lecture09

December 2, 2019

1 MPCS 51042-2 Fall 2019

1.1 Lecture 9: Concurrency

2 Concepts

2.0.1 Concurrent vs. Parallel Tasks

Concurrent tasks run one-at-a-time on a given physical resource (CPU or core) * Tasks are switched when one task is stalled or asleep. * Good for **I/O bound** applications that wait on slow resources * Apps with a lot of I/O, network requests, or user input.

Parallel tasks run simultaneously on different physical resources (CPUs or cores) * Good for **CPU bound** applications that do a lot of small, fast operations. * Cryptography, most physical simulations, machine learning

Tasks are a common term for processes and threads.

2.0.2 Compute, Memory and Network Speeds

This compares the best-case performance two computer systems: * A decent laptop * One node of the Summit supercomputer at Oak Ridge National Laboratory (https://www.top500.org/system/179397)

Component	Laptop	Summit	Difference
CPU:	25 GFLOP/s	1000 GFLOP/s	40x
RAM:	20 GB/s	300 GB/s	15x
Local network:	1 GB/s	25 GB/s	25x
Ethernet:	0.2 GB/s	10 GB/s	50x

2.0.3 Memory and Network Times Relative to CPU

Relative to a DP operation, this is how long it takes to load/store a double from memory or network.

Component	Laptop	Summit
CPU:	1x	1x
RAM:	10x	20x
Local Network:	200x	320x

Component	Laptop	Summit
Ethernet:	1000x	800x

When the CPU has to wait until a value arrives from memory or the network, we say the CPU is **stalled** or **blocked** on resources.

With concurrent programming, the CPU can switch to another task when it is stalled on resources.

2.0.4 What is a process?

A **process** is an instance of an executable program.

Most process activities are controlled by the *operating system*.

Each process has its own: * *User-space memory*: program's variables/objects and executable code * *Kernel data*: information about the runtime state of the process * Virtual memory tables * Table of open file discriptors * Signaling info * Resource limits, priorities

2.0.5 What is a thread?

A **thread** is an execution context of a process. Every process has at least one thread and can create more.

Most thread activities are controlled by *the process that created the thread.*

Within a process, threads share: * Dynamically-allocated ("heap") memory * Kernel data

Each thread has its own: * Local variables * Call stack and program counter

2.0.6 Processes vs. Threads

They share a number of charateristics: * On a single CPU or core, they can be executed concurrently. * On multiple CPUs or cores, they can be executed in parallel.

However, they differ in several important ways: * Threads cannot span multiple network hosts.
* Threads can share memory more quickly and easily than processes.
* Threads can be created and destroyed more quickly than processes.

2.0.7 Threading and the GIL in Python

The CPython interpreter is not fully thread-safe. * An individual object's memory is not protected against race conditions * E.g: two threads simultaneously increment the reference count of the same object

To support multi-threading, the interpreter has a **Global Interpreter Lock** (GIL) that protects **all objects** from race conditions.
* A thread must hold the GIL before it can execute **any** bytecode. * Only **one thread** can hold the GIL at a time.

2.0.8 Working with the GIL

I/O Bound Apps

- All standard library (and many third-party) I/O functions release the GIL when waiting on a result.

- For these apps, **concurrency with multiple threads** performs very well.

CPU Bound Apps

- Without extra C-level code, you can implement **parallelism with processes processes** (each with its own interpreter).
- C-level code can manage the GIL, launch its own OS threads, and use parallelism effectively.

3 Example: Sequential Network I/O

3.0.1 Downloading Images Sequentially

This example (*Fluent Python*, Ch 17) downloads images of country flags over HTTP. Consists of a few functions:

`get_flag` takes a country code ('BR', 'FR', etc), downloads the image, and returns the binary contents of the response.

```
BASE_URL = 'http://flupy.org/data/flags'

def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = requests.get(url)
    return resp.content
```

3.0.2 Downloading Images Sequentially, cont.

`save_flag` takes a (binary) string and writes it to file.

```
DEST_DIR = 'downloads/'

def save_flag(img, filename):
    path = os.path.join(DEST_DIR, filename)
    with open(path, 'wb') as fp:
        fp.write(img)
```

3.0.3 Downloading Images Sequentially, cont.

`download_many` is the main loop that gets and saves flags from a list of country codes.

