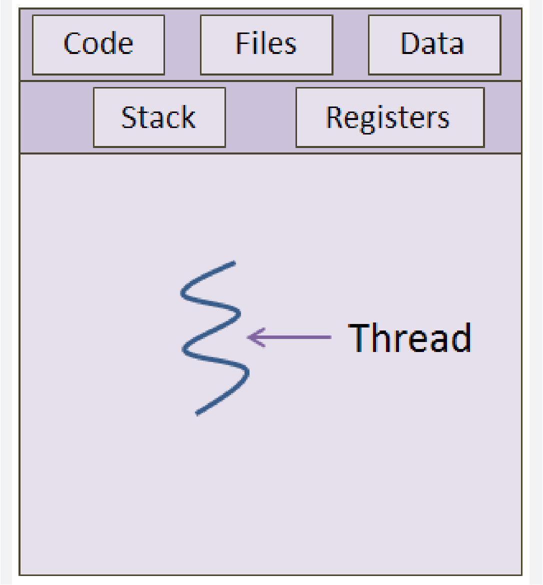
Embedded System Software

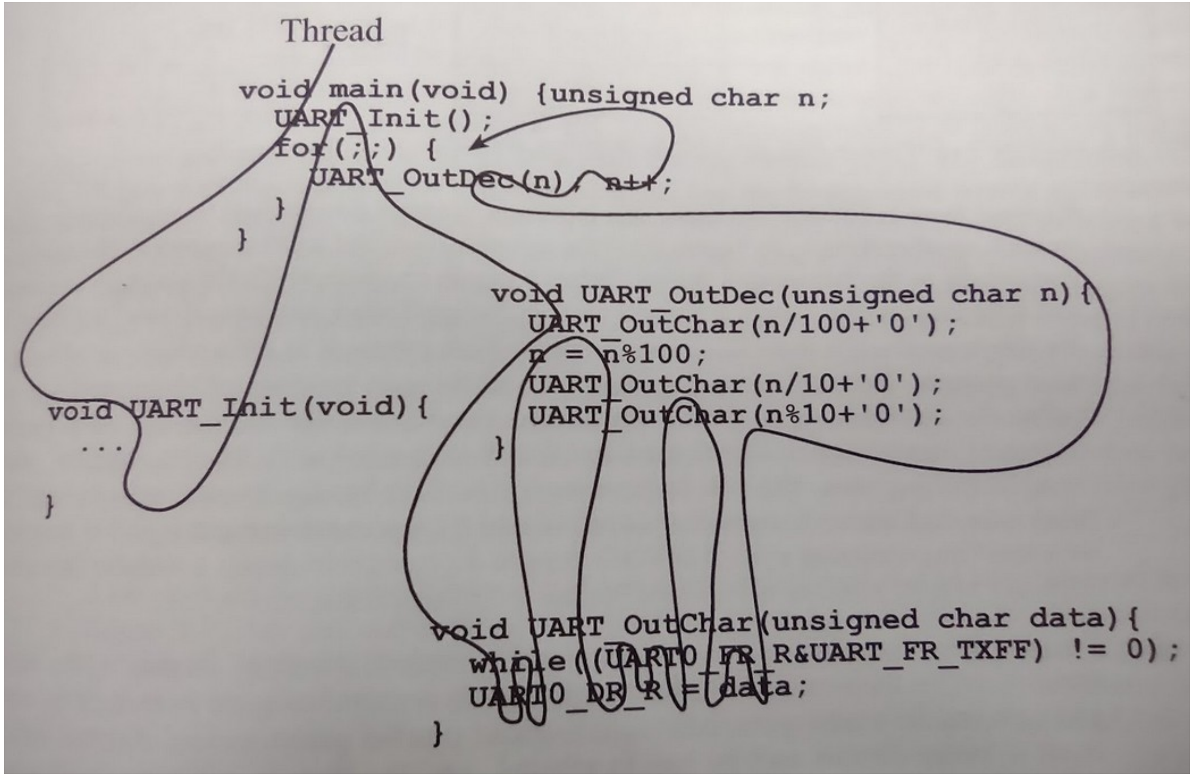
Lecture 13 : Thread in Micropython

Definition of Thread

Thread is defined as the **path of action** of software as it executes.



