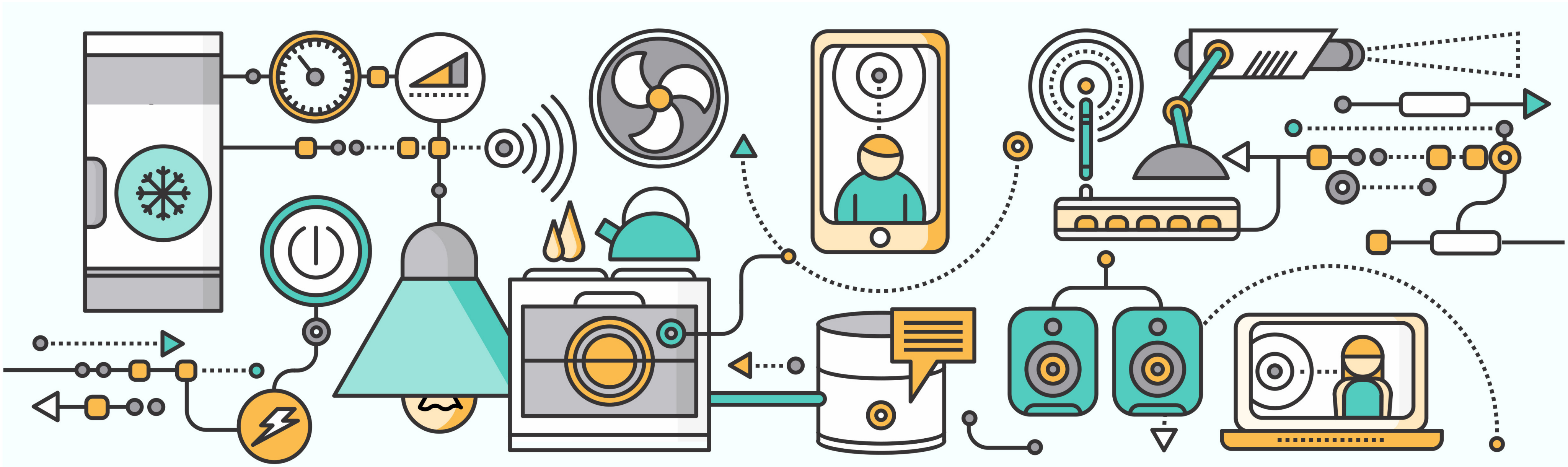
ENME 441
Mechatronics and the Internet of Things



Concurrency: Threading & Multiprocessing

Consider original *servo.py* code:

```
import RPi.GPIO as gpio
import time

gpio.setmode(gpio.BCM)
pwmPin = 24
gpio.setup(pwmPin, gpio.OUT)

dcMin = 3
dcMax = 12

pwm = gpio.PWM(pwmPin, 50)
pwm.start(0)

while(1):
  for dc in range(dcMin, dcMax):
    pwm.ChangeDutyCycle(dc)
    time.sleep(0.5)
```

There is a problem with *servo.py*

- In *servo.py* a loop continually changes the motor angle – what if we wanted something else to happen at the same time?

- For example, say we want to have 2 motors looping, but with slightly different delays between each step – this would be VERY cumbersome to implement in the current code.

- We need some form of concurrency to handle this type of situation: threading or multiprocessing

Threading

https://docs.python.org/3/library/threading.html

- A *thread* is a process that runs independent of other threads, including the main thread.
- Threads can share information and access the same variables (with a bit of work).
- Threading allows multiple independent processes to run simultaneously.
  - Very useful for mechatronics & IoT objects, e.g. allowing precise timing of outputs to different GPIO pins regardless of what else is happening in our code
  - Similar in concept to threaded callbacks, but with code-based control over new threads (rather than through a GPIO trigger).

_thread vs. threading

https://docs.python.org/3/library/_thread.html

- The `_thread` module provides low-level control over threads.
- The `threading` module is built on top of `_thread` and provides an easier-to-use and higher-level threading API.
- Note that some Python implementations (e.g. Micropython) do not support the `threading` module.

Threading

```
import threading
```

Import the threading module

```
def myFunction():
  print("Thread started")
  for i in [3,2,1]: print(i)
  print("Thread ended")
```

Create a function

```
t = threading.Thread(target=myFunction)
```

Create a Thread object that targets the desired function

```
t.start()          # Start the thread (only once!)
```

Start the thread (call the Thread start method) to execute the function concurrently with the main thread

Passing Arguments

Pass data
(optional)

```
t = threading.Thread(target=myFn, args=(countdown,))
```

- A single <u>iterable</u> argument can be passed to the threaded function
- Canonically the argument is a tuple, so if only a single value is in the tuple you must place a comma after the value (otherwise Python doesn't know you are defining a tuple!)
- Alternately the values can be packaged in a list or other iterable

Thread syntax and methods

```
import threading
import time

def countdown(count):
  print("Thread started")
  for i in count:
    print(i)
    time.sleep(0.5)
  print("Thread ended")

count = [3,2,1]

t = threading.Thread(name='myname',target=countdown,args=(count,))

t.daemon = True   # Daemon threads are forced to end when the
                  # main code terminates

t.start()         # Start the thread (only once!)

t.join()          # Force the calling process to wait for the thread to
                  # end before continuing

t.join(n)         # Set an upper limit to the waiting time
```

Name the thread (optional)

Function to run In new thread

Pass data (optional)

Threading Example

```
import time
import threading

def fn():
  while True:
    print('fn')
    time.sleep(0.5)

try:
  myThread = threading.Thread(target=fn)
  myThread.start()

  while True:
    print('main')
    time.sleep(0.75)

except:
  pass

myThread.join()
```

Thread Subclassing

- We can also make our custom classes inherit from the Thread class, allowing class objects to operate in a separate thread.

