Concurrency in Python

Concepts, frameworks and best practices

PyCon DE

Stefan Schwarzer, SSchwarzer.com
info@sschwarzer.com

Karlsruhe, Germany, 2018-10-26

About me

- Using Python since 1999
- Software developer since 2000
- Freelancer since 2005
- Book “Workshop Python”, Addison-Wesley, using the then brand new Python 2.2 ;-)
- About 15 conference talks
- Maintainer of ftputil (high-level FTP client library) since 2002

Overview

- Basics
- Concurrency approaches
- Race conditions
- Deadlocks
- Queues
- Higher-level concurrency approaches
- Best practices

Basics

reasons, terms

Reasons for concurrency

- CPU intensive tasks
  Speed up algorithms by executing parts in parallel.
- Input/output
  Other parts of the program can run while waiting for I/O.
- Reactivity
  While a GUI application executes some lengthy operation, the application should still accept user interaction.

Terms

- Resource
  Anything that's used by an execution thread (not necessarily an OS thread), for example simple variables, data structures, files or network sockets.
- Concurrency
  There are multiple execution threads. They don't have to progress at the same time.
- Parallelism
  Execution threads run at the very same time (for example on different CPU cores).
- Atomic operation
  A task that can't be interrupted by another execution thread

Concurrency approaches

multithreading, multiprocessing, event loop

Concurrency approaches

Multithreading

- Concurrency of OS threads in a single process
- Module `threading` in the standard library
- Threads can share data in process memory
- For CPython the global interpreter lock (GIL) applies
- The GIL prevents the parallel execution of Python code.
  The GIL is released during I/O operations.
  Also, C extensions can release the GIL.

Concurrency approaches

Multiprocessing

- Concurrency of OS processes
- Module `multiprocessing` in the standard library
- Data transfer between processes via messages or shared memory
- When transferring messages, they must be serialized. This is additional work.
- Advantage of multiprocessing: no limitation of parallel execution, not even for CPU-limited work. The GIL is per Python process.

Concurrency approaches

Event loop

- Loop ("main loop") detects events (examples: mouse click, incoming network data)
- Variants:
  - Depending on the event, a "handler" is called and processes the event. Control returns to the main loop after the handler execution.
  - Code looks sequential, but execution is switched to other code if the event loop has to wait for I/O.
  - Both variants may be used in the same program.
- An event loop implementation is in the package `asyncio` in the standard library.

Race conditions

definition, code example, explanation, fix

Race conditions

Definition

While a resource is modified by an execution thread,
another execution thread modifies or reads the resource.

Race conditions

Code without protection against concurrent access

```
import threading, time  # ‘sys.setswitchinterval‘ omitted

counter = 0
def count():
    global counter
    for _ in range(100):
        counter += 1


threads = []
for _ in range(100):
    thread = threading.Thread(target=count)
    thread.start()  # Start thread. Don’t confuse with ‘run‘.
    threads.append(thread)
for thread in threads:
    thread.join()  # Wait until thread is finished.
print("Total:", counter)
```

Race conditions

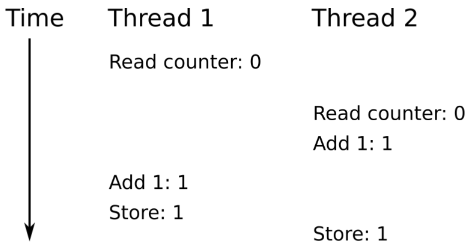
Output without protection against concurrent access

```
$ python3 race_condition.py
Total: 9857
$ python3 race_condition.py
Total: 9917
$ python3 race_condition.py
Total: 9853
$ python3 race_condition.py
Total: 9785
$ python3 race_condition.py
Total: 9972
$ python3 race_condition.py
Total: 9731
```

Race conditions

Explanation – race condition because of concurrent access

This is only one of many possibilities.



Thread 2 reads the earlier value of `counter` because thread 1 hasn't stored the new value yet.

Race conditions

Code with protection against concurrent access

```
import threading, time  # ‘sys.setswitchinterval‘ omitted

counter = 0
lock = threading.Lock()

def count_with_lock():
    global counter
    for _ in range(100):
        with lock:
            counter += 1  # Atomic operation


threads = []
for _ in range(100):
    thread = threading.Thread(target=count_with_lock)
    thread.start()
    threads.append(thread)
...
```

Deadlocks

definition, code example

Deadlocks

Definition

A deadlock happens if execution threads mutually
claim resources that the other execution threads need.

