(self):
        with Foo.lock:
             ...
             self.bar()
             ...
```

Semaphores



Counter-based Synchronization

- Semaphore is one of the oldest synchronization primitives in computer science (Dijkstra)

```
m = threading.Semaphore(n) # Create a semaphore
m.acquire()                # Acquire
m.release()                # Release
```

- `acquire()` – if the counter is > 0, decrement by one and return immediately. If it is == 0, then block and wait until someone calls `release()`
- `release()` – increments the internal counter by one. If the counter is zero when called, wake up a waiting thread as well.

Use cases

- Resource control
  - Setting upper-bound limits for such things as network connections or database accesses
- Signaling
  - Can be used to signal threads into action

Resource Control Example

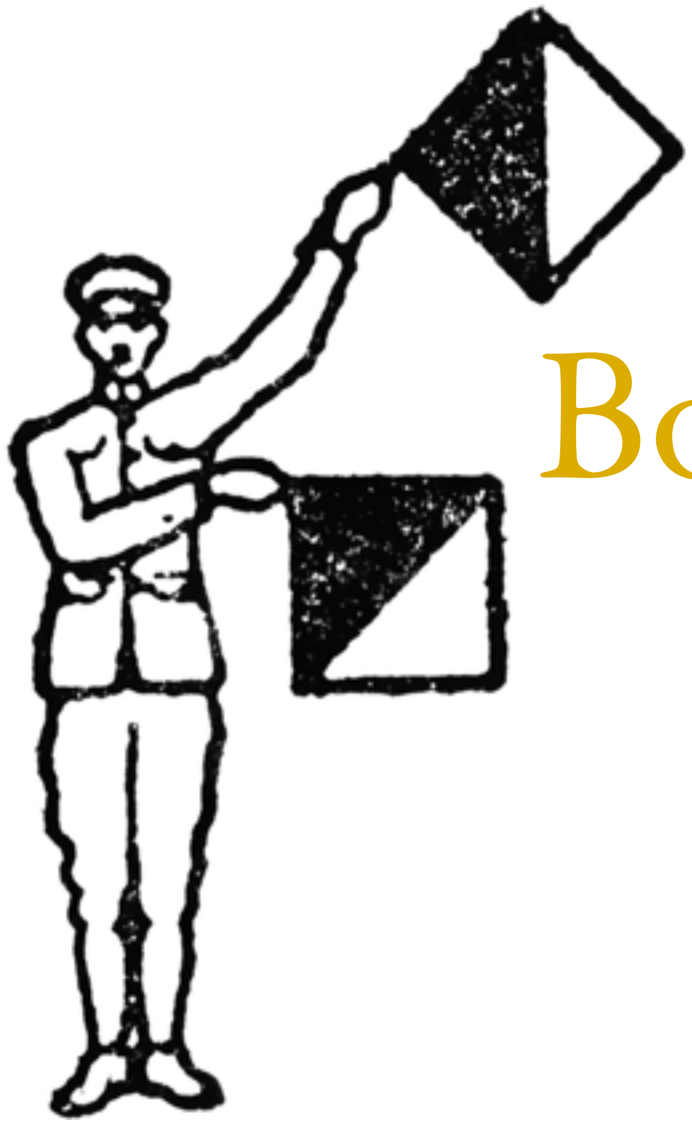
```
sema = threading.Semaphore(5)

def fetch_page(url):
    sema.acquire()
    try:
        u = urllib.urlopen(url)
        return u.read()
    finally:
        sema.release()
```

- Semaphore Resource Control
- Maximum of 5 threads are executing this function at once.
    - Other threads will wait until a semaphore signals a `release()`

Thread Signaling Example

```
done = threading.Semaphore(0)

