Concurrency / Multithreading

An Introduction to Concurrency

CS 5012

UNIVERSITY OF VIRGINIA
DATA SCIENCE
INSTITUTE

Introduction to Threads

- *Often it is useful for a program to carry out two or more tasks at the same time. This can be achieved by implementing **threads***
- **Thread**: a program unit that is executed independently of other parts of the program
- Each thread in the program is executed for a short amount of time [“time slice”]
- This gives the ***impression*** of parallel execution
- If a computer has multiple central processing units (CPUs), then some of the threads can run in parallel, one on each processor
- Therefore don’t worry about not having access to multiple CPUs – programs that we’ll run and this theory is still relevant and important!

Threads

- Thread scheduler: runs each thread for a short amount of time (a **time slice**)
- Then the scheduler activates another thread
- There will ***always be slight variations in running times*** – especially when calling operating system services (e.g. input and output)
- THERE IS NO GUARANTEE ABOUT THE ORDER IN WHICH THREADS ARE EXECUTED!

Basic Concurrency elements in Python Code

- **Imports**
  - `import time`
    - Often threads “go to sleep” for a period of time, and this is achieved via `time.sleep(s)`, where ‘s’ is time in seconds (e.g. `time.sleep(5)`)
  - `from threading import Thread`
    - From the ‘*threading*’ class import the *Thread* method that is responsible for creating a new thread

Basic Concurrency elements in Python Code

- **Create a thread "t"**
  - `t = Thread (…)`
- A thread has a **specific target function** that could have parameters, e.g. myfunc(i,param)
  - `t = Thread(target=myfunc, args=(i,param))`
    - Creating thread 't' and instructing it to run statements in the target function (`myfunc`) using the supplied parameter arguments (i and param)
- **Start the thread "t"**
  - `t.start()`
  - Simply call the start() method – thread executes tasks in target

[Run This!]
Python Script Example

example1.py

Simple script that shows how to create threads. Also demonstrates the fact that you cannot guarantee the order in which threads will run (observe the output!) Notice how it is not a perfect interleaving of "hello" and "goodbye".

Run this script using the Anaconda prompt and observe the output. Take note of what you see!

