

Performance

What is performance

- The degree to which the system is able to meet its throughput and/or latency requirements in terms of the number of transactions per second or time taken for a single transaction

Multithreading

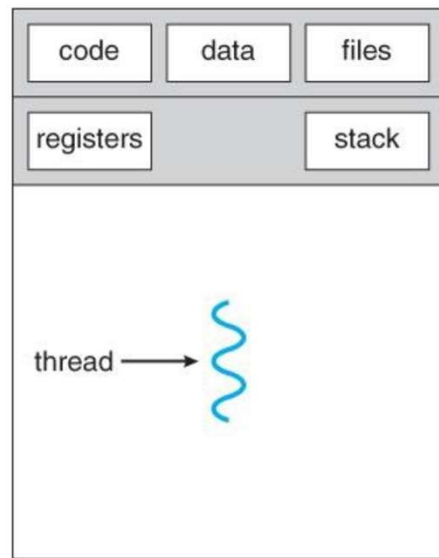
Types of threads

- User-level threads (Python works at this level)
 - Threads we actively create and run for all of our tasks
- Kernel-level threads
 - Very low-level threads acting on behalf of the operating system

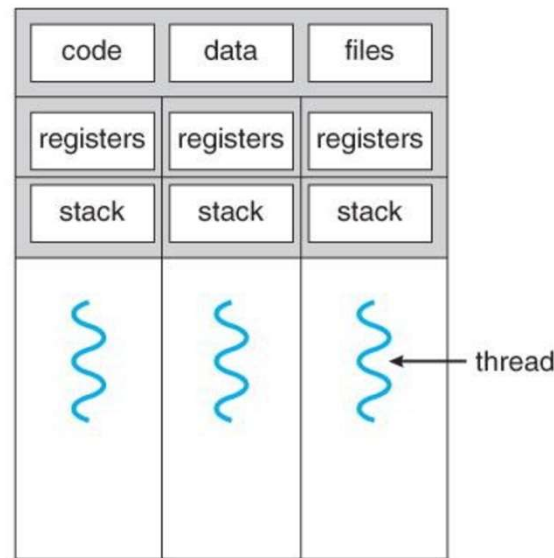
Threads compared with Processes

- Processes are very similar in nature to threads
- Key advantage is that they are not bound to a singular CPU core

Single or multiple threads



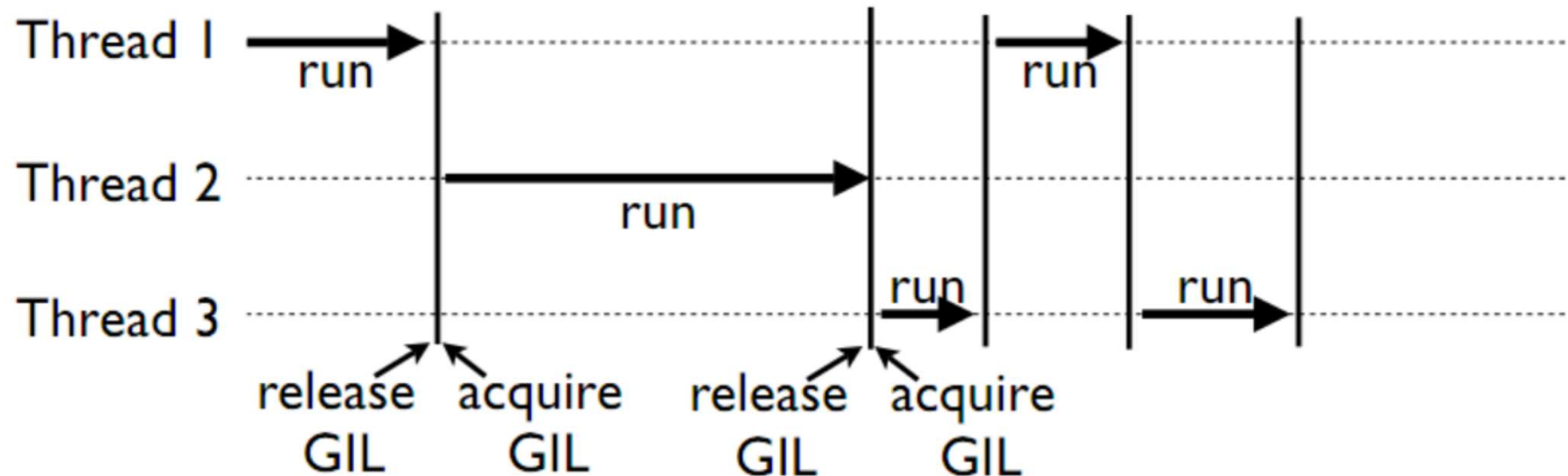
single-threaded process



multithreaded process

The limitations of CPython

- The Global Interpreter Lock (GIL) is a mutual exclusion lock (mutex) which prevents multiple threads from executing CPython code in parallel