Now, it is done sequentially. In later examples, this loop will be done concurrently.

```
def download_many(cc_list):
    for cc in sorted(cc_list):
        image = get_flag(cc)
        save_flag(image, cc.lower() + '.gif')
    return len(cc_list)
```

3.0.4 Downloading Images Sequentially, cont.

In `flags.py`, the sequential implementation (from above) is run for 20 countries

```
In [1]: %run ./flags.py
```

```
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 4.00s
```

We will next implement a few concurrent versions with `ThreadPoolExecutor`:

```
In [2]: %run ./flags_threadpool.py
```

```
CNBD  CDBR  EGID  MX IRDE  IN JP FR VNNG ET  RU PK USTR  PH
20 flags downloaded in 0.12s
```

4 Concurrency with Futures

4.0.1 What are Futures?

A future is a task that will eventually be scheduled for execution.

A `Future` object (from `concurrent.futures`) encapsulates future tasks * The states of pending, current, and completed tasks * The results of the tasks

A `Future` should be managed by a higher-level framework, not the client code. The framework handles the actual scheduling and execution of the tasks.

4.0.2 What are Thread- and ProcessPools?

`concurrent.futures` has interfaces for easily handling processes or threads: * `ThreadPoolExecutor` * `ProcessPoolExecutor`

They let you execute callables on different threads or proceses.

Internally, each manages: * An internal pool of worker threads (or processes) * A queue of tasks to be executed

4.0.3 Using `ThreadPoolExecutor` and `ProcessPoolExecutor`

Both implement the generic `Executor` interface and can be used polymorphically.

Simplest usage is `map`-like. This returns an iterable of results.

```
with futures.ThreadPoolExecutor() as executor:
    results = executor.map(function, inputs)
```

We'll look at ways to query and get results, too.

5 Example: Concurrent I/O with Threadpool and Map

The stdlib `concurrent.futures` module provides

5.0.1 Downloading One Flag

This downloads and saves the flag for one country code.
* It uses the same `get_flag()` and `save_flag()` functions as before.
* In our sequential implementation, it was the body of the `for`-loop * In our concurrent implementation, it will be executed many times by multiple threads.

```
def download_one(cc):
    image = get_flag(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc
```

5.0.2 Downloading Many Flags Concurrently

This runs `download_one` for each country code in `cc_list`.
1. The number of worker threads to create.
2. Instantiate the `ThreadPoolExecuter`. The context manager will block until all workers are done (by calling `executor.shutdown(wait=True)` during `__exit__`).
3. Call `download_one` concurrently on multiple threads. Returns a generator of ordered results.

```
def download_many(cc_list):
    workers = min(MAX_WORKERS, len(cc_list))              # 1
    with futures.ThreadPoolExecutor(workers) as executor: # 2
        res = executor.map(download_one, sorted(cc_list)) # 3
    return len(list(res))
```

5.0.3 How many threads should you create?

This example creates only $\leq$ 20 worker threads.

Most systems can easily handle thousands of worker threads. * Threads go to sleep when they're stalled.
* Sleeping threads have little (if any) impact on other programs running on the same system.
* However, if many threads are awake at the same time, performance can take a big hit.

5.0.4 How many threads should you create?, cont.

Rules of thumb: * Use large worker counts only on I/O-bound tasks.
* Always create a hard limit (even a large one) on `max_workers` to prevent misuse.

5.0.5 What does the context manager do?

```
with futures.ThreadPoolExecutor(workers) as executor:
    res = executor.map(download_one, sorted(cc_list))
```

- Instantiates the `ThreadPoolExecuter` and waits until all workers are finished.
  - Inside the with-block, `executor.map()` can execute threads asynchronously.
  - At the end of the with-block, `executor.__exit()` is called.
  - `executor.__exit()` will call `executor.shutdown(wait=True)`

5.0.6 What is the return value?

```
res = executor.map(download_one, sorted(cc_list))
```

The return value (`res`) is a generator that returns ordered results from each function call.