Example:

- Both thread 1 and 2 need resources A and B to finish a task.
- Thread 1 already holds resource A and wants resource B.
- Thread 2 already holds resource B and wants resource A.

→ Deadlock!

Deadlocks

Example code

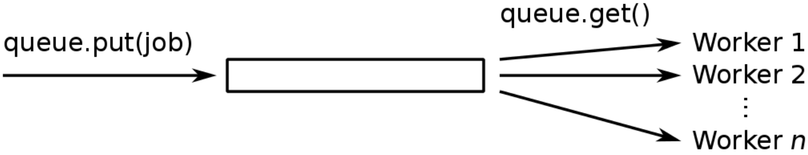
```
# Thread 1
with input_lock:        # 1st
    with output_lock:  # blocks
        input_line = input_fobj.readline()
        # Process input ...
        output_fobj.write(output_line)


# Thread 2
with output_lock:       # 2nd
    with input_lock:   # blocks
        input_line = input_fobj.readline()
        # Process input ...
        output_fobj.write(output_line)
```

Queues

code example with worker threads

Queues

Schema for the following example



Principle: `put` and `get` are atomic operations.

Queues

Setup

```
import logging, queue, random, threading, time

logging.basicConfig(level=logging.INFO, format="%(message)s")
logger = logging.getLogger("queue_example")

WORKER_COUNT = 10
JOB_COUNT = 100
# Needed to shut down threads without race conditions.
STOP_TOKEN = object()
job_queue = queue.Queue()


class Job:

    def __init__(self, number):
        self.number = number
```

Queues

Worker thread

```
class Worker(threading.Thread):

    def run(self):
        while True:
            job = job_queue.get(block=True)
            if job is STOP_TOKEN:
                break
            self._process_job(job)

    def _process_job(self, job):
        # Wait between 0 and 0.01 seconds.
        time.sleep(random.random() / 100.0)
        # Atomic output
        logger.info("Job number {:d}".format(job.number))
```

Queues

Creation and execution of jobs

```
def main():
    workers = []
    # Create and start workers.
    for _ in range(WORKER_COUNT):
        worker = Worker()
        worker.start()
        workers.append(worker)
    # Schedule jobs for workers.
    for i in range(JOB_COUNT):
        job_queue.put(Job(i))
    # Schedule stopping of workers.
    for _ in range(WORKER_COUNT):
        job_queue.put(STOP_TOKEN)
    # Wait for workers to finish.
    for worker in workers:
        worker.join()
```

Higher-level concurrency approaches

`concurrent.futures`, active objects, process networks

concurrent.futures

Example

```
import concurrent.futures
import logging
import random
import time


WORKER_COUNT = 10
JOB_COUNT = 100


class Job:

    def __init__(self, number):
        self.number = number
```

concurrent.futures

Example

```
def process_job(job):
    # Wait between 0 and 0.01 seconds.
    time.sleep(random.random() / 100.0)
    # Atomic output
    logger.info("Job number {:d}".format(job.number))


def main():
    with concurrent.futures.ThreadPoolExecutor(
           max_workers=WORKER_COUNT) as executor:
        # Distribute jobs.
        futures = [executor.submit(process_job, Job(i))
                      for i in range(JOB_COUNT)]
        # Wait for work to finish.
        for future in concurrent.futures.as_completed(futures):
            pass
```

concurrent.futures

Comparison with queue example

- `process_job` is now a function, no need to inherit from `threading.Thread` and implement `run`
- No queue needed
- No error-prone token handling needed to stop the workers at the right time

→ Use `concurrent.futures` if you can! :-)

Active objects

- Principle: Locks, queues or other synchronization mechanisms are not part of the API of an object.
- Synchronization, as far as needed, is hidden in high-level methods.

Active objects

Example – constructor

```
import queue
import threading


STOP_TOKEN = object()


class Adder:

    def __init__(self):
        self._in_queue = queue.Queue()
        self._out_queue = queue.Queue()
        self._worker_thread = threading.Thread(
                                    target=self._work)
        self._worker_thread.start()
```

Active objects

Example – internal method

```
def _work(self):
    while True:
        work_item = self._in_queue.get(block=True)
        if work_item is STOP_TOKEN:
            break
        result = work_item + 1000
        self._out_queue.put(result)
```

