# IEMS 5703
# Network Programming and System Design

## Lecture 3 - Concurrent Programming

**Albert Au Yeung**
**25th January, 2018**

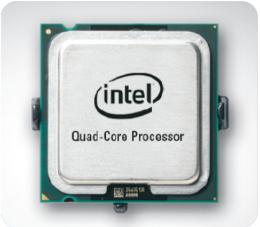
# Agenda

- What is concurrent programming?
- Python's characteristics
- Multi-threading
- Multi-processing
- Message passing and shared memory
- Application in socket programming

# Concurrent Programming

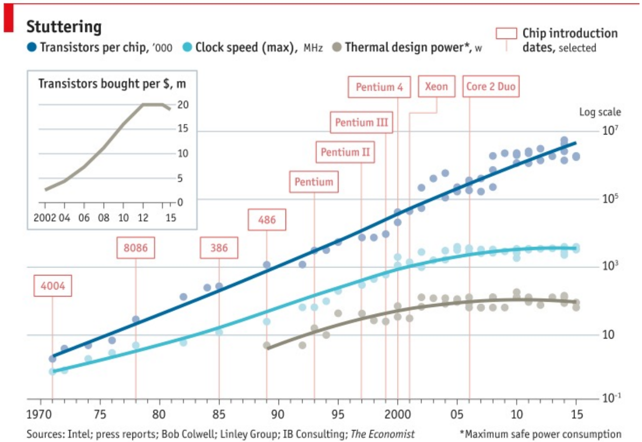
# Concurrent Programming

- Simple programs are usually **sequentially** executed
- **Concurrent**: *Existing or occurring at the same period of time*
- Why do we want to do things at the **same time**?




1. Our processor is very powerful (multi-core processors)
2. Some tasks would take a relative long time to finish:
   - Disk I/O
   - Network calls
   - User input

# Moore's Law



Stuttering
Transistors per chip, '000
Clock speed (max), MHz
Thermal design power*, w
Chip introduction dates, selected
Transistors bought per $, m
20
15
10
5
0
2002 04 06 08 10 12 15
4004
8086
386
486
Pentium
Pentium II
Pentium III
Pentium 4
Xeon
Core 2 Duo
Log scale
$10^7$
$10^5$
$10^3$
10
$10^{-1}$
1970 75 80 85 90 95 2000 05 10 15
Sources: Intel; press reports; Bob Colwell; Linley Group; IB Consulting; *The Economist*
*Maximum safe power consumption

# Concurrent Programming

- **In the past**
  - Parallel computing has been mainly used in very advanced technical computation (high performance computing)
  - Concurrent programming is mainly used on the operating system level
  - Normal programs are mostly sequential (CPUs would get faster and faster anyway)
- **Nowadays**
  - CPUs get faster *more slowly*
  - A program will be slow if not keeping concurrency in mind
  - The rise of Internet applications with millions of concurreny users make it even more necessary

# Concurrent Programming

## Benefits of concurrent programming

1. **Abstraction**:
   we would like to separate different tasks which can be executed independently
2. **Performance**:
   speed up a process by distributing sub-tasks to different processors/cores available on the machine
3. **Responsiveness**:
   user interface can remain responsive to the user (either taking further user input or showing a constant update of the progress to the user)

# Concurrency vs. Parallelism

In many cases, concurrency and parallel computing are used exchangeably. However, they are conceptually different.

1. **Concurrency**
    - The ability of different executing units in a program to be executed out-of-order or in partial order without affecting the final outcome
    - It mainly concerns how a program is **designed** and **structured**
    - It does not necessarily mean that things happen at the same time
2. **Parallel Computing**
    - The execution of processes or computations are carried out at the same time
    - It mainly concerns how can we execute a program **faster**
    - It usually refers to **physical** parallelism