Thread 1                Thread 2
...                     done.acquire()
statements              statements
statements              statements
statements              statements
done.release()          ...
```

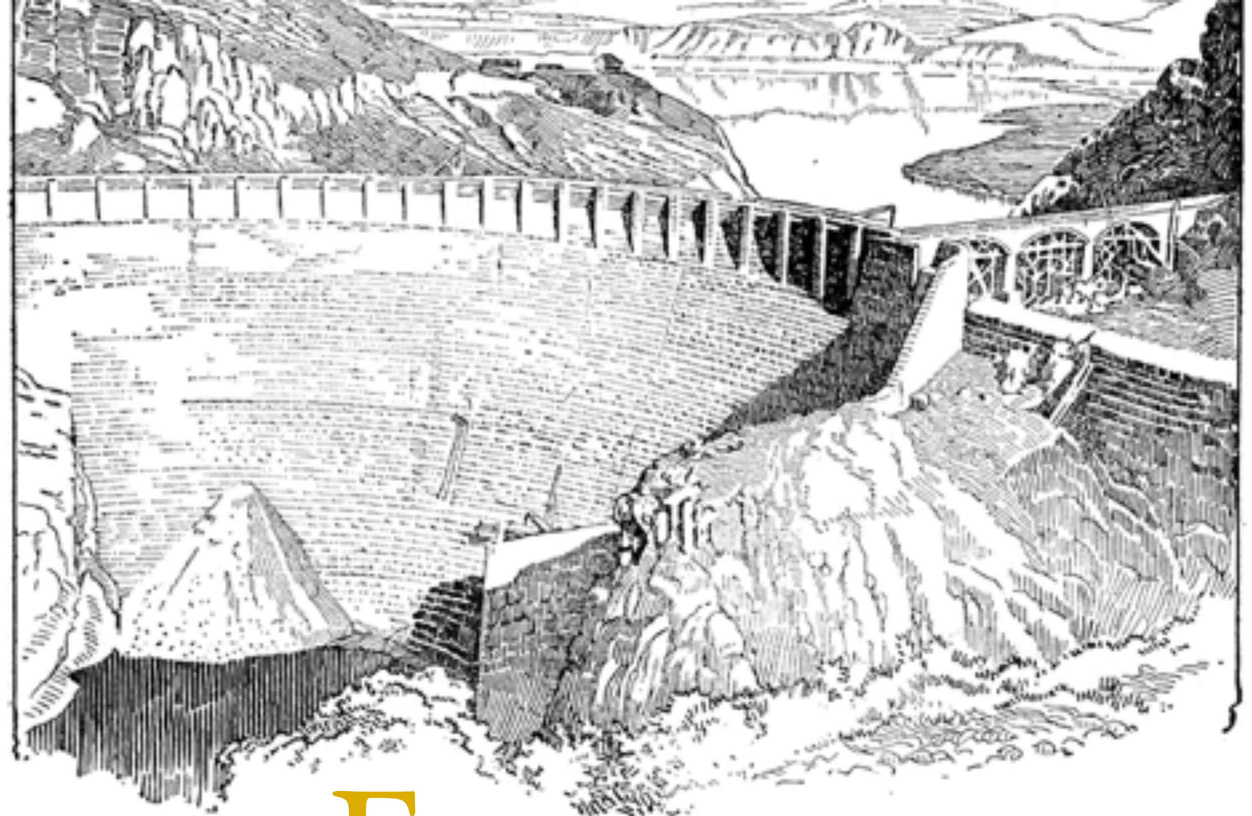
- Semaphore Thread Signaling
- `acquire()` and `release()` are in two different threads and arbitrary order
- Use case: Consumer-Producer problems



Bounded Semaphores

Semaphore Release Checks

- A minor variation of `threading.Semaphore(n)`, `threading.BoundedSemaphore(n)`
- An exception is thrown if too many `release()’s` are called, in which case a `ValueError` exception is called



Events

Events

- Event Objects

```
e = threading.Event()
e.isSet()     # Return True if event set
e.set()       # Set event
e.clear()     # Clear event
e.wait()      # Wait for event
```

- Used if multiple threads are waiting for an event to occur
- A set event will unblock all waiting threads
  - Commonly used for barriers and notifications

Events Example

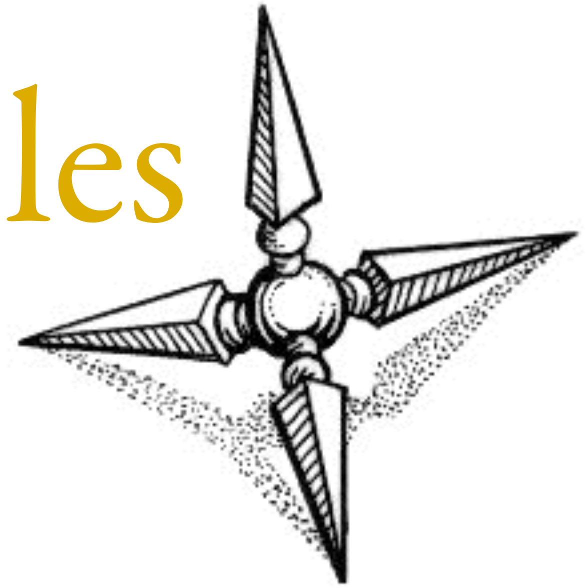
```
init = threading.Event()

def worker():
    init.wait()    # Wait until initialized
    statements
    ...