If any call failed, the exception will be raised when getting the corresponding return value * i.e., exception is raised when `__next__()` is called on `res` * This prevents the entire threadpool from failing when only one call has failed

6 Example: Using `Executor.submit` and `futures.as_completed`

6.0.1 Submitting tasks one at a time

We can schedule callables one-at-a-time on an existing `Executor` by using `Executor.submit()`

We can retrieve results as they are completed using `futures.as_completed()`

Contrast this scheme with `Executor.map()` where: * Tasks are submitted all at once * Results can only be retrieved when all tasks are done

6.0.2 Step 1: Instantiating the ThreadPool

Submitting tasks and getting results both take place inside a single context manager:

```
with futures.ThreadPoolExecutor(max_workers=3) as executor:
    # ... schedule tasks and get results
```

For this demo, we only use `max_workers=3` so we can monitor results. This scheme can handle large thread counts, just like `map`.

6.0.3 Step 2: Scheudling Tasks

1. Schedules a call to `download_one` in the threadpool `executor`. This returns an instance of `Future` for this single task.

2. Store the `Future` in a simple `list` so we can retrieve it later.

```
to_do = []
for cc in sorted(cc_list):
    future = executor.submit(download_one, cc)   # 1
    to_do.append(future)                         # 2
    print('Scheduled for {}: {}'.format(cc, future))
```

6.0.4 Step 3.1: Retrieving Results

`as_completed` yields each `Future` as it's completed.
* `as_completed` takes an iterable of `Futures` and an optional timeout * Here, every `Future` was created by the same `Executioner`, but `as_completed()` can take arbitrary `Futures` from any `Executioners`. * With timeout, a `TimeoutError` will be raised when `__next__` is called and a result isn't returned within the timeout.

```
results = []
for future in futures.as_completed(to_do):     # 3.1
    res = future.result()
    print(msg.format('{} result: {!r}', res))
    results.append(res)
```

6.0.5 Step 3.2: Retrieving Results

`future.result()` returns the result of the callable associated with the `Future` * `result` takes an optional timeout * In general, `result` stalls until result is ready (or timeout if provided) * In this case, we've ensured that `result()` will not stall. Any stalls will hit `as_completed()` first.

```
results = []
for future in futures.as_completed(to_do):     # 3.1
    res = future.result()                      # 3.2
    print(msg.format('{} result: {!r}', res))
    results.append(res)
```

6.0.6 Running Implementation with `submit` and `as_completed`

When we run this version, the results (`<Future at ..>`) are not returned in the same order as they are scheduled. This is expected.

```
In [3]: %run flags_threadpool_ac.py
```

```
Scheduled for BR: <Future at 0x1060f4828 state=running>
Scheduled for CN: <Future at 0x106048cc0 state=running>
Scheduled for ID: <Future at 0x106101978 state=running>
Scheduled for IN: <Future at 0x1061015f8 state=pending>
Scheduled for US: <Future at 0x1061013c8 state=pending>
<Future at 0x106101978 state=finished returned str> result: 'ID'
<Future at 0x106048cc0 state=finished returned str> result: 'CN'
<Future at 0x1060f4828 state=finished returned str> result: 'BR'
<Future at 0x1061013c8 state=finished returned str> result: 'US'
<Future at 0x1061015f8 state=finished returned str> result: 'IN'

5 flags downloaded in 0.13s
```

7 Example: Error Handling with `as_completed`

7.0.1 Features of `flags2` Example

`flags2_threadpool.py` introduces several new features: error handling for HTTP requests; a progress bar for completed tasks; etc. (See *Fluent Python*, Ch 17 for all features).