Definition of Thread



```
void main(void) {unsigned char n;
  UART_Init();
  for(;;) {
    UART_OutDec(n); n++;
  }
}
```

```
void UART_Init(void){
  ...
}
```

```
void UART_OutDec(unsigned char n){
  UART_OutChar(n/100+'0');
  n = n%100;
  UART_OutChar(n/10+'0');
  UART_OutChar(n%10+'0');
}
```

```
void UART_OutChar(unsigned char data){
  while((UART0_FR_R&UART_FR_TXFF) != 0);
  UART0_DR_R = data;
}
```

Thread in Embedded

- **Main program** is called **foreground thread**. In embedded applications, It executes a never-end loop
- Foreground thread can be broken by **Interrupt**. Which create **multi-threads**

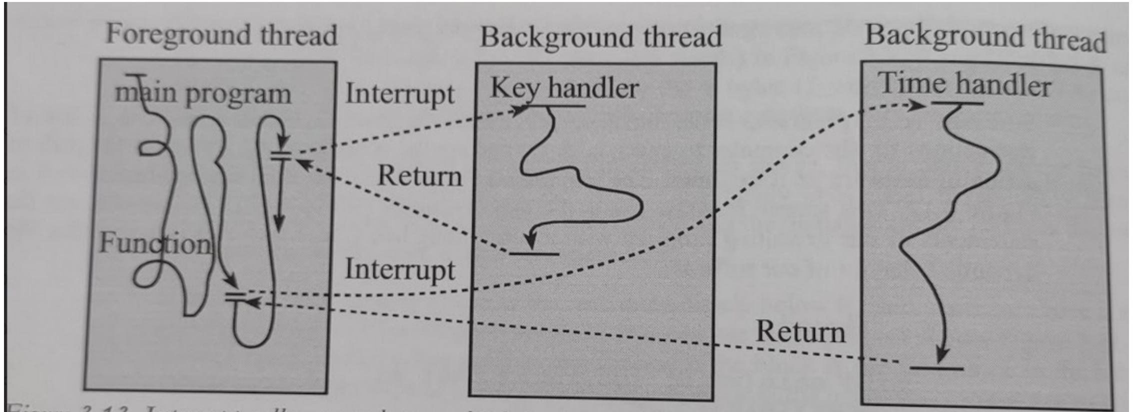
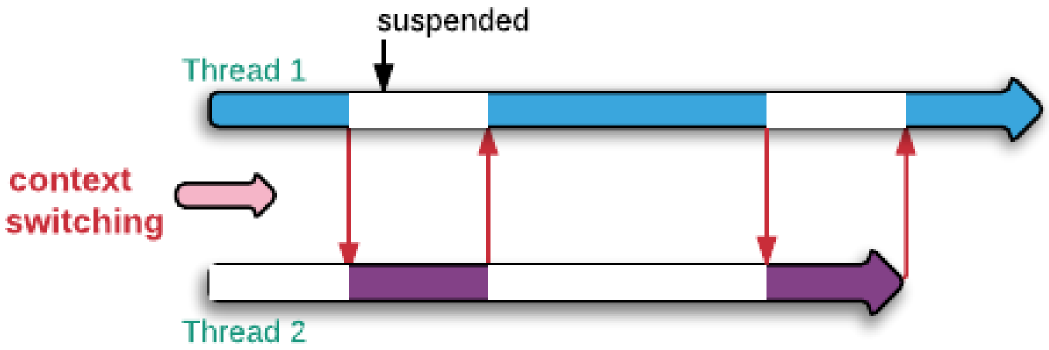


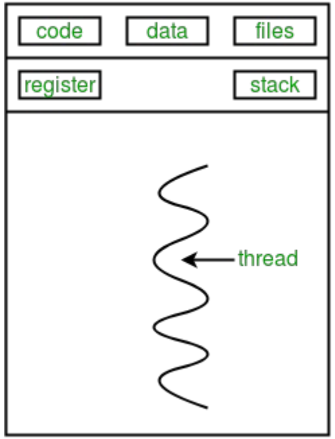
Figure 3.13. Interrupts allow us to have multiple background threads.