- Ref: Yet another blog post about how parallelism is not concurrency

# Concurrent Programming

## Remember our TCP server program?

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((socket.gethostname(), 50001))
server_socket.listen(10)

while True:
    (client_socket, address) = server_socket.accept()
    data = client_socket.recv(1024)
    client_socket.sendall(data)
    client_socket.close()
```

- It can only serve **one client** at a time
- What if serving a client takes a long time?

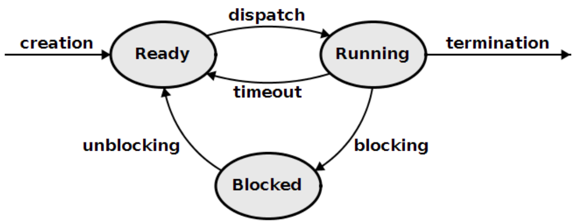
# Concurrent Programming

- Concurrency can be achieved by several means:
  - Multi-threading
  - Multi-processing
  - Asynchronous / Event driven approach
- We will focus on the first two in this lecture

# Processes and Threads

# Processes

- A **process** is an independent unit of execution (a running program in your computer)
- It has:
  - process identifier
  - memory space
- Its execution is scheduled to be carried out by the operating system on the available processor(s)



# Threads

- A **thread** is also called a **light-weighted process**
- It is created within a process to carry out various sub-tasks
- It has:
  - thread identifier
  - shared memory space in the process
  - its own memory space
- Consider a Web browser (a process)
  - Different threads are used to handle loading of pages in different tabs
  - A thread is used to allow user to type in the address bar

# Challenges in Multi-threading

## Consider the following example

- Three variables initialized: `int x = y = z = 0`
- In thread 1, we do:

```
z = x + y
```

- In thread 2, we do:

```
x = 2
y = 3
```

- If we execute two threads in parallel, what are the final values of `x`, `y`, and `z`?

# Concurrent Programming in Python

Python provides a number of APIs for concurrent programming

1. `threading`
    - A module for working with threads in a standard way
2. `multiprocessing`
    - A module with similar API as the `threading` module but spawn processes instead of threads
3. `concurrent.futures`
    - Provides an even higher-level abstraction layer for asynchronous execution of threads or processes
4. Others (e.g. `subprocesses`, `gevent`, `greenlets`, `celery`, `asyncio`)

# The Global Interpreter Lock

- **Python** is famous for its **Global Interpreter Lock (GIL)**
- At any time, only **one** native thread can be executed in one interpreter process, even when the machine has multiple processors
- NOTE: GIL is a characteristic of the CPython implementation, not of Python itself
- References:
  - Global interpreter lock - Python.org
  - Global interpreter lock - Wikipedia
- Does it mean that we cannot benefit from multi-threading in Python?

# The Global Interpreter Lock

## What does GIL mean for concurrent programming in Python

- Only the thread that has acquired the **GIL** may operator on Python objects or call Python/C API functions
- This generally means any operations that are **CPU-bound**
- A Python program may still benefit from multiple threads running concurrently if your program:
  - is I/O-bound (often has to wait for user input, network I/O, etc.) (GIL is released on I/O operations)
  - uses a lot of `numpy` array operations
  - uses some other libraries that would release the GIL when performing computation

# Threads in Python

# Threads in Python

```
from threading import Thread

def print_hello():
    """A function that a thread will execute"""
    print("Hello!")

# Create 5 threads
threads = []
for i in range(5):
    t = Thread(target=print_hello)
    t.start()
    threads.append(t)

# Wait until all threads are finished
for t in threads:
    t.join()
```

Executing the script:

```
Hello!
Hello!
Hello!
Hello!
Hello!
```

- `.start()` will start the thread's activity (only call this at most **ONCE**)
- `.join()` will be blocked until that thread has terminated.

# Threads in Python

- You can pass arguments to the target function

```
from threading import Thread

def print_hello(n):
    print("Hello from Thread %d!" % n)

threads = []
for i in range(5):
    t = Thread(target=print_hello, args=(i,))
    t.start()
    threads.append(t)

...
```

- Executing the script:

```
Hello! 0
Hello! 1
Hello! 2
Hello! 3
Hello! 4
```

- Note that arguments must be in the order as defined in the function signature

# Daemonic and Non-daemonic Threads

- Threads can either be **non-daemonic** (default) or **daemonic** (means running in the background)