def initialize():
    statements     # Setting up
    statements
    ...
    init.set()     # Done initializing


Thread(target=worker).start()    # Launch
workers
Thread(target=worker).start()
Thread(target=worker).start()
initialize()                     # Initialize
```

Events Example 2

```
def master():
    ...
    item = create_item()
    evt = Event()
    worker.send((item,evt))
    ...
    # Other processing
    ...
    ...
    ...
    ...
    ...
    # Wait for worker
    evt.wait()
```

```
def worker():
    item, evt = get_work()
    <processing>
    <processing>
    ...
    ...
    # Done
    evt.set()
```

Condition Variables



Conditions

- Condition Objects

```
cv = threading.Condition([lock])
cv.acquire() # Acquire the underlying lock
cv.release() # Release the underlying lock
cv.wait() # Wait for condition
cv.notify() # Signal that a condition holds
cv.notifyAll() # Signal all threads waiting
```

- Lock and Signaling
  - The lock protects critical sections
  - The signal notifies other threads that a state condition has changed

Conditions

```
items = []
items_cv = threading.Condition()
```

Producer Thread

```
item = produce_item()
with items_cv:
    items.append(item)
```

Consumer Thread

```
with items_cv:
    ...
    x = items.pop(0)
# Do something with x
...
```

Conditions

```
items = []
items_cv = threading.Condition()
```

Producer Thread

```
item = produce_item()
with items_cv:
    items.append(item)
    items_cv.notify()
```

Consumer Thread

```
with items_cv:
    while not items:
        items_cv.wait()
    x = items.pop(0)
# Do something with x
...
```

Conditions

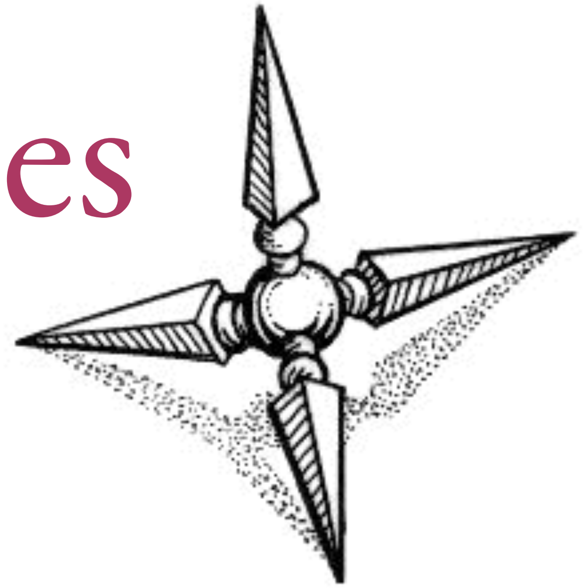
- Before waiting, a lock needs to be acquired
- Conditions are transient, and a verification of the current state is needed, served by the while loop
- `wait()` releases the lock during the wait, and re-locks when woken\

```
Consumer Thread
with items_cv:
    while not items:
        items_cv.wait()
    x = items.pop(0)
# Do something with x
...
```

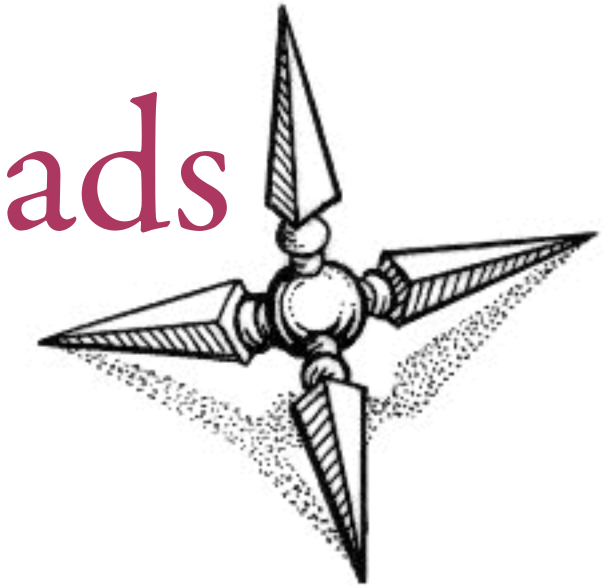
Synchronization Conclusions

- `Lock`, `RLock`, `Condition`, `Semaphore`, and `BoundedSemaphore` objects may be used as `with` statement context managers
- Synchronization primitives are a necessity to make life easy, but once complexity is replaced with another
- Lots of places where things go wrong
  - performance, deadlock, livelock, starvation, scheduling

Part 5: Queues



Part 6: Unraveled Threads



Bad News

- We’ve established threading as a hornets nest of confusion and problems
  - Locks, shared data, queues and synchronization primitives all working together
- On top of that, Python has it’s own platform specific issues, major ones
  - Pathological performance!

Performance Example

- Consider this CPU-bound function