This usage will attempt to get all two-letter codes that start with A, B, or C. Not all these codes are valid countries.

```
In [4]: %run flags2_threadpool.py -s REMOTE a b c
```

```
  3%|          | 2/78 [00:00<00:04, 16.27it/s]

REMOTE site: http://flupy.org/data/flags
Searching for 78 flags: from AA to CZ
30 concurrent connections will be used.


100%|| 78/78 [00:00<00:00, 107.63it/s]

--------------------
43 flags downloaded.
35 not found.
Elapsed time: 0.79s
```

7.0.2 Error Handling in `flags2`

Here, we will look at how error handling is implemented. The error handling can be seen with the `-v` option.

```
In [5]: %run flags2_threadpool.py -v -s REMOTE a b c

REMOTE site: http://flupy.org/data/flags
Searching for 78 flags: from AA to CZ
30 concurrent connections will be used.
ABAH AAAIAC not found  not foundAK not found
not found
 BC
AJAQAPnot found
  AS not found
ANnot foundAX not foundnot found AW

AYAV
 not found not found
AOnot found
AE
AG
not foundnot foundAR not found
ATnot found OK

AD OKAMAF
BABD OK


 OKOK
OKOKOKOKOKBB
```

```
AL

AU

AZ OK
   OKOKOK


BI OK
BLBMBH not found
 OK
 not found
BJBF OK
 OK
BE BPOK
 not found
BG OK
BN OK
BKBS not found
BQ OK
 not found
BO OK
BT OK
BR OK
BX not found
CBCEBVCC not found
BY not found
 CF OK
not found
CHCA OK
BW  CG not found
 OK
OK
OK BUOK

 not found
BZ OK
CD OK
CK not found
CI OK
CLCPCJ   not found
OK
not found
CSCN not found
 OK
CO OK
CM OK
```

```
CQ not found
CR OK
CW not found
CU OK
CT not found
CXCV not found
 OK
CZ OK
CY OK
--------------------
43 flags downloaded.
35 not found.
Elapsed time: 0.27s
```

7.0.3 Raising Errors in `get_flag()`

Here, `resp` is a `Response` object. The response has a stored `HTTPError`, if one has occurred. If the response's status code is not 200, then raise the stored `HTTPError`.

```
def get_flag(base_url, cc):
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    resp = requests.get(url)
    if resp.status_code != 200:
        resp.raise_for_status()
    return resp.content
```

7.0.4 Handling Exceptions in `download_one()`

`download_one()` will be the tasks that we submit. It handles HTTP errors using this structure

```
Result = namedtuple('Result', 'status data')
def download_one(cc, base_url, verbose=False):
    try:
        image = get_flag(base_url, cc)
    except requests.exceptions.HTTPError as exc:
        # ...
    else:
        # ...
    return Result(status, cc)
```

7.0.5 Handling Exceptions in `download_one()`, cont

If everything is okay, we save the image and manually set the `status` that will be returned.

```
    else:
        save_flag(image, cc.lower() + '.gif')
        status = HTTPStatus.ok
```

7.0.6 Handling Exceptions in `download_one()`, cont

We allow a "404 not found" error to pass and set the `status` that will be returned. Otherwise, we reraise the `HTTPError`.

```
    except requests.exceptions.HTTPError as exc:
        res = exc.response
        if res.status_code == 404:
            status = HTTPStatus.not_found
        else:
            raise
```

7.0.7 Submitting tasks in `download_many()`

When `download_many()` queues up the tasks, it does **no exception handling**:

```
def download_many(cc_list, base_url, verbose, concur_req):
    with futures.ThreadPoolExecutor(max_workers=concur_req) as executor:
        to_do_map = {}
        for cc in sorted(cc_list):
            future = executor.submit(download_one, cc, base_url, verbose)
            to_do_map[future] = cc
        # ... etc. ...
```

7.0.8 Getting results in `download_many()`

The exception handling happens when we get the results. We explicitly set `status` rather than propogating the error.

```
    with futures.ThreadPoolExecutor(max_workers=concur_req) as executor:
        # ... We have all the Futures in to_do_map ...
        for future in futures.as_completed(to_do_map):
            try:
                res = future.result()
            except (requests.exceptions.HTTPError,
                    requests.exceptions.ConnectionError) as exc:
                status = HTTPStatus.error
            else:
                status = res.status
```

8 Using `add_done_callback()`

8.0.1 Using `add_done_callback()`

Rather than processing `Futures` as completed, we can attach the post-processing directly to the `Future`!