```
from threading import Thread
from time import sleep

def print_hello():
    sleep(0.2)
    print("Hello!")

threads = []
for i in range(5):
    t = Thread(target=print_hello, daemon=True)
    t.start()

# Nothing will be printed when program exits
```

- A program will wait until all **non-daemonic threads** have completed and terminated (even without explicitly calling `join()`)
- If you want to avoid some threads to from blocking the main program from exiting, use *daemonic* threads

# Notes on Threads in Python

- Once the `start()` function is called on a thread object, the target function will be executed in a new thread
- Your program continues to execute other lines in the **main thread**
- Call `join()` when you want to make sure that all threads have terminated
- All threads (either daemonic or non-daemonic) will be terminated if the main program stops
- It is **NOT guaranteed** that threads are executed in the order of calling the `start()` function

# Getting a list of all active threads

```
import time
import threading
from threading import Thread

def print_hello(n):
    time.sleep(0.5)
    print("Hello from Thread %d!" % n)

for i in range(5):
    t = Thread(target=print_hello, args=(i,))
    t.start()

main_thread = threading.main_thread()
for t in threading.enumerate():
    if t != main_thread:
        t.join()
```

- `threading.main_thread()` returns the main thread in the process
- `threading.enumerate()` returns all threads that are currently active, including threads that have not yet started
- You CANNOT join the main thread (raise `RuntimeError`)

# More examples

- Print squares of all integers up to 100

```
import time
import threading
from threading import Thread

def print_squares(n):
    print(n*n)

start_time = time.time()
for i in range(100):
    Thread(target=print_squares, args=(i+1,)).start()

for t in threading.enumerate():
    if t != threading.main_thread():
        t.join()

print("Using threads:", time.time() - start_time)  # print the time used
```

# More examples

- Do we have any **speed up**?
- See an example run:

```
...
9604
9801
10000
multi-threading: 0.00979924201965332
...
9604
9801
10000
Sequential: 0.001138925552368164
```

- Using multi-threading is even **slower** in this case

# Another Example: Image Downloading

```
import threading
import time
from urllib import request

images = [...]  # A list of URLs to images

def download(url, i):
    print("Downloading image %d..." % i)
    request.urlretrieve(url, "%d.jpg" % i)

start_time = time.time()
for i, url in enumerate(images):
    threading.Thread(target=download, args=(url, i)).start()

for t in threading.enumerate():
    if t != threading.main_thread():
        t.join()

print(time.time() - start_time)
```

# More examples

- Do we have any speed up?
- See an example run:

```
Downloading image 0...
Downloading image 1...
Downloading image 2...
Downloading image 3...
Downloading image 4...
Downloading image 5...
multi-threading: 1.6151726245880127
...
Downloading image 0...
Downloading image 1...
Downloading image 2...
Downloading image 3...
Downloading image 4...
Downloading image 5...
Sequential: 4.215069532394409
```

# When to Use Multi-theading?

- If your program is **CPU-bound**, using multi-threading will actually be **slower**
  - The GIL is almost never released by a thread
  - Need time to create a new thread (overhead)

- If your program is **I/O-bound**, you may consider using multi-threading
  - A thread will release the GIL when it is waiting for I/O (e.g. image being downloaded from the network)
  - The longer your I/O operation is, the more you benefit from multi-threading

- You should also consider multi-threading if you don't want a particular task to **block** your program's flow

# Subclassing Thread

- When you have more complex operations in a thread

```
import threading

class SquareThread(threading.Thread):

    def __init__(self, n):
        self.n = n

    def run(self):
        print(n * n)

for i in range(5):
    t = SquareThread(i)
    t.start()
```

- When subclassing `Thread`, your class should always have a `run()` method
- Operations in the `run()` method will be executed once the `start()` method is invoked

# Shared Object between Threads

- Sometimes it is necessary to **share some resources** among all threads
- Examples:
    - A global counter which all threads will update
    - A dictionary tracking the number of times different values have appeared
- The `threading` module provides the `Lock` class, which can be used to make sure that only one thread is accessing an object at a time
- Consider the example of a *global counter*

# Using Lock