```
def count(n):
   while n > 0:
n -= 1
```

- Sequential Execution

```
count(100000000)
count(100000000)
```

- Threaded Execution

```
t1 = Thread(target=count,args=(100000000,))
t1.start()
t2 = Thread(target=count,args=(100000000,))
t2.start()
```

Unexpected Results

- From David Beazley, http://www.dabeaz.com
- Performance comparison
  - Dual-Core 2Ghz Macbook, OS-X 10.5.6

    Sequential : 24.6s
    Threaded :45.5s (1.8Xslower!)
- With one of the CPU cores disabled:

  Threaded : 38.0s

Part 7: The Inside Story

Nature of Python Threads

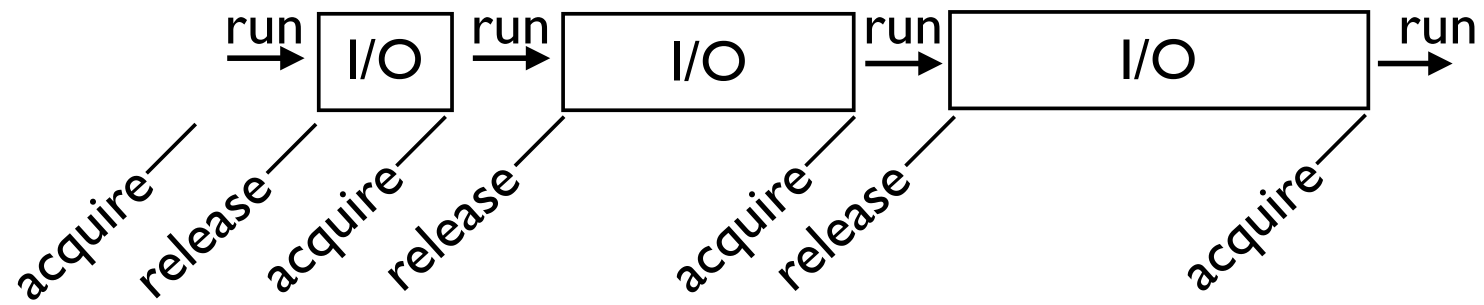
- Python threads are real system threads (POSIX pthreads)
- Scheduled by the host kernel
- Python threads represent the threaded execution of the Python interpreter process which is written in C

The GIL

- Only one Python thread can execute in the interpreter at a time
- The global interpreter lock carefully controls thread execution
- Ensures that each thread gets exclusive access to all interpreter internals when running

I/O Bound GIL Behavior

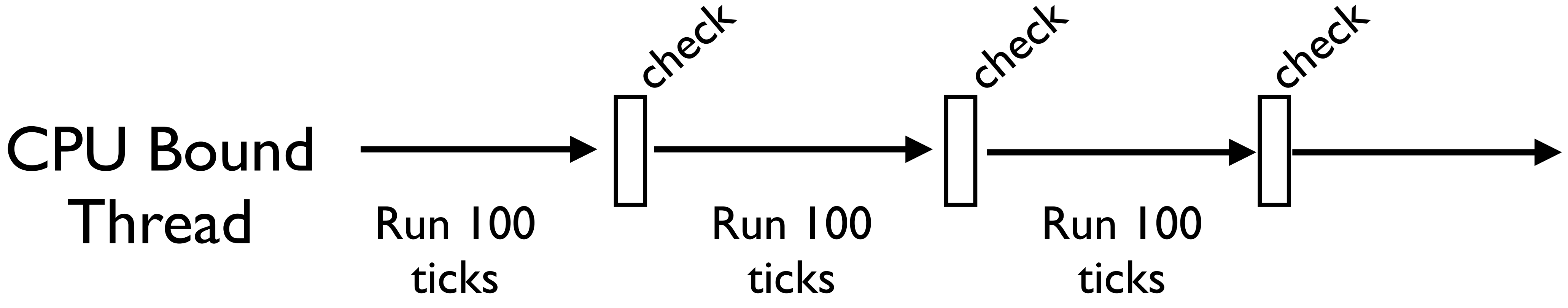
- When a Python-based thread runs, it holds the GIL



- The GIL is released on any block I/O
- When a thread is forced to wait, an idle thread activates
  - Cooperative multitasking

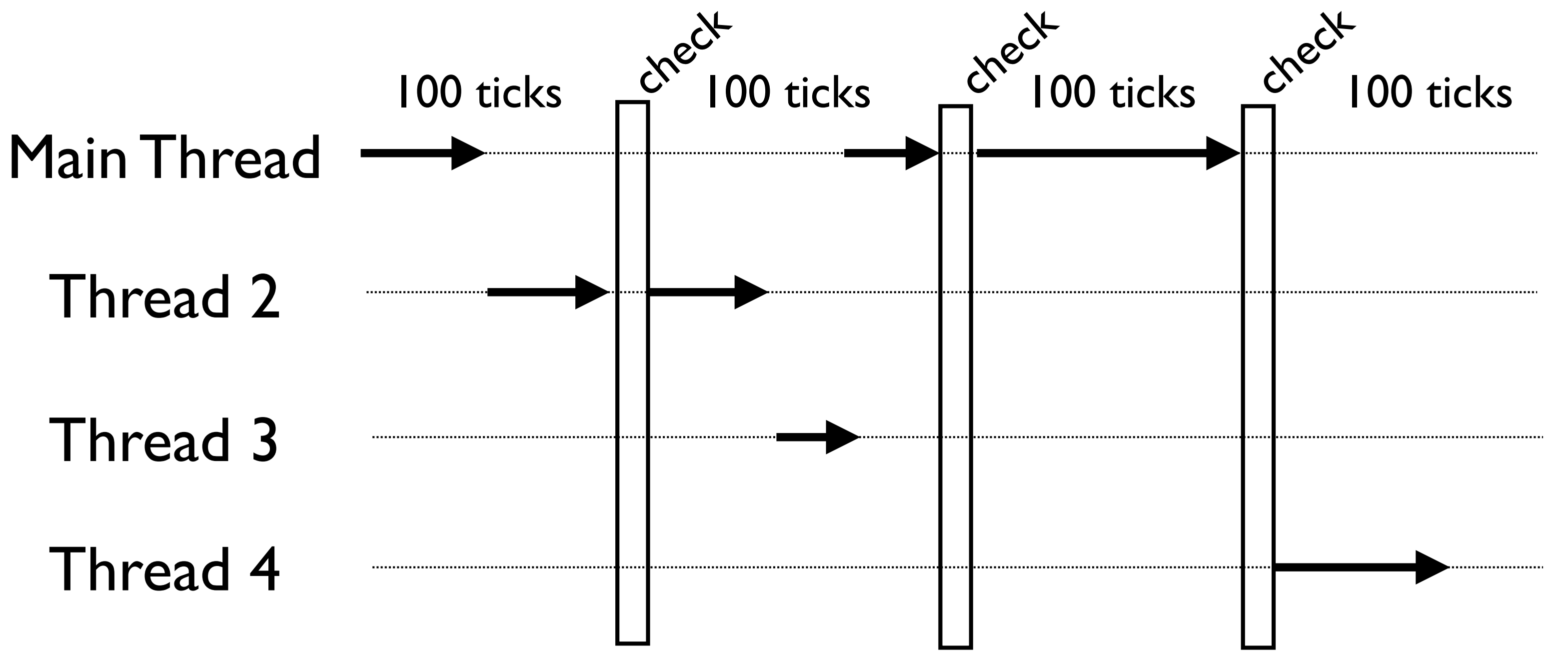
CPU Bound GIL Behavior

- When a thread is CPU-bound, the interpreter periodically checks every 100 interpreter "ticks"