- Inherit from `threading.Thread` and override the `__init__()` and `run()` methods:

```
import threading

class newThreadedClass(threading.Thread):
  def __init__(self):
    threading.Thread.__init__(self)
  def run(self):
    # put thread code here


t = newThreadedClass()
```

Thread Subclassing Example

```
import threading
import time

class Countdown(threading.Thread):
  def __init__(self, count, thread_name):
    threading.Thread.__init__(self, name=thread_name)
    self.count = count
  def run(self):
    print("Thread started")
    for i in self.count:
      print(i)
      time.sleep(0.5)
    print("Thread ended")

for i in range(3):
  t = Countdown([3,2,1], "name="+str(i))
  t.start()
  print('t.is_alive() =', t.is_alive())
  print(t.getName())
  t.join()
  print('t.is_alive() =', t.is_alive())
```

Pass the thread name (optional)

Pass the thread name

True if thread is running

Returns name of thread

Threading notes

- Be careful of spawning rogue threads
  - processor load issues
  - memory leaks

- Killing a thread takes some extra work (not covered here) – use the daemon flag to ensure threads are killed when the main Python code ends

- The *global interpreter lock* (GIL) limits the utility of multiple simultaneous threads...to overcome this, use <u>multiprocessing</u> instead

Multiprocessing

https://docs.python.org/3/library/multiprocessing.html

- Similar to threading in concept and syntax:

```
import threading
class Classname(threading.Thread):

import multiprocessing
class Classname(multiprocessing.Process):
```

- Bypasses the global interpreter lock to allow Python to execute multiple simultaneous bytecodes (pyc files)
- Each new process is spawned in a new Python instance, each with a unique memory space
- Benefits:
  - higher performance (with caveats)
  - easy control over process termination (unlike threads)
  - total number of processes limited only by resources (unlike threads)

Multiprocessing

```
import multiprocessing
import time

def Countdown(count):
  print("Process started")
  for i in count:
    print(i)
    time.sleep(0.5)
  print("Process ended")

if __name__ == '__main__':    # Required!
  x = [3,2,1]
  p = multiprocessing.Process(name='myname',target=Countdown,args=(x,))


  p.daemon = True   # Force process termination when main code ends
  p.start()         # Start the process (only once!)
  p.terminate()     # Terminate the process (no equivalent for threads)
                    #   (always 'join' after termination)
  p.join()          # Force the calling process to wait for the new
                    #   process to end before continuing
  p.join(n)         # Pause the calling process for up to n seconds,
                    #   then join even if not ended
```

Multiprocessing Example

```
import time
import multiprocessing

def fn():
  while True:
    print('fn')
    time.sleep(0.5)

if __name__ == '__main__':
  try:
    myProcess = multiprocessing.Process(target=fn)
    myProcess.start()

    while True:
      print('main')
      time.sleep(0.75)

  except:
    pass

  myProcess.terminate()
  myProcess.join()
```

Process Subclassing Example

```
import multiprocessing
import time

class Countdown(multiprocessing.Process):
  def __init__(self, count, process_name):
    multiprocessing.Process.__init__(self, name=process_name)
    self.count = count
  def run(self):
    print("Process started")
    for i in self.count:
      time.sleep(0.5)
      print(i)
    print("Process ended")

if __name__ == '__main__':     # Required!
  for i in range(3):
    p = Countdown([3,2,1], "name="+str(i))
    p.start()
    print('p.is_alive() =', p.is_alive())
    print(p.name)
    p.join()
    print('p.is_alive() =', p.is_alive())
```

Passing data between main & secondary processes using shared memory: multiprocessing.Array and multiprocessing.Value

```
import multiprocessing, time

myValue = multiprocessing.Value('i')
myArray = multiprocessing.Array('f',3)

def fn(myArray, myValue):
  for (idx,n) in enumerate([3,2,1]):
    myArray[idx] = n**2
    myValue.value = int(sum(myArray))
    print("In the process, iter={}:".format(idx))
    print(" Array: {}".format(myArray[:]))
    print(" Value: {}".format(myValue.value))
if __name__ == '__main__':    # Required!
  p1 = multiprocessing.Process(target=fn, args=(myArray, myValue))

  print("Before starting process:")
  print(" Array: {}".format(myArray[:]))
  print(" Value: {}".format(myValue.value))

  p1.start()
  print("\n\nImmediately after starting process:")
  print(" Array: {}".format(myArray[:]))
  print(" Value: {}\n\n".format(myValue.value))

  p1.join()
  print("\n\nAfter completing process:")
  print(" Array: {}".format(myArray[:]))
  print(" Value: {}".format(myValue.value))
```

Type of data to be stored in Array or Value

# of elements in Array (can also pass initial array values)

Names of Array and Value in shared memory space

value is in a wrapper, and must be extracted using *.value

Multiprocessing vs. Threading

- Processes use separate memory spaces, while threads run in the same memory space.
  - the GIL is needed to prevent threads from writing to the same memory at the same time.
  - Harder to share variables between processes than between threads.
  - Global variables can be directly modified in new threads, but not in new processes!
- Threads can be spawned within processes, but not visa versa.
- Spawning processes is slower than threads (more overhead to launch a new Python instance and allocate memory).
- Process scheduling handled by the OS, while thread scheduling handled by the threading library.
  - Processes have independent I/O scheduling, while threads share I/O scheduling (can be a bottleneck).
  - Multiprocessing can take advantage of multiple CPUs & cores.

When to use Threading?

- When initialization speed is important
- When concurrency across multile threads is not required
- Tasks requiring easy transfer of information & communication between threads
- Tasks that need to be run after a specific delay using threading.Timer

When to use Multiprocessing?

- Everything else