`future.add_done_callback(fn)` attaches a callable (`fn`) to the future: * `fn` will be called when the future is completed or cancelled * `fn` will be passed the future as its only argument.

8.0.2 Using `add_done_callback()`, cont.

A tiny example:

```
def when_done(r):
    print('Got:', r.result())

with ProcessPoolExecutor() as pool:
    future_result = pool.submit(work, arg)
    future_result.add_done_callback(when_done)
```

9 Threads, Queues, and Locks

9.0.1 The `Thread` Object

The `threading` module provides a low-level `Thread` object that runs a given callable:

```
t = Thread(target=my_callable, args=(my, args))
```

You must explicitly start and stop a thread. Stopping a thread blocks the caller until the thread is complete (or times-out).

```
t.start()
# ... do some other stuff ...
t.stop(timeout=600)
```

You can also query whether a thread is alive (i.e., not completed) using `t.is_alive()`. This is non-blocking.

9.0.2 Example: Making a Crummy Little Threadpool

Let's try to run this worker on a bunch of threads...

```
In [6]: import random
        def worker():
            print('Doing some work...', flush=True)
            sleepy_time = random.randint(1, 10)
            time.sleep(sleepy_time)
            print('Done after {} sec!'.format(sleepy_time), flush=True)
```

9.0.3 Example: Making a Crummy Little Threadpool, cont.

```
In [7]: from threading import Thread

        # Start 4 threads
        threads = []
        for i in range(4):
            t = Thread(target=worker)
            t.start()
```

```
        threads.append(t)

    # Finish up
    for t in threads:
        t.join()
```

```
Doing some work...
Doing some work...
Doing some work...
Doing some work...
Done after 2 sec!
Done after 6 sec!
Done after 7 sec!
Done after 8 sec!
```

9.0.4 The `Queue` Object

The `Queue` object allows threads to communicate in a thread-safe manner. This is very useful and something we didn't explore earlier.

We assume this worker is passed a `Queue`. The `put()` method is blocking by default and has an optional timeout.

```
from queue import Queue

def worker(item, queue):
    result = do_work(item)
    queue.put(result)
```

9.0.5 The `Queue` Object, cont.

Now we start our workers:

```
results = Queue()
threads = []
for i in range(4):
    t = Thread(target=worker, args=(i, results))
    t.start()
    threads.append(t)
```

9.0.6 The `Queue` Object, cont.

Now we end our workers and get the results. The `get()` method is blocking by default and has an optional timeout.

```
for i in range(4):
    print(results.get())

for i in range(4):
    threads.join()
```

9.0.7 The `Lock` Object

For more unstructured synchronization, you can use the `Lock` class. A `Lock` has `acquire()` and `release()` methods. They can be called explicitly or implicitly in a context manager.

Consider this worker:

```
count = 0

def add_to_count(lock):
    global count
    with lock:
        count += 1
```

9.0.8 The `Lock` Object, cont.

Now we instantiate a `Lock` and start up our threads:

```
lock = threading.Lock()
threads = []

for i in range(N):
    t = Thread(target=add_to_count, args=(lock,))
    t.start()
    threads.append(t)
```

9.0.9 The `Lock` Object, cont.

Now we join our threads and see our counter

```
for t in threads:
    t.join()
print(count)
```

```
In [8]: import threading
        count = 0
        N = 100

        def add_to_count(lock):
            global count
            with lock:
                count += 1

        lock = threading.Lock()
        threads = []

        for i in range(N):
            t = Thread(target=add_to_count, args=(lock,))
            t.start()
            threads.append(t)
```

```python
    for t in threads:
        t.join()
    print(count)
```

```
100
```