The Check Interval

- The check interval is independent of thread scheduling, where a check is made every 100 ticks



The Check Interval

- During this periodic check
  - Signal handlers in the main thread execute if there are any pending signals
  - Release and reacquisition of the GIL
    - This is how multiple CPU-bound threads get to run, by briefly releasing the GIL, other threads get a chance to run.

Tick

- A Tick has some loose mapping to Python interpreter instructions

```
def countdown(n):
    while n > 0:
        print n
        n -= 1
```

```
         >>> import dis
         >>> dis.dis(countdown)
         0 SETUP_LOOP             33 (to 36)
-------
         3 LOAD_FAST               0 (n)
         6 LOAD_CONST              1 (0)
         9 COMPARE_OP              4 (>)
Tick 1   12 JUMP_IF_FALSE          19 (to 34)
         15 POP_TOP
         16 LOAD_FAST               0 (n)
         19 PRINT_ITEM
-------
Tick 2   20 PRINT_NEWLINE
-------
         21 LOAD_FAST               0 (n)
Tick 3   24 LOAD_CONST              2 (1)
         27 INPLACE_SUBTRACT
-------
         28 STORE_FAST              0 (n)
Tick 4   31 JUMP_ABSOLUTE           3
-------
         ...
```

Tock

- Ticks are not time-based
- Ticks don't have consistent execution times
- Long operations can block all threads, trying hitting Ctrl-C