MultiThread

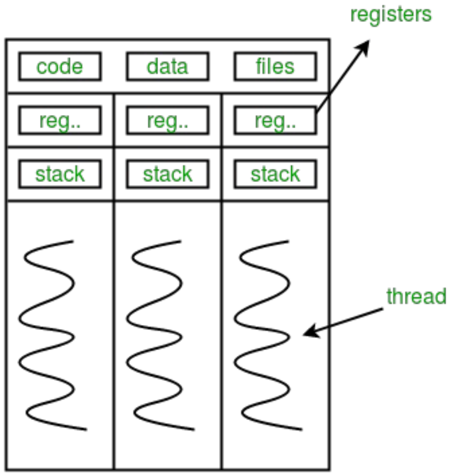
- Concurrency is achieved using frequent switching between threads **-> Multi-thread**
- OS Scheduler (RTOS for example) will handle the switching process.



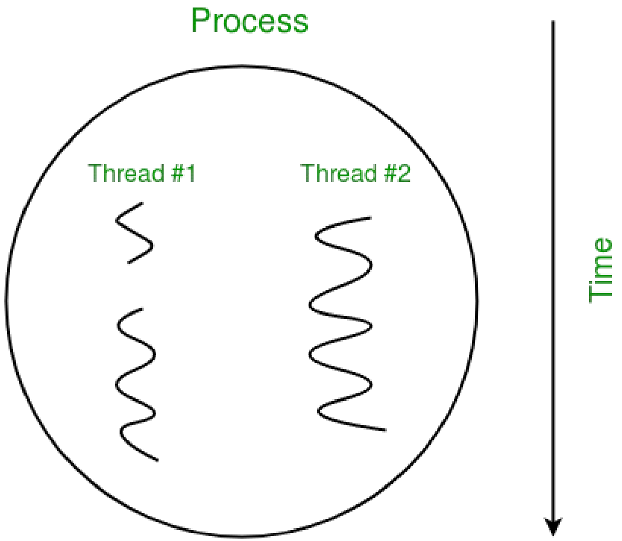
MultiThread



single-threaded process



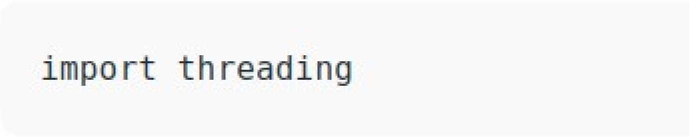
multithreaded process



MultiThread in Python

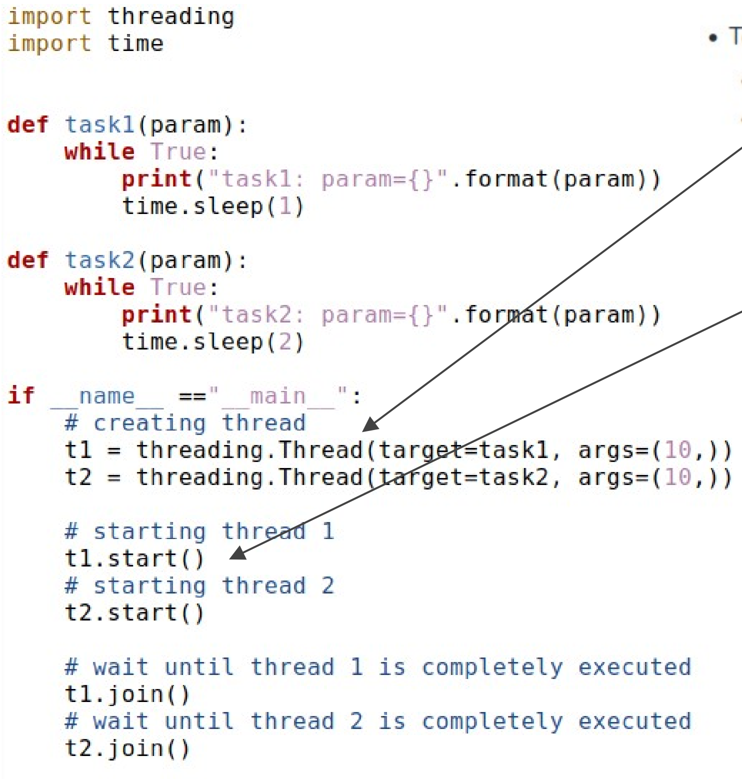
- Using threading module.