```
class Counter:
    def __init__(self):
        self.lock = threading.Lock()
        self.count = 0
    def increment(self):
        self.lock.acquire()
        try:
            self.count += 1
        finally:
            self.lock.release()

def add_two(counter):
    for i in range(2):
        counter.increment()

counter = Counter()
for i in range(5):
    Thread(target=add_two, args=(counter,)).start()
```

- We create a custom class called `Counter`
- Whenever `increment` is invoked, we first try to `acquire` the lock
- `acquire` will be blocked if another thread has acquired the lock before, until that thread `release` the lock

# Using Lock

- The previous example can be simplified using the `with` statement

```
class Counter:
    def __init__(self):
        self.lock = threading.Lock()
        self.count = 0
    def increment(self):
        with self.lock:
            self.count += 1

def add_two(counter):
    for i in range(2):
        counter.increment()

counter = Counter()
for i in range(5):
    Thread(target=add_two, args=(counter,)).start()

...
```

# Getting output from threads

- Using a `Queue`, you can receive data from other threads (Ref: [A Synchronized Queue Class])

```
from threading import Thread
from queue import Queue

def square(n, queue):
    queue.put(n * n)

# Pass a queue as an argument to each thread
output_queue = Queue()
for i in range(10):
    Thread(target=square, args=(i, output_queue)).start()

... # join the threads

# Retrieve the results from the queue
while not output_queue.empty():
    print(output_queue.get())
```

# Multiprocessing in Python

# Multiprocessing in Python

- `multiprocessing` is a module in Python that provides APIs similar to that of `threading` for spawning processes
- Each process would be executed in a **separate** Python interpreter, thus will NOT be restricted by the global interpreter lock (GIL)
- This allows you to fully leverage multiple processors on a machine
- For parallel execution of functions, `multiprocessing` is preferred

# Multiprocessing

```
# Import the Process class
from multiprocessing import Process

# A function to print Hello World
def f(i):
    print("Hello World %d!" % i)

if __name__ == '__main__':
    processes = []
    for i in range(10):
        p = Process(target=f, args=(i, ))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()
```

# Multiprocessing

```python
# Import the Process class
from multiprocessing import Process

# A function to print Hello World
def f(i):
    print("Hello World %d!" % i)

if __name__ == '__main__':
    processes = []
    for i in range(10):
        p = Process(target=f, args=(i, ))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()
```

- As you can see, the API is the same as using `Thread`
- Each spawned process will have its own Python interpreter
- Note that the `if __name__ == '__main__':` line is necessary (protecting the entry point of the main program)
- You must `join` all processes, otherwise they will become *zombies*

# Multiprocessing

## What is the line `if __name__ == '__main__':` necessary?

- When spawning new processes, the module will be **imported** by these processes, such that they can access the **target function**
- This is similar to the case when you use want to use a function defined in another Python file
- When importing a module, the codes inside the module will be executed, UNLESS it is protected by the name checking line
- When a module is imported, its `__name__` will NOT be `__main__`

# Multiprocessing

- Using **Pool** in multiprocessing to process lines of text

```
from multiprocessing import Pool

def count_words(line):
    counts = {}
    for w in line.lower().split(" "):
        counts[w] = counts.get(w, 0) + 1
    return counts

with Pool(4) as pool:  # Use with will make sure that all processes are closed and joined
    all_counts = pool.map(count_words, lines)

word2count = {}  # collect the counts from each output
for c in all_counts:
    for w, n in c.items():
        word2count[w] = word2count.get(w, 0) + n
```

# Using Pool

```
...

with Pool(4) as pool:
    all_counts = pool.map(count_words, lines)

...
```

- **`Pool(4)`** creates a pool of 4 processes
- If you don't specify the number, it will be equal to the number of CPUs on your machine (which can also be obtained using **`os.cpu_count()`**)
- **`lines`** is a list (an iterable), **`map`** will split it into chunks and pass them to each processes

# Multiprocessing Pool

## What if your target function receives more than one argument?

- use `starmap()` of the pool object

