MODULE 5

THREAD PROGRAMMING

Threads

> What are threads?

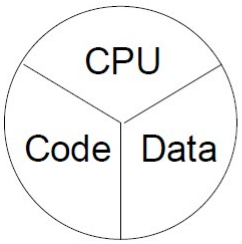
–Threads are a virtual CPU.

> The three parts of at thread are:

– CPU

– Code

– Data



A thread or execution context

Starting a New Thread

> To start a separate thread, you create a Thread instance and then tell it to **`.start()`:**

```
import threading
import logging
import time

def print_time(threadName, delay):
    logging.info("Thread %s: starting", threadName)
    count = 0
    while count < 3:
        time.sleep(delay)
        count += 1
        logging.info("%s: is running" % (threadName))
    logging.info("Thread %s: finishing", threadName)
```

Starting a New Thread

```
# Create two threads as follows
try:
    t1 = threading.Thread(target=print_time, args=("Thread-1", 2,))
    t2 = threading.Thread(target=print_time, args=("Thread-2", 4,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
except:
    logging.error("Error: unable to start thread")
logging.info("done.")
```

Daemon Threads

> Python threading has a more specific meaning for daemon.
> A daemon thread will shut down immediately when the program exits.
> If a program is running Threads that are not daemons, then the program will wait for those threads to complete before it terminates.
> Threads that are daemons, however, are just killed wherever they are when the program is exiting.

```
x = threading.Thread(target=fun, args=(1,), daemon=True)
```

join() a Thread

> To tell one thread to wait for another thread to finish, you call **`.join()`**.

```
from threading import Thread
from random import randint

lottery_numbers = []

def draw_numbers(max, size):
    numbers = set()
    while len(numbers) < size:
        numbers.add(randint(1, max))
    numbers = list(numbers)
    numbers.sort()
    lottery_numbers.append(numbers)

threads = []
for i in range(0, 100):
    threads.append(Thread(target=draw_numbers, args=(50, 6)))
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
for numbers in lottery_numbers:
    print(numbers)
print(len(lottery_numbers))
print("done.")
```

Using a ThreadPoolExecutor

- Constructing a new thread is somewhat expensive because it involves interaction with the operating system.
- If your program creates a large number of short-lived threads, then it should instead use a thread pool.
- A thread pool contains a number of idle threads that are ready to run.
- You give a Runnable to the pool, and one of the threads calls the run method.
- When the run method exits, the thread doesn't die but stays around to serve the next request.

Using a ThreadPoolExecutor

```
from random import randint

from concurrent.futures.thread import ThreadPoolExecutor

futures = []
lottery_numbers = []


def draw_numbers(max, size):
    numbers = set()
    while len(numbers) < size:
        numbers.add(randint(1, max))
    numbers = list(numbers)
    numbers.sort()
    return numbers
```

Using a ThreadPoolExecutor

```
with ThreadPoolExecutor(max_workers=8) as executor:
    for i in range(0,100):
        futures.append(executor.submit(draw_numbers, 50, 6))


for ft in futures:
    lottery_numbers.append(ft.result())

for numbers in lottery_numbers:
    print(numbers)

print("Done.")
```

Race Conditions

> Race conditions can occur when two or more threads access a shared piece of data or resource.

```
import os
from threading import Thread
counter = 0
threads = []

def fun():
    global counter
    for i in range(0, 500000):
        counter = counter + 1

for i in range(0, os.cpu_count()):
    t = Thread(target=fun)
    t.start()
    threads.append(t)
for t in threads:
    t.join()
print(counter)
```

Basic Synchronization Using **Lock**

```
import os
from threading import Thread, Lock

counter = 0
threads = []

v = Lock()


def fun():
    global counter
    with v:
        for i in range(0, 500000):
            counter = counter + 1
```

Thread Safe Class

```
class Account:
    def __init__(self, iban, balance=100.0):
        self.iban = iban
        self.balance = balance
        self._lock = Lock()

    def withdraw(self, amount):
        with self._lock:
            if amount <= 0:
                raise ValueError("amount must be positive.")
            if amount > self.balance:
                raise InsufficientBalance("negative balance.", amount - self.balance)
            self.balance -= amount

    def deposit(self, amount):
        with self._lock:
            if amount <= 0:
                raise ValueError("amount must be positive.")
            self.balance += amount
```

How to Use Thread Locks in Python

> In Python, thread locks are implemented using the **`threading.Lock()`** class.
> To use a lock in your code, you first need to create an instance of the **`Lock()`** class:

```
import threading

lock = threading.Lock()
```

How to Use Thread Locks in Python

> Once you have created a lock instance, you can use it to protect a shared resource.
> To acquire the lock, you use the **`acquire()`** method:

```
lock.acquire()
```

> This will block the thread until the lock becomes available.
> Once the lock is acquired, the thread can access the shared resource safely.

How to Use Thread Locks in Python

> When the thread is done accessing the resource, it must release the lock using the **`release()`** method:

```
lock.release()
```

> This will release the lock and allow other threads to access the shared resource.

How to Use Thread Locks in Python

> It's important to note that when using locks, you must ensure that you release the lock in all possible code paths.