[Run This!]
Python Script Example

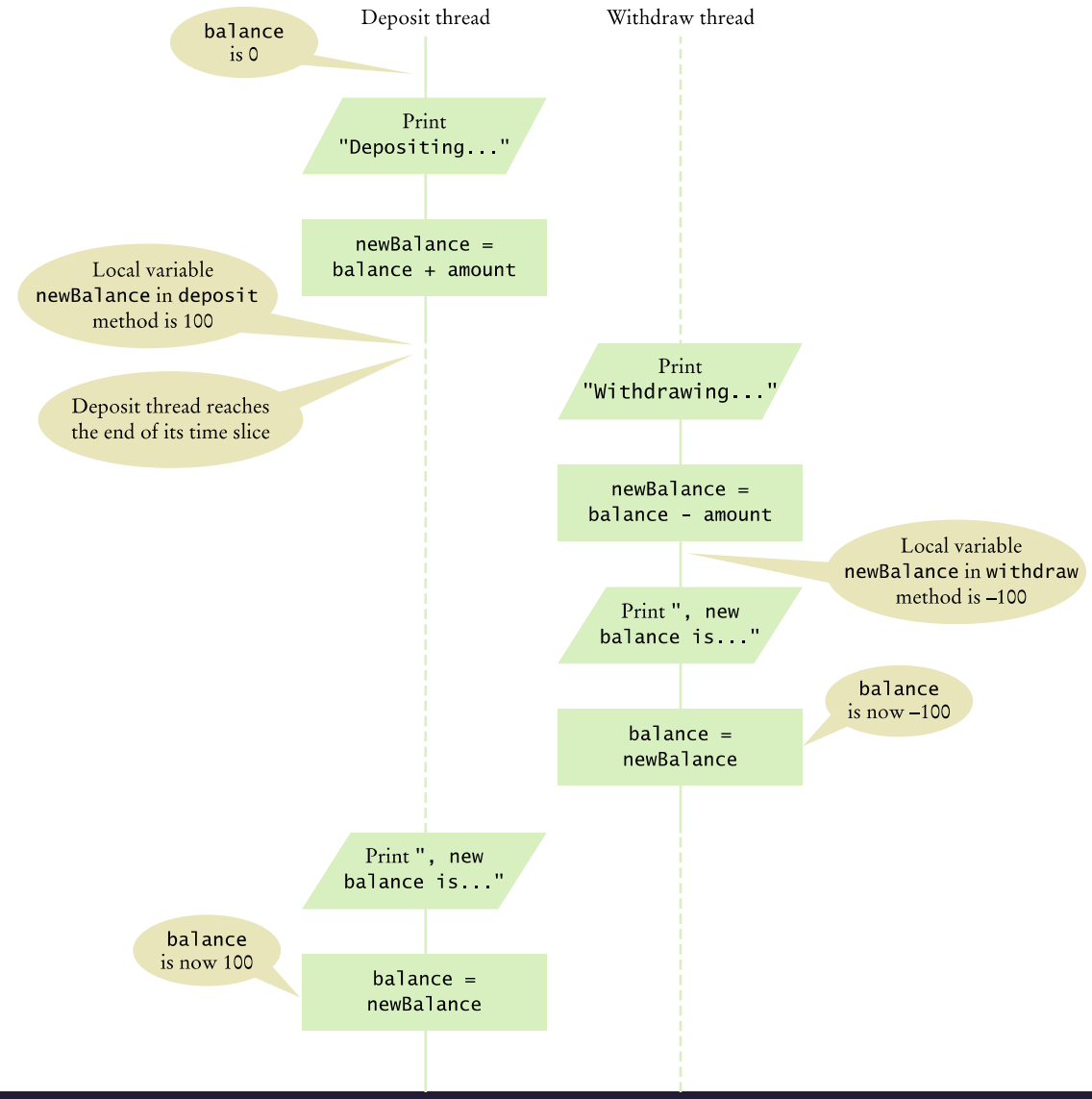
example2.py

This script invokes threads to count words within "books" (text files). Each thread counts the words of one book and reports the number of words.
Run this script using the Anaconda prompt and observe the output. Take note of what you see!

Race Conditions

- When threads share a *common object*, they can conflict with each other
- Sample program: **multiple threads manipulate a bank account**
- Create two sets of threads:
  - Each thread in the first set repeatedly ***deposits*** $100
  - Each thread in the second set repeatedly ***withdraws*** $100
- **Reminder**: no guarantee about the order threads are executed
- **Ideal case**: balance would always be $0.00 – *but does it happen?*
  (run the next Python script and find out!)

Corrupting the Contents of "balance"



Race Condition

- Occurs if the effect of multiple threads on shared data *depends on the order in which they are scheduled*
- It is possible for a thread to reach the *end of its time slice in the middle of a statement*
- It may evaluate the right-hand side of an equation but not be able to store the result until its next turn:
  - balance = the right-hand-side value

[Run This!]
Python Script Example

example3.py

This script simulates a bank, where threads deposit $100 and threads withdraw $100. This script illustrates the occurrence of race conditions

Run this script using the Anaconda prompt and observe the output. Take note of what you see!

Synchronizing Object Access

- To solve such problems as the one just seen, use a ***lock object***
- **Lock object**: used to control threads that manipulate shared resources
- Typically, a lock object is added to a class whose methods access shared resources
- Try…Finally block is used
  - Incase an **exception** is thrown after the lock is acquired the unlock will *never* happen!
  - So if the unlock is placed in the **Finally** block, then no matter what, the unlock will occur and give other threads the ability to acquire the lock and manipulate the shared resource

Lock and Try-Finally elements in Python Code

- File imports threading
  - `import threading`
- Create **a lock object** (let's simply call it "lock") by calling `threading.Lock()`
  - `self.lock = threading.Lock()`
- This allows for the lock object to **lock** ("acquire") and **unlock** ("release")
  - To lock: `self.lock.acquire()`
  - To unlock: `self.lock.release()`

Lock and Try-Finally elements in Python Code

- **Try/Finally** block structure
  - Lock before the "`try`" block and unlock in the "`finally`" block to ensure unlock always happens
  - Introducing locks within the try-finally structure will prevent race conditions – one thread cannot interrupt another thread until all the work is done. However, if an exception occurs, the lock is successfully released since that statement (release of lock) is in the finally block (otherwise, that thread, if interrupted, would hold on to the lock and cause problems)

Lock and Try-Finally elements in Python Code

- Code that manipulates shared resource is surrounded by calls to acquire and release (lock and unlock)
- What if an exception occurs…? *The lock would never be released!*

```
class BankAccount:
   def deposit(self, amount):
      self.lock.acquire()
           … stuff with shared resource …
      Exception thrown here… code afterwards does NOT execute
      self.lock.release() # never gets executed!
```

Elements in Python Code (better solution)

- Code that manipulates shared resource is surrounded by calls to lock (“acquire”) and unlock (“release”); and try-finally structure
- Important to unlock (“release”) in the finally block

```
class BankAccount:
    def __init__(self):
            self.lock = threading.Lock()
            self.balance = 0
    def deposit(self, amount):
            self.lock.acquire()
            try:
                … stuff with shared resource …
                                    Exception could be thrown here
            finally:
                self.lock.release() # will always be executed!
```

Synchronizing Object Access

- When a thread **acquires** a lock, **it owns the lock** until it calls release
- A thread that tries to acquire a lock, while another thread owns the lock, is temporarily ***deactivated***
- Thread scheduler periodically reactivates threads so they can try to acquire the lock
- Eventually, waiting threads can acquire the lock

- In the bank context, adding **locks** will ensure the balance will be zero at the end! Although the balance could have negative values

[Run This!]
Python Script Example

example4.py

This script introduces locks to combat the problem of race conditions. A lock object can lock ("acquire") and unlock ("release"). Also, introduces the try-finally structure.
Run this script using the Anaconda prompt and observe the output. Take note of what you see!