```
from multiprocessing import Pool

def multiply(x, y):
    return x * y

args = [(1, 2), (3, 4), (5, 6)]
with Pool() as pool:
    results = pool.starmap(multiply, args)

print(results)
# prints [2, 12, 30]
```

# Summary

- For CPU-bound tasks, use `multiprocessing` instead of `threading`, so that you can get around the **GIL**
- `threading` is more for *concurrent programming*, while `multiprocessing` is more for *parallel computing*
- You should always test whether you can achieve any speed gain, especially for `multiprocessing` because it introduces more overhead
- Be careful when sharing objects among threads and processes, use thread-safe objects such as `Queue` if possible

# Logging in Python

# Logging

When writing your programs, it is important that you display some messages (logs) for various purposes

- Check for runtime exceptions (when you have `try...except` blocks)
- Understand how the application is performing (how much time is taken to run certain lines)
- Facilitate debugging and future development (e.g. what is the value of `x` at some point?)

# Logging

## Why not use `print`?

```
try:
    infile = open(“data.txt”, “r”)
    lines = infile.readlines()
    infile.close()
    print("Read file successfully")
except Exception as ex:
    print(ex)
    print("Error in reading file")
```

- Cannot control when to or when not to print
- Cannot easily control where to print
- Cannot change the message format easily
- Cannot print to multiple destinations (e.g. both the standard output and a log file)

# The `logging` Module

- Python provides a powerful `logging` module in its standard library
- Provides a flexible event logging system
- Can choose to send log to standard output or files
- Allows customisation of the log format
- Features different log levels

# The `logging` Module

- An example

```
import logging
logging.basicConfig(level=logging.DEBUG)

logging.debug("This is a debug message")
logging.info("This is an information message")
logging.error("An error has occurred!"")

# The above will output the following on the screen
# DEBUG:root:This is a debug message
# INFO:root:This is an informational message
# ERROR: An error has occurred!
```

# Log Levels

The following levels are defined in Python

- **Debug**
  - For debugging purpose, should not be shown in a production system
- **Information**
  - For informational messages
- **Warning**
  - For giving warnings when something unexpected happens (but does not stop the flow of the program)
- **Error**
  - For error messages
- **Critical**
  - Something that needs to be attended immediately

# Concepts in `logging`

## Logger

- An object that can be used to create log messages
- Each logger is identified by a unique name

## Handler

- Defines what to do when a log message is created in the logger
- Each handler can have its own log format
- Common handlers: file, standard error, standard output

# Examples

```python
import logging

# Create a logger with the program name and level=DEBUG
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Create a file handler that will write to a file
fh = logging.FileHandler("log.txt")

# Create a handler that will write log to stderr
ch = logging.StreamHandler()

# Add the handlers to the logger
logger.addHandler(ch)
logger.addHandler(fh)

logger.info("Hello!")
```

# Formatting

- You can create a **`formatter`** for each handler to format their log messages

```
# Create a formatter and add it to the handlers
formatter = logging.Formatter("TIME=%(asctime)s, %(name)s, [%(levelname)s] : %(message)s")
ch.setFormatter(formatter)
fh.setFormatter(formatter)
```

- The above will print:

  **`TIME=2018-01-24 07:42:54,232, __main__, [INFO] : Hello!`**

- References: https://docs.python.org/3/library/logging.html#logrecord-attributes

# References

For more examples and tutorials, check the official documents at:

- The logging module (https://docs.python.org/3/library/logging.html)
- Logging HOWTO (https://docs.python.org/3/howto/logging.html)
- Logging Cookbook (https://docs.python.org/3/howto/logging-cookbook.html)

# Assignment 1

# Assignment 1

- Instructions of Assignment 1 available at http://iems5703.albertauyeung.com/assignment-1
- **Deadline**: 10th February, 2018
- Develop a TCP socket server and a TCP socket client
- The server provides POS tagging service
- You have to install the `nltk` library in order to finish the assignment
- Read the instructions *carefully* and make sure that your output is the same as in the examples
- Ask questions on Slack!

End of Lecture 3