```
>>> nums = xrange(100000000)
>>> -1 in nums
^C^C^C   (nothing happens, long pause)
...
KeyboardInterrupt
>>>
```

Scheduling Disaster

- Python does not have a thread scheduler
- No notion of thread priorities, preemption, round-robin scheduling, etc.
- All thread scheduling is left to the host OS

GIL Implementation

- The GIL is just a mutex lock
- The Unix implementation is
  - A POSIX unnamed semaphore
  - or a pthreads condition variable
- All interpreter locking is based on signaling
  - To acquire the GIL, check if it is free. If not, sleep and wait for a signal
  - To release the GIL, free it and signal

CPU-bound Threads

- CPU-bound threads have horrible performance
  - Why?

Signaling Overhead

- GIL thread signaling is the source of that
- After every 100 ticks, the interpreter
  - Locks the mutex
  - Signals on a condition variable/semaphore where another thread is always waiting
  - Because of waiting threads, extra pthreads processing and system calls are triggered to deliver

Single-Core Measurements

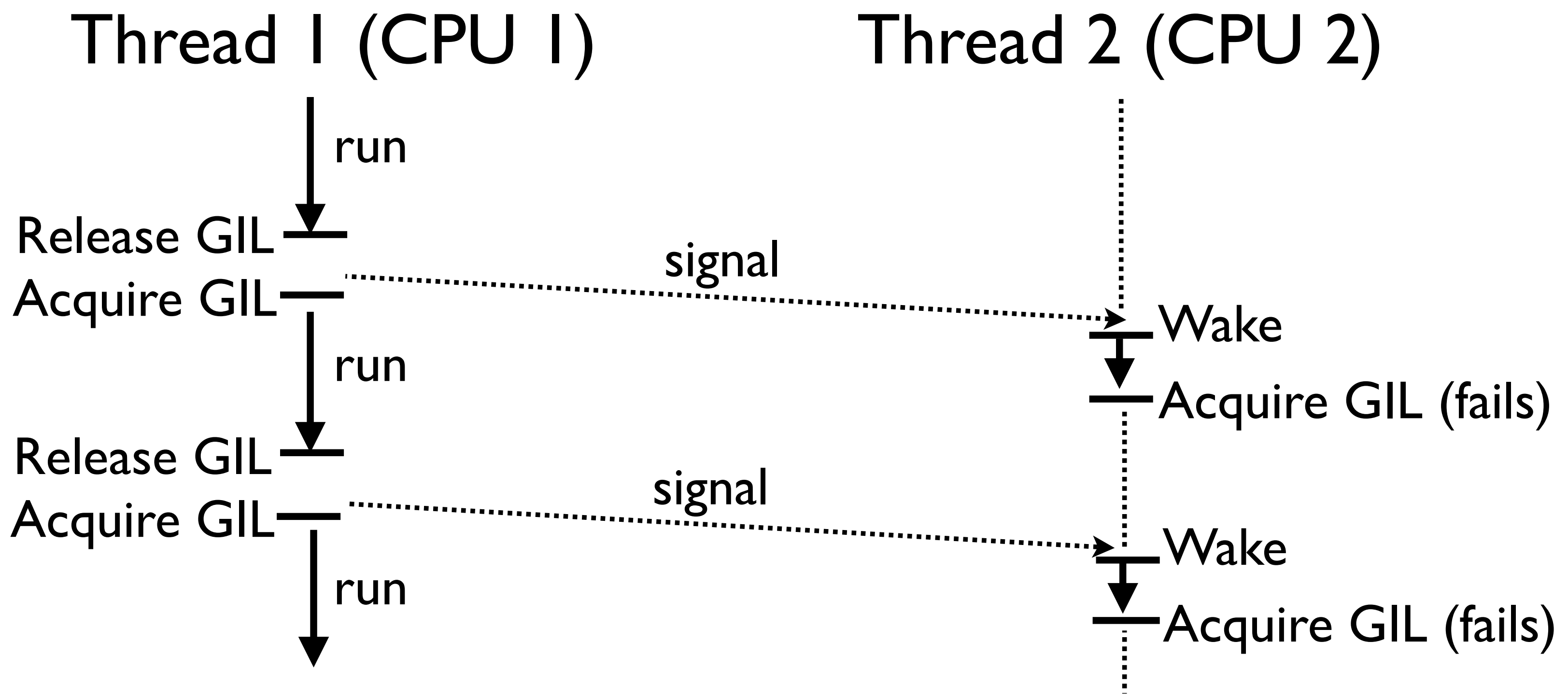