  The **threading** module provides a very simple and intuitive API for spawning multiple threads in a program.



```
import threading
```

MultiThread in Python



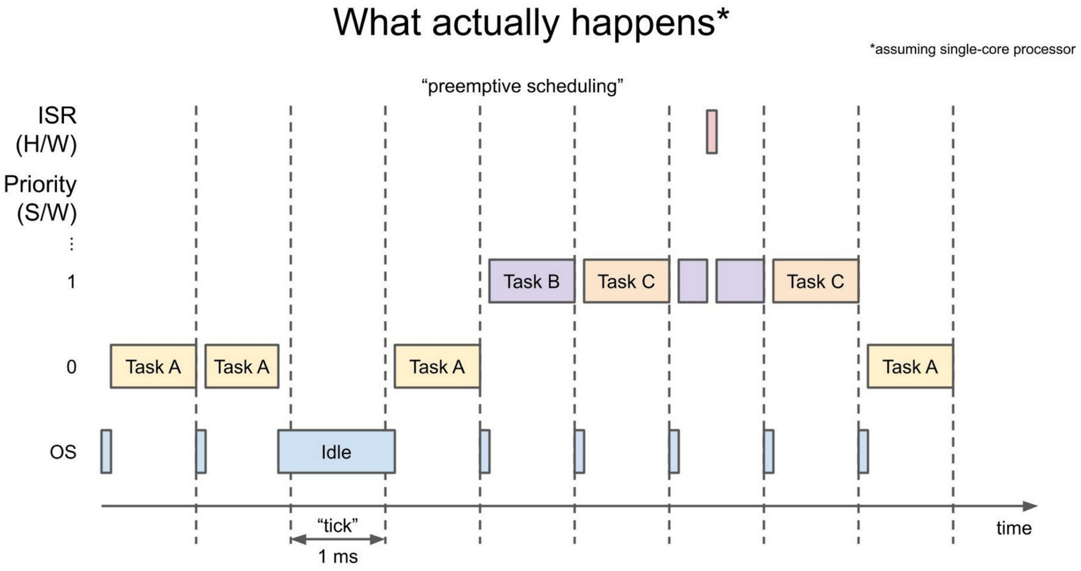
```
import threading
import time


def task1(param):
    while True:
        print("task1: param={}".format(param))
        time.sleep(1)

def task2(param):
    while True:
        print("task2: param={}".format(param))
        time.sleep(2)

if __name__ =="__main__":
    # creating thread
    t1 = threading.Thread(target=task1, args=(10,))
    t2 = threading.Thread(target=task2, args=(10,))

    # starting thread 1
    t1.start()
    # starting thread 2
    t2.start()

    # wait until thread 1 is completely executed
    t1.join()
    # wait until thread 2 is completely executed
    t2.join()
```

- To create a new thread, we create an object of **Thread** class. It takes following arguments:
  - **target**: the function to be executed by thread
  - **args**: the arguments to be passed to the target function