Global Interpreter Lock (GIL) in CPython

- The GIL prevents multiple native threads from executing byte codes at same time
- Necessary mainly because the memory management in CPython is not thread-safe
- Other features have grown to depend on the guarantees that the GIL enforces
- GIL is a bottleneck
 - Prevents multithreaded Python programs from taking full advantage of multiprocessor systems
 - Potentially blocking or long-running operations such as I/O, image processing and NumPy operations happen outside the GIL

GIL degrades performance

- System call overhead is significant especially on multicore hardware
 - Two threads calling a function may take twice as much time as a single thread calling the function twice
 - Can cause I/O-bound threads to be scheduled ahead of CPU-bound threads
 - Prevents signals from being delivered
- To minimize the effect of GIL on performance call the interpreter with the -O flag
 - Generates an optimized bytecode with fewer instructions and less context changes

Other Pythons

- Other implementations of Python, such as Jython and IronPython don't feature any form of Global Interpreter Lock
- They can fully exploit multiprocessor systems
- Jython
 - An implementation of Python that works directly with Java
 - CPython's single-core execution typically outperforms Jython and its multicore approach
 - Jython outperforms CPython when working with large datasets
- IronPython
 - Works on top of .NET framework

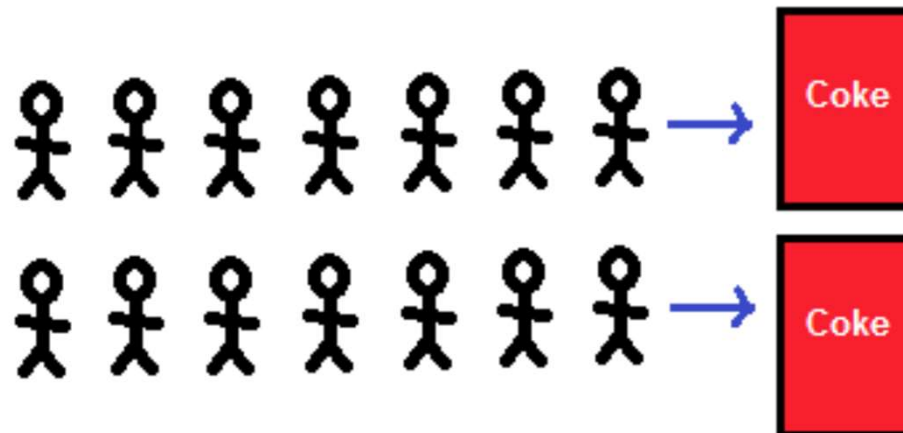
Threads

- Multiple executions in the same process
 - parallelism within a process
- A thread has its own stack
- Each thread shares
 - Heap
 - Static data
 - Process resources (open files etc.)
- Thread scheduling is performed by kernel
 - co-operative
 - pre-emptive

Concurrency and Parallelism



Concurrent: 2 queues, 1 vending machine



Parallel: 2 queues, 2 vending machines

Reasons to use Multi-threading

- Complex applications may need to perform parallel tasks
 - updating graphical information
 - waiting for user input
 - Reading/writing data via sockets
 - performing call-back operations
- Single threaded programs
 - each task runs sequentially
 - tasks cannot run at the same time
 - Can be inefficient
- Multithreaded programs
 - each task can run in parallel
 - harder to program
 - Often more efficient