Avoiding Deadlocks

- A **deadlock** occurs if no thread can proceed because each thread is waiting for another to do some work first

- Let's say we add a statement in the withdraw() method such as:

```
while (balance < amount)
        # Wait for balance to grow
```

- This is "fine" however prior to this, the thread acquired the lock!
- Does this mean that while it's "asleep" or "waiting" nobody else can acquire the lock?
  - YES! --- *This is a problem!*
  - Why? --- It will prevent all threads including threads that call the deposit () method to run… so NO funds can be added!

Condition Objects

- To overcome this problem, use a **condition object**
- Condition objects allow a thread to temporarily release a lock, and to regain the lock at a later time

- It is customary to give the condition object a **name** that describes the condition to test

```
def __init__(self):
  self.lock = threading.Lock()
  self.sufficientBalanceCondition=threading.Condition(self.lock)
  self.balance = 0
```

Condition object given a name that *describes* the condition

- You must remember to implement an ***appropriate test*** (`while…`)

```
while(self.balance < amount):
  self.sufficientBalanceCondition.wait()
```

Condition Objects

- Calling `.wait()`
  - As long as the condition is not fulfilled, make the current thread **wait**
  - Allows another thread to acquire the lock object (current thread ***temporarily gives up its lock***)
- ***To unblock***, another thread must execute `.notifyAll()` on the same condition object
- `notifyAll()` unblocks all threads ***waiting on the condition***

- In the bank context, adding **conditions** will prevent balance going below zero. (Balance will also end in zero due to locks.)

Condition Elements in Python Code

- Create condition by using `threading.Condition(…)` and associate it with a particular lock object
  - ```
    self.lock = threading.Lock()
    self.sufficientBalanceCondition = threading.Condition(self.lock)
    ```
- Condition objects allow you to `wait` until a condition is met, usually using a while loop and a condition
  - ```
    while(self.balance < amount):
        self.sufficientBalanceCondition.wait()
    ```
- To unblock another thread must call `notifyAll`
  - ```
    self.sufficientBalanceCondition.notifyAll()
    ```
    ```

[Run This!]
Python Script Example

example5.py

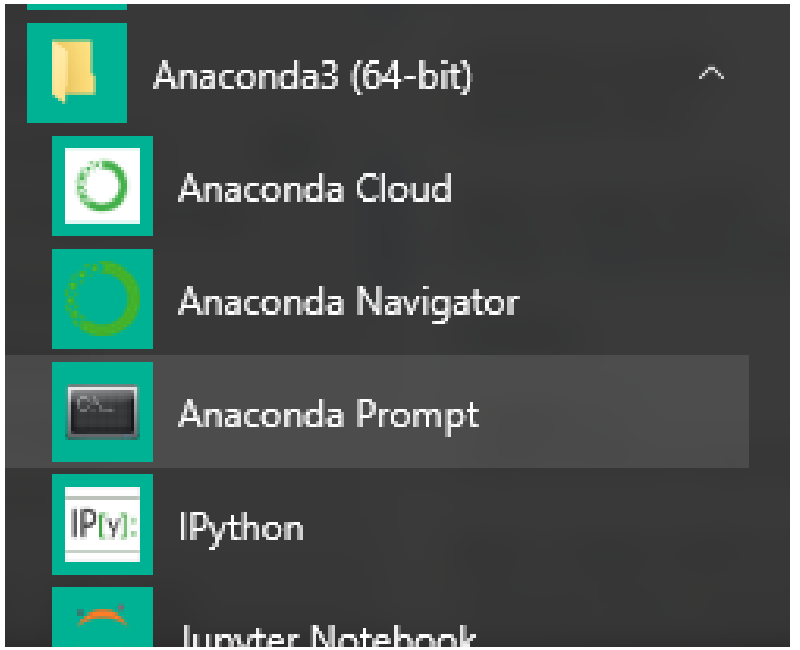
This script introduces a means to avoid deadlock by using condition objects. Condition objects allow a thread to temporarily release a lock, and to regain the lock at a later time. Condition objects are associated with a particular lock object.
Run this script using the Anaconda prompt and observe the output. Take note of what you see!

Running Python Scripts…

- run concurrency python code using the **ANACONDA prompt**

Anaconda Prompt

- Run the Anaconda prompt



- Copy the path of the directory your files are in and “cd” (change directory) to it
- You can type “dir” to see the contents in your current directory

Anaconda Prompt

- To execute a python file, type: **`python <filename>.py`**
  (don't include "<" or ">", simply replace "<filename>" with the name of your file.) Following is example output (last 6 lines) for **example5.py**

```
Withdrawing: 100, new balance is 500

Withdrawing: 100, new balance is 400

Withdrawing: 100, new balance is 300

Withdrawing: 100, new balance is 200

Withdrawing: 100, new balance is 100

Withdrawing: 100, new balance is 0
```