> If you acquire a lock and then exit the function without releasing the lock, the lock will remain locked, preventing other threads from accessing the shared resource.

> To avoid this, it's a good practice to use a try-finally block to ensure that the lock is released, even if an exception occurs:

```
lock.acquire()
try:
    # access the shared resource
finally:
    lock.release()
```

What are Thread Semaphores?

> Thread semaphores are another synchronization mechanism used in multithreaded applications.
> Semaphores are like thread locks in that they prevent race conditions by allowing only a limited number of threads to access a shared resource at a time.
> However, semaphores offer more flexibility than locks by allowing multiple threads to access the shared resource simultaneously, up to a specified limit.

What are Thread Semaphores?

> Thread semaphores are synchronization mechanisms that allow a limited number of threads to access a shared resource simultaneously.
> When a thread wants to access the shared resource, it must first acquire a semaphore.
> Each semaphore has a specified limit that determines how many threads can access the resource simultaneously.
> Once the limit is reached, any additional threads that try to acquire the semaphore will be blocked until a thread releases the semaphore.

How do Thread Semaphores Work?

- Thread semaphores work by allowing a limited number of threads to access a shared resource simultaneously.
- When a thread wants to access the resource, it must first acquire a semaphore.
  - If the semaphore is available and the limit has not been reached, the thread can acquire the semaphore and access the resource.
  - If the semaphore is not available or the limit has been reached, the thread will be blocked until a semaphore becomes available.
- Once a thread has acquired a semaphore, it can access the shared resource simultaneously with other threads that have also acquired the semaphore.
- When the thread is done accessing the resource, it must release the semaphore so that other threads can acquire it and access the resource.

How do Thread Semaphores Work?

- Thread semaphores can be used to protect any shared resource that is accessed by multiple threads, including variables, files, and network connections.

Example (1/2)

```
import threading
from time import sleep

items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Limiting the number of threads that can access the list simultaneously to 4
semaphore = threading.Semaphore(value=4)

def process_item(item):
    semaphore.acquire()                  # acquire the semaphore
    try:
        sleep(3)                         # simulate some processing time
        print(f'Processing item {item}') # process the item
    finally:                             # Make sure we always release the semaphore
        semaphore.release()              # release the semaphore
```

Example (2/2)

```
# create a list of threads to process the items
threads = [
    threading.Thread(target=process_item, args=(item,))
    for item in items
]
[thread.start() for thread in threads]      # start all threads
[thread.join() for thread in threads]       # wait for all threads to finish
```

Synchronizing Threads with Barriers

- In a multithreaded application, it's common for multiple threads to work together to accomplish a task.
- However, there are certain situations where a thread must wait for other threads to reach a certain point in the execution before proceeding.
- This is where thread barriers come into play.
- A thread barrier is a synchronization mechanism that allows threads to wait for each other to reach a certain point in the execution before continuing.
- Thread barriers can be used to ensure that all threads have completed a certain task before moving on to the next one, or to coordinate the order of execution of multiple threads.

How Thread Barriers Work

- Thread barriers work by blocking threads until a certain number of threads have reached the barrier.
- Once the required number of threads have reached the barrier, all the blocked threads are released and allowed to continue execution.
- A thread barrier typically has a count, which specifies the number of threads that must reach the barrier before the threads are released.
- When a thread reaches the barrier, it calls the **`wait()`** method on the barrier object.
- If the required number of threads have not yet reached the barrier, the calling thread is blocked.
- Once the required number of threads have reached the barrier, all the blocked threads are released and allowed to continue execution.

Example (1/2)

```
import threading

# create a barrier requiring 2 threads to reach the barrier
barrier = threading.Barrier(2)

def worker():
    print('Worker started')
    # do some work
    print('Worker reached the barrier')
    # wait for other threads to reach the barrier
    barrier.wait()
    # continue work
    print('Worker finished')
```

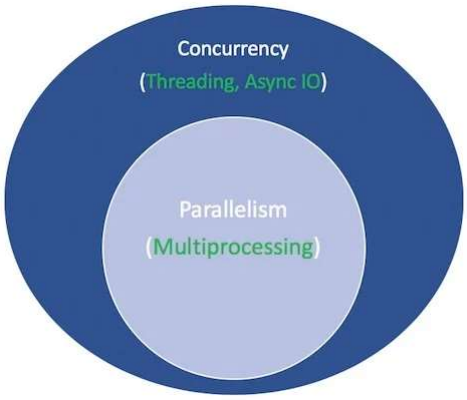
Example (2/2)

```
# create two threads
t1 = threading.Thread(target=worker)
t2 = threading.Thread(target=worker)

# start the threads
t1.start()
t2.start()

# wait for the threads to finish
t1.join()
t2.join()
```

Concurrency and Parallelism



Blocking/Synchronous

```
import time


def count():
    print("One")
    time.sleep(1)
    print("Two")


def main():
    for _ in range(3):
        count()


if __name__ == "__main__":
    s = time.perf_counter()
    main()
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

The async/await Syntax

```
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")


async def main():
    await asyncio.gather(count(), count(), count())


if __name__ == "__main__":
    import time

    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```