Thread Objects

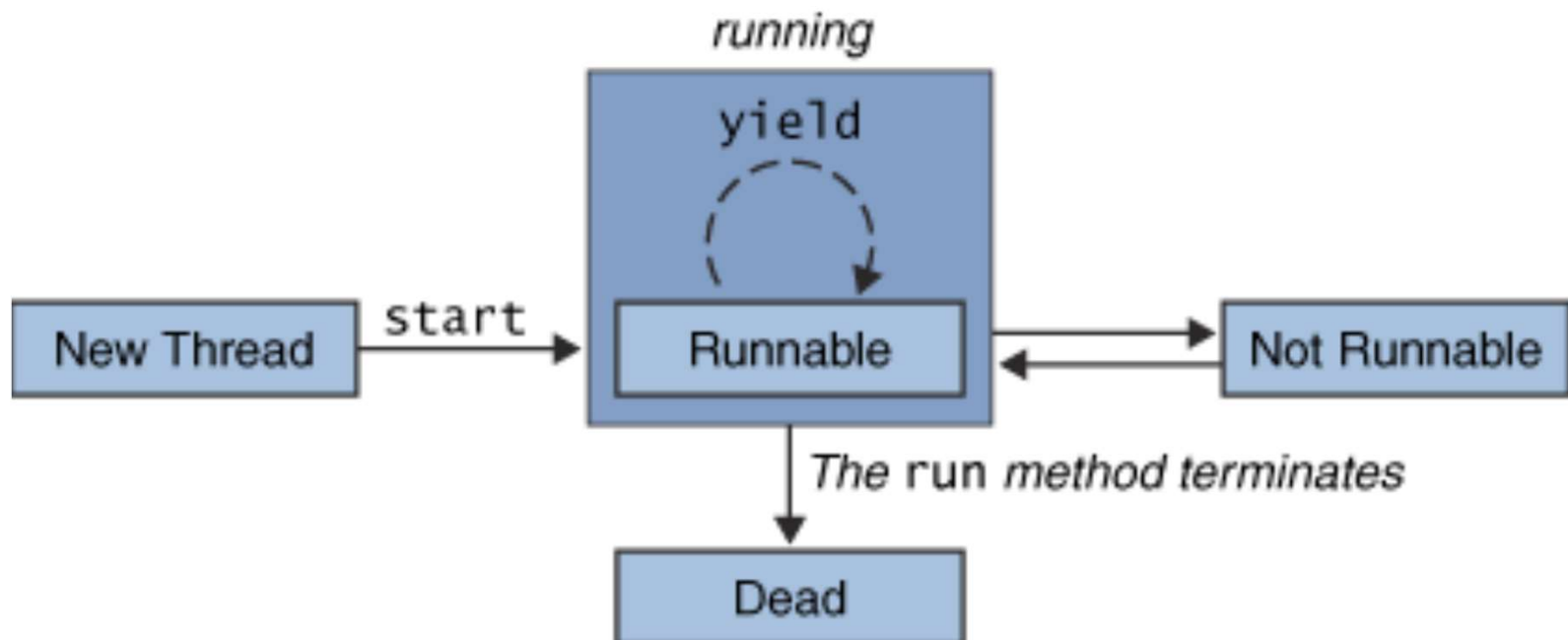
- A thread object represents a thread
 - A thread object is not the same as the thread
 - A thread object is a helper object
- Thread Objects have methods to facilitate thread management
 - Invoke thread object methods by the calling thread
 - Usually only the run() method is executed by the new thread

Thread class definition

```
# Python Thread class Constructor
def __init__(self, group=None, target=None, name=None,
             args=(), kwargs=None, verbose=None):
```

- Constructor takes five real arguments
- group
 - reserved for future extension
- target
 - The callable object to be invoked by the run() method
- name
 - The thread name
- args
 - The argument tuple for target invocation
- kwargs
 - A dictionary of keyword arguments to invoke the base class constructor

Thread Life-Cycle



Invoking Threads

- Use Thread class in the threading module
 - threads have no priorities
 - runs to completion
 - cannot be suspended
- Threading strategies
 - New thread executes a call-back managed by the Thread class
 - The call-back can be a function or a class with `__call__()`
 - Derive from Thread class and define a `run()` method for the thread to execute
- Callable classes are the most flexible
 - you can pass parameters to the new thread
- Call-back function are the simplest

Thread Locking

- Locks are recommended when using threads
 - always release locks in a finally block in case an exception occurs
 - Using the 'with' construct does this automatically
- Shared resources should be locked for increased reliability

Thread Re-entrant-locks (Rlocks)

- Synchronization primitives
 - like standard locks
 - can be acquired by a thread multiple times if that thread already owns it

Thread Events

- Use event objects as signals
 - Thread can wait() on signal
 - Another thread can set() the signal

Thread Semaphores

- Semaphores provide a shared lock
 - define a count
 - Number of threads given simultaneous access to a resource
 - Count is decrement each time semaphore acquired

Immutable Objects

- An object is immutable if its state cannot change after it is constructed
 - A very good strategy for creating reliable multi-threaded code
 - Since they cannot change state they cannot be corrupted
 - No need to use locks

Multiprocessing

Multiprocessing

- There is no reason to have more processes than there are processors on the computer
 - Only `cpu_count()` processes can run simultaneously
 - Each process consumes resources with a full copy of the Python interpreter
 - Communication between processes is expensive
 - Creating processes takes a nonzero amount of time

Choosing an approach

- Always examine the parallel problem to decide which of the many solutions is the best for that problem
- The primary drawback with multiprocessing is that sharing data between processes is costly since it requires pickling the data objects

Futures

- A wrapper around multiprocessing pools and thread pools
- Provides a cleaner API and encourages nicer code
- Function should not access any global variables
- All interaction with the external environment is passed into the function or returned from it
 - Not a technical requirement but is best practice

Process Module

- Similar API to threading module
- Includes synchronization objects
 - Array/RawArray
 - Pipe
 - Pool
 - Manager

Inter-Process Communication (Python 3)

- The subprocess module allows you to create new processes
- Launch OS commands using the run method
 - For in-shell commands use shell=True argument
- Can detect if a command has returned a code other than 0
 - Set the check argument to true so if an error code is returned then a CalledProcessError exception is raised
- Use pipes to control output from one command and input to another
- A series of commands can be piped together

Handling Signals

- Receive signals from other processes