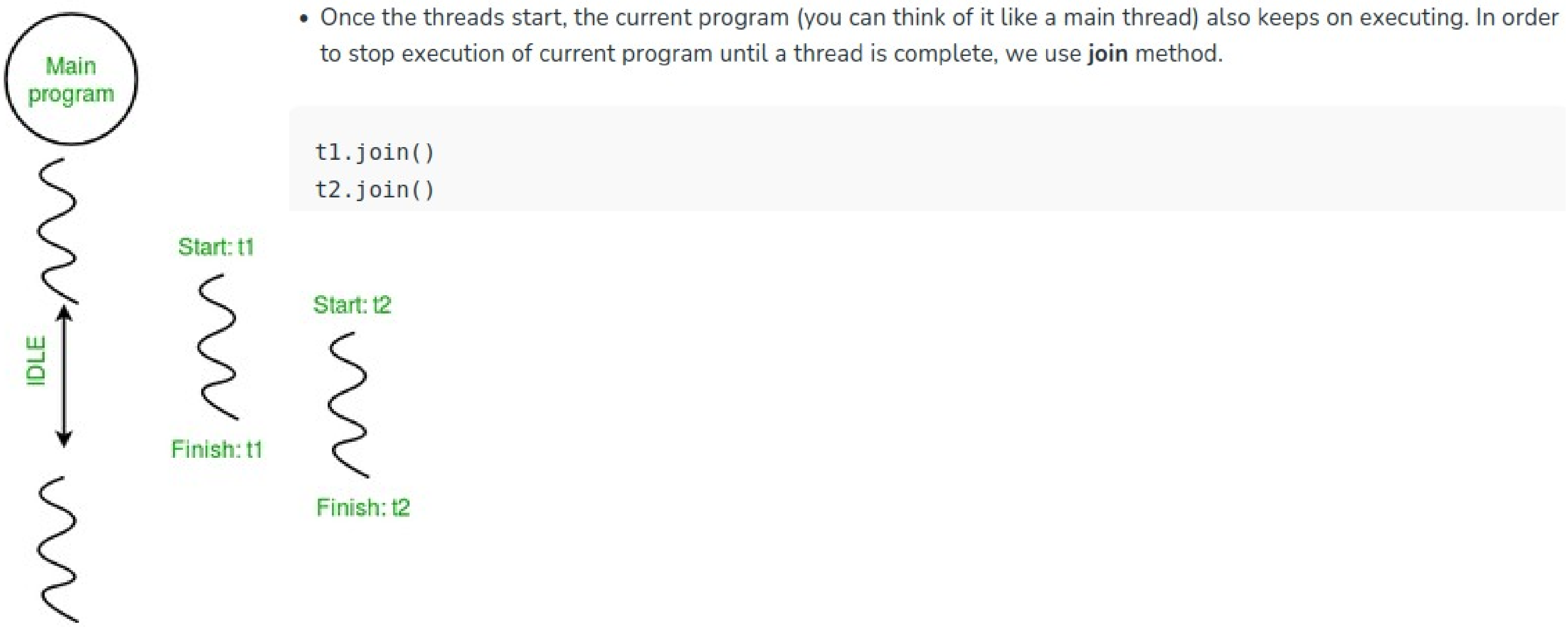
- To start a thread, we use **start** method of **Thread** class.

MultiThread in Python

- Multi-Thread in python is **Preemptive Multitasking**
- Similar to the RTOS



MultiThread in Python

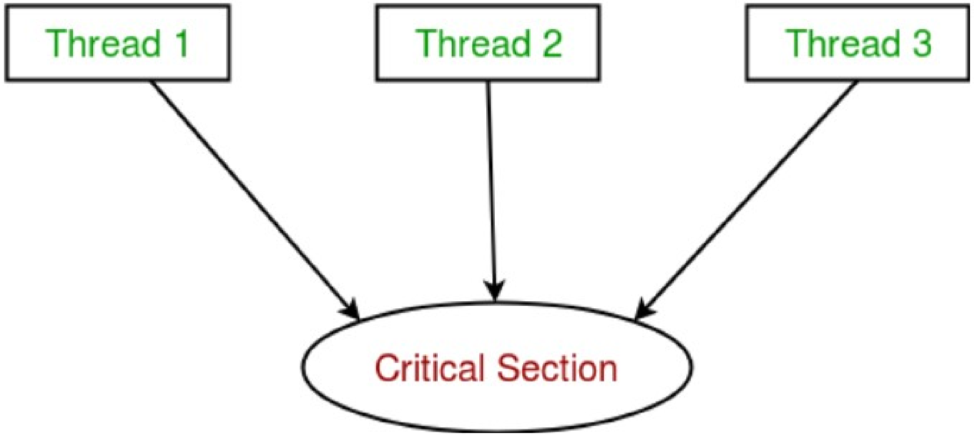


- Once the threads start, the current program (you can think of it like a main thread) also keeps on executing. In order to stop execution of current program until a thread is complete, we use **join** method.

```
t1.join()
t2.join()
```