Active objects

Example – public methods

```
def submit(self, work_item):
    self._in_queue.put(work_item)

def next_result(self):
    return self._out_queue.get(block=True)

def stop(self):
    self._in_queue.put(STOP_TOKEN)
    self._worker_thread.join()
```

Active objects

Example – usage

```
def main():
    ITEM_COUNT = 100
    adder = Adder()
    for i in range(ITEM_COUNT):
        # Doesn't block
        adder.submit(i)
    # Do other things.
    # ...
    # Collect results.
    for _ in range(ITEM_COUNT):
        # May block
        print(adder.next_result())
    # May block
    adder.stop()
```

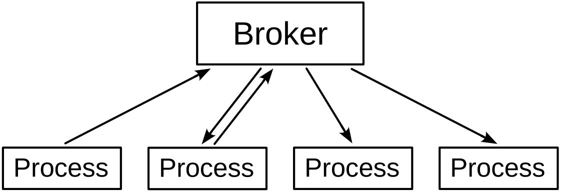
Process networks

- Processes receive input data and/or send output data.
- Data transfer between processes by message passing
- Processes can use different programming languages if they use a message format that the communicating processes understand.
- Some overhead due to data serialization and protocols

Process networks

With broker

- Processes communicate with a broker service, but not with each other.

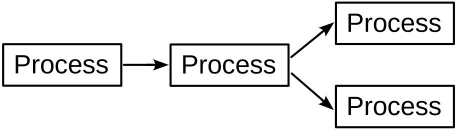


- Broker protocol examples: AMQP, MQTT
- Declarative configuration
- Message persistence (optional)

Process networks

Without broker

- Processes communicate directly.



- Example: ZeroMQ

Best practices

caveats, general design advice, approaches, shared state

Best practices

Caveats

- The following "best practices" aren't necessarily written down in books or online, but are my recommendations.
- Different advice may apply to different areas of your code.

Best practices

General design advice

- Concurrency is an optimization.
  Like other optimizations, use it only if necessary.
- Try to keep code simple and easy to understand.
  In many cases this would mean queues or higher-level APIs to communicate between threads or processes.
- If you use low-level APIs, hide them. Don't make locks, queues etc. a part of the public interface.

Best practices

Choose a concurrency approach

- I/O-limited concurrency
  multithreading
  asyncio (for many concurrent tasks)
  process networks
- CPU-limited concurrency
  multiprocessing
  multithreading (if using extensions that can release the GIL)
  process networks
- GUI frameworks
  usually come with their own event loop
- Concurrent processes in different languages
  process networks

Best practices

Shared state

- Be extremely careful not to read shared state while it may be written. Even query methods may be problematic if they implicitly update an internal cache of an object, for example.
- Make sure the APIs you use from multiple threads are thread-safe. You can only count on the documentation because the code may be different in the next version.
- Try to avoid shared state. Pass immutable objects or set up the state before starting threads that access the state.
- Concurrency involving shared state is difficult to test. Don't assume your code doesn't have concurrency issues only because it seems to run fine. Invest some time to create a solid design. Have your code reviewed.

Thank you for your attention! :-)

Questions?

Remarks?

Discussion?

info@sschwarzer.com

https://sschwarzer.com

Appendices

links, asyncio example

Links

- Dr. Dobb's Parallel Computing
  http://www.drdobbs.com/parallel (overview page)
  http://www.drdobbs.com/212903586 (introduction)
- "The problem with threads"
  https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf
- Design recommendations
  https://stackoverflow.com/questions/1190206/, especially
  https://stackoverflow.com/questions/1190206/threading-in-python/1192114#1192114
- Active object pattern
  http://www.drdobbs.com/225700095
- "Notes on structured concurrency"
  https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful

asyncio

Example – Setup

```python
import asyncio
import logging
import random


logging.basicConfig(level=logging.INFO, format="%(message)s")
logger = logging.getLogger("asyncio_example")


JOB_COUNT = 100


class Job:

    def __init__(self, number):
        self.number = number
```

asyncio

Example – asynchronous code

```
async def process_job(job):
    # Wait between 0 and 0.01 seconds.
    await asyncio.sleep(random.random() / 100.0)
    logger.info("Job number {:d}".format(job.number))


def main():
    loop = asyncio.get_event_loop()
    tasks = []
    for i in range(JOB_COUNT):
        task = loop.create_task(process_job(Job(i)))
        tasks.append(task)
    for task in tasks:
        # Similar to ‘Thread.start‘ plus ‘Thread.join‘
        loop.run_until_complete(task)
    loop.close()
```