- David Beazley, http://www.dabeaz.com
- Sequential Execution (OS-X, 1 CPU)
  - 736 Unix system calls
  - 117 Mach System Calls
- Two threads (OS-X, 1 CPU)
  - 1149 Unix system calls
  - ~ 3.3 Million Mach System Calls

Multiple-Core Measurements

- David Beazley, http://www.dabeaz.com
- Two threads (OS-X, 1 CPU)
  - 1149 Unix system calls
  - ~ 3.3 Million Mach System Calls
- Two threads (OS-X, 2 CPUs)
  - 1149 Unix system calls
  - ~9.5 Million Mach System calls

Multicore GIL Contention

- CPU-bound threads running on multi-core systems get scheduled simultaneously on different processors, and there is a GIL storm

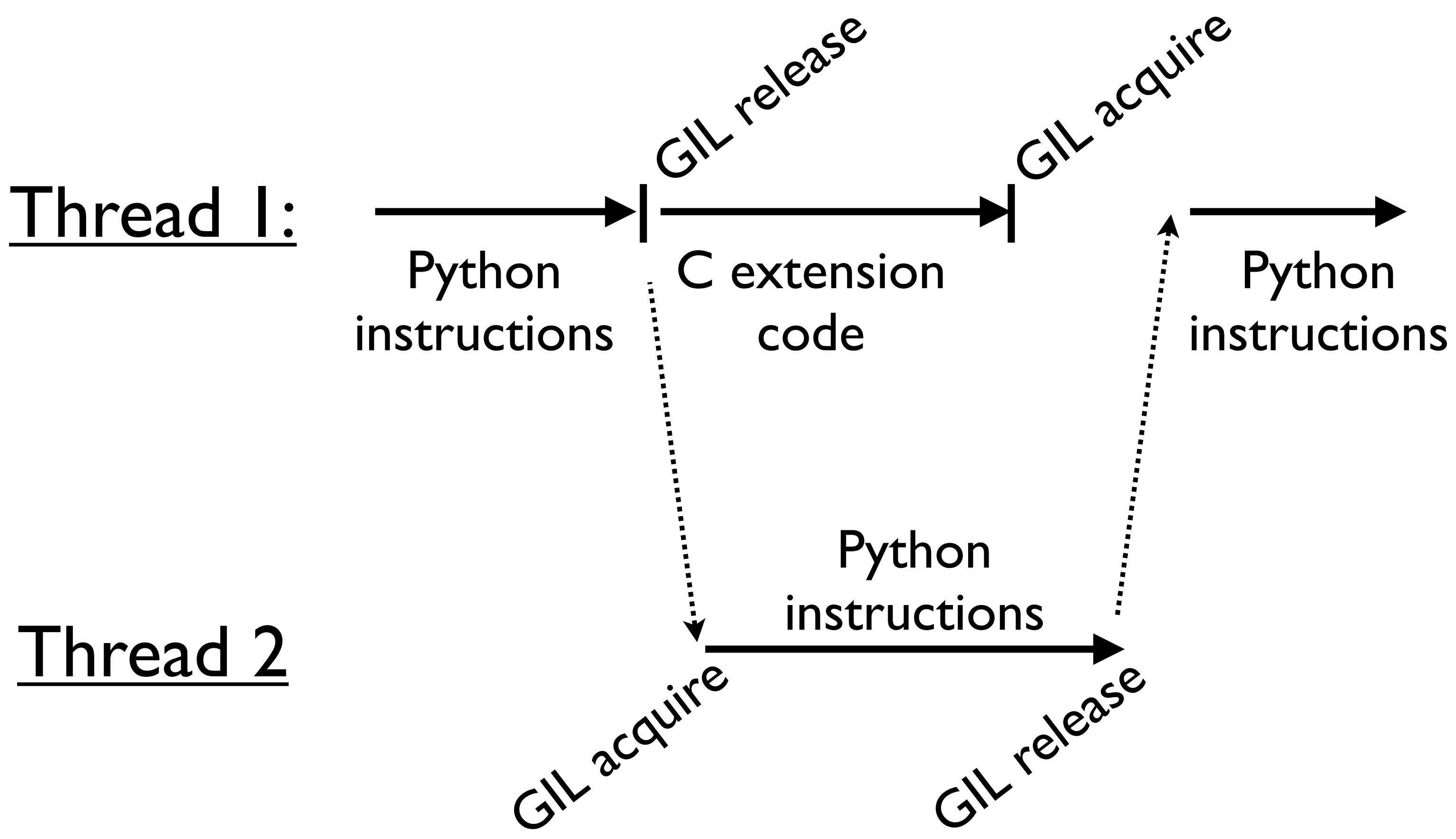


GIL and C

- C/C++ extensions can release the GIL and run independently
- Once released, the C code shouldn't do any state change in the Python interpreter or Python objects
- The C code itself needs to be thread-safe

GIL and C

- It is through C extensions that Python can realize performance parallel computing



Releasing the GIL

- ctypes already releases the GIL when calling C code
- For custom C extensions, you use preprocessor macros