MultiThread in Python

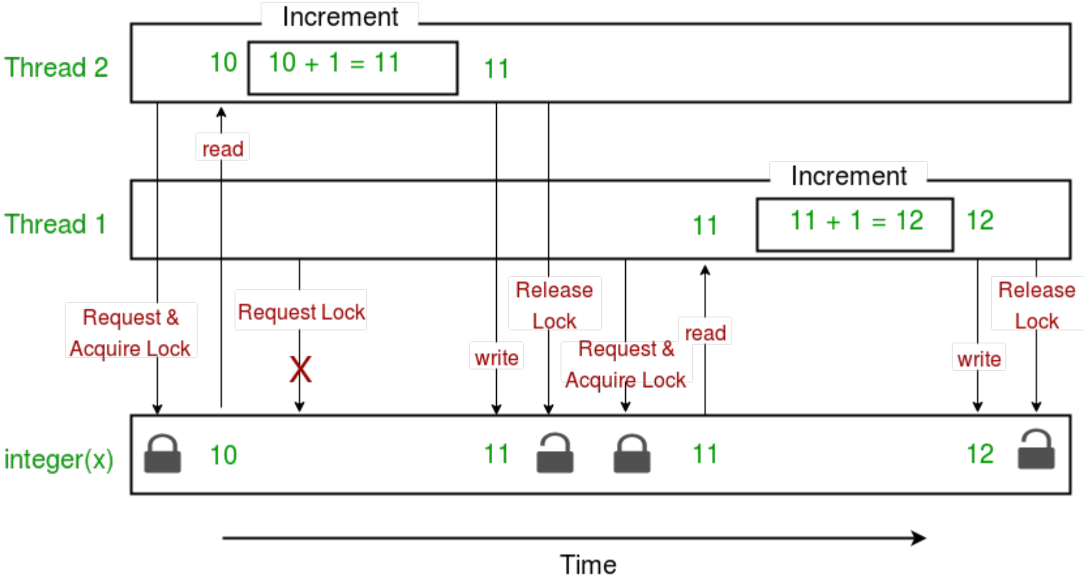
- Shared Resource?



Concurrent accesses to shared resource can lead to **race condition**.

