Overcoming The Limitations Of Threads: Takeaways

Syntax

- Turning a Python object into bytecode:

```
def myfunc(alist):
    return len(alist)
dis.dis(myfunc)
```

- Processing objects to represent activity that is run in a separate process:

```
import multiprocessing
def task(email):
    print(email)
    process = multiprocessing.Process(target=task, args=(email,))
    process.start()
    process.join()
```

- Using the Pipe class to pipe data between processes:

```
import multiprocessing
def echo_email(email, conn):
 # Sends the email through the pipe to the parent process.
conn.send(email)
 # Close the connection, since the process will terminate.
conn.close()
 # Creates a parent connection (which we'll use in this thread), and a child connection
(which we'll pass in).
parent_conn, child_conn = multiprocessing.Pipe()
 # Pass the child connection into the child process.
p = multiprocessing.Process(target=echo_email, args=(email, child_conn,))
 # Start the process.
p.start()
 # Block until we get data from the child.
print(parent_conn.recv())
 # Wait for the process to finish.
p.join()
```

- Creating a Pool of processes:

```
from multiprocessing import Pool
 # Create a pool of workers.
p = Pool(5)
```

Concepts

- The GIL (Global Interpreter Lock) in Cpython only allows one thread at a time to execute Python code using a locking mechanism.

- Python enables us to write at a high abstraction layer, which means that code can be extremely terse, but still achieve a lot.
- Threading can speed up I/O bound programs since the GIL only applies to executing Python code.
- The GIL gets released when we do I/O operations, but can also get released in situations where you're calling external libraries that have significant components written in other languages that aren't bounded by the GIL.
- Threads are good for situations where you have long-running I/O bound tasks but they aren't so good where you have CPU-bound tasks or you have tasks that will run very quickly.
- Processes are best when your task is CPU bound or when your task will take long enough.
- Threads run inside processes and each process has its own memory, and all the threads inside share the same memory.
- One thread can be running inside each Python interpreter at a time, so starting multiple processes enables us to avoid the GIL.
- Creating a process is a relatively "heavy" operation, and takes time. Threads, since they're inside processes, are much faster to make.
- Deadlocks happen when two threads or processes both require a lock that the other process has before proceeding.

Resources

- CPython
- Global Interpreter Lock
- Multiprocessing library