```
PyObject *pyfunc(PyObject *self, PyObject *args) {

    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

Why the GIL

- Simplification of Python Interpreter Implementation
- Better suited for Python's reference counting
- Simplifies use of C/C++ extensions, they don't need to worry about thread synchronization with the interpreter

Part 8: Threading Conclusion

Again, why threads?

- There are areas where threads are useful and perform well

I/O Bound Processing

- Threads are still useful for I/O bound processes
  - e.g. A network server managing thousands of long-lived TCP connections, with low CPU overhead
  - This case is limited by the host OS's ability to provided resources
  - Most systems handle this kind of case just fine

I/O Bound Processing

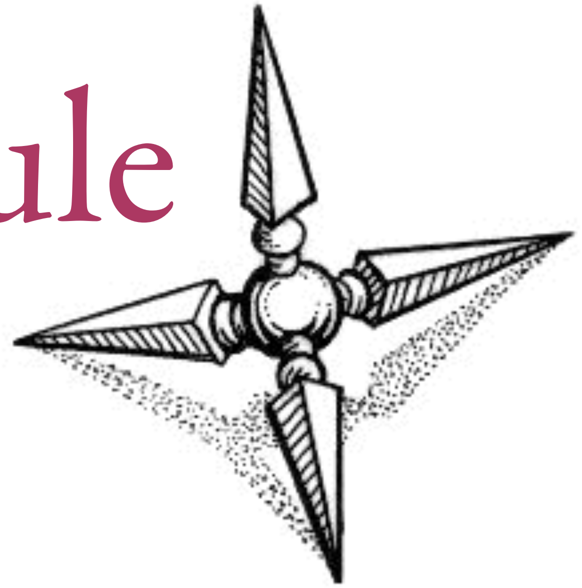
- If everything is I/O bound, there is quick response time to any I/O activity
- Python, as mentioned earlier, does not do the scheduling
  - So, it's behavior will mimic the performance of a C program with a similar I/O boundedness

and Finally…

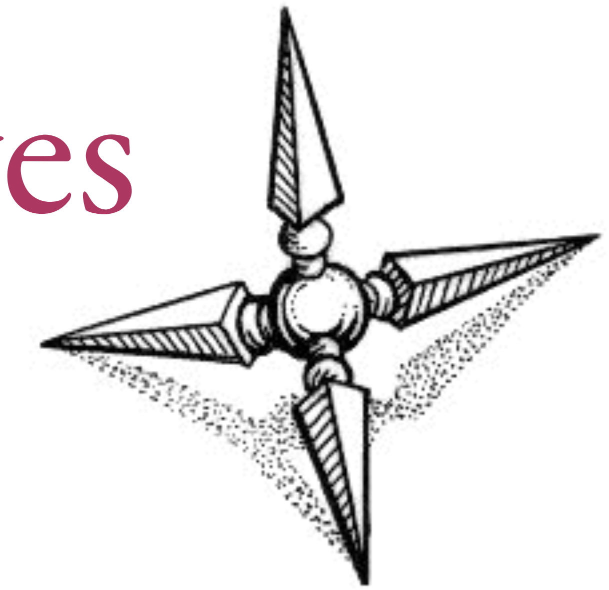
- Python threads are useful:
  - If you use them for I/O bound processing only
  - Limit CPU-bound processing to C extensions that release the GIL
- Threads are only one idiom for parallel processing.
  - A discussion for another time…

Part 9: Processes and Messages

Part 10: Multiprocessing Module



Part 11: Alternatives



Part 12:Closing

References