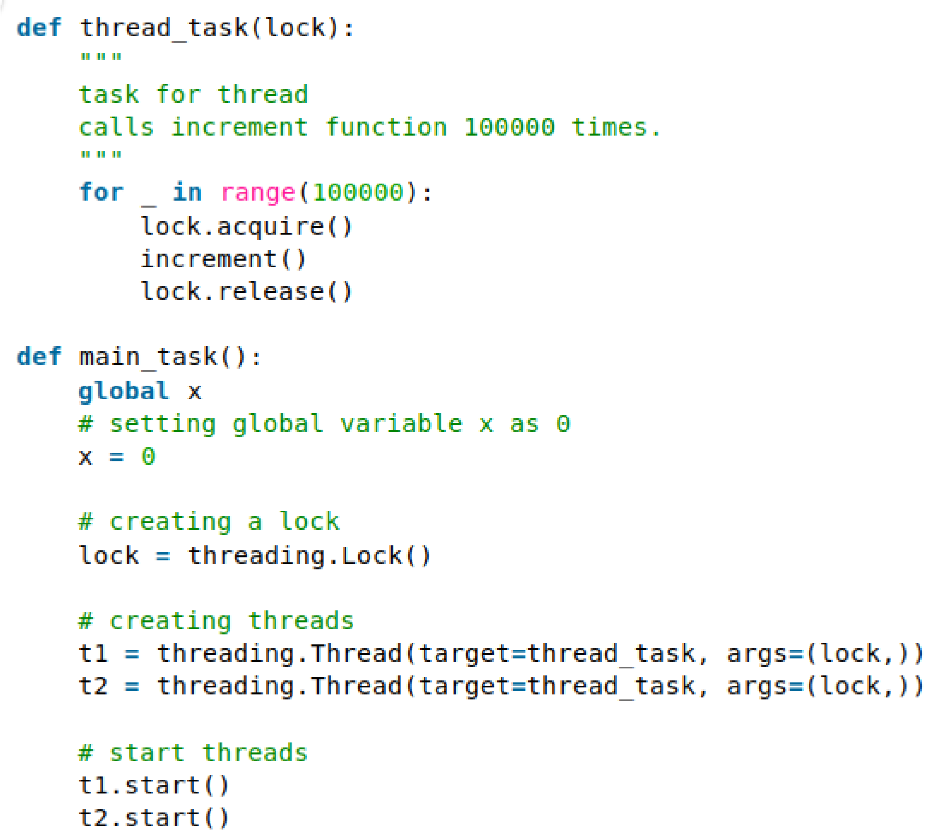
MultiThread in Python

- MUTEX/Binary Semaphore is needed
- Use threading.Lock() for that.



MultiThread in Python

Example

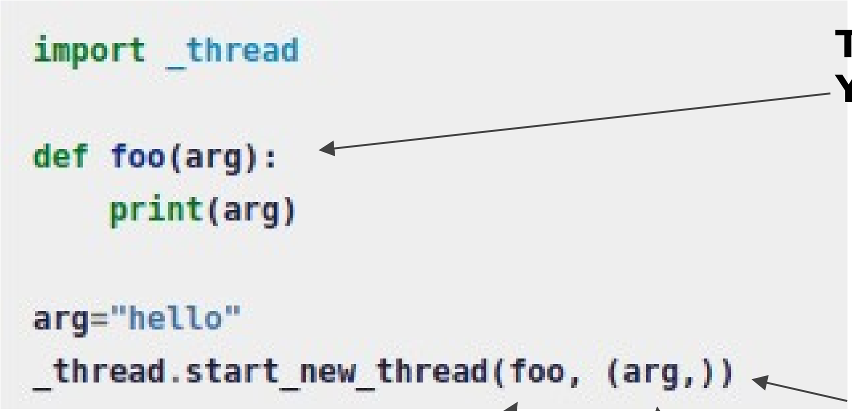


```python
def thread_task(lock):
    """
    task for thread
    calls increment function 100000 times.
    """
    for _ in range(100000):
        lock.acquire()
        increment()
        lock.release()

def main_task():
    global x
    # setting global variable x as 0
    x = 0

    # creating a lock
    lock = threading.Lock()

    # creating threads
    t1 = threading.Thread(target=thread_task, args=(lock,))
    t2 = threading.Thread(target=thread_task, args=(lock,))

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

MultiThread in MicroPython?

- Depending on the ports of micropython (STM32, ESP32, RP2 etc.)
- **Threading** module is not available. Use **_thread** module instead.

MultiThread in MicroPython



```
import _thread

def foo(arg):
    print(arg)

arg="hello"
_thread.start_new_thread(foo, (arg,))
```

Thread function
You can put your task(s) in here

Start the thread

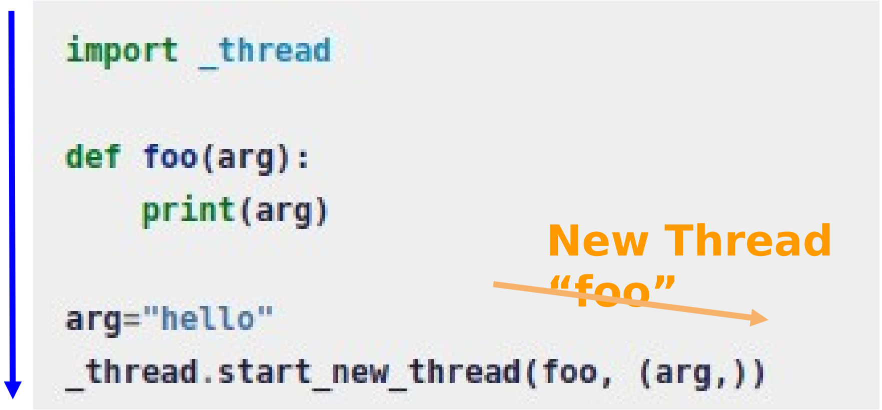
Thread function

Argument of Function (Tuple)

MultiThread in MicroPython

Q: How many thread in this?

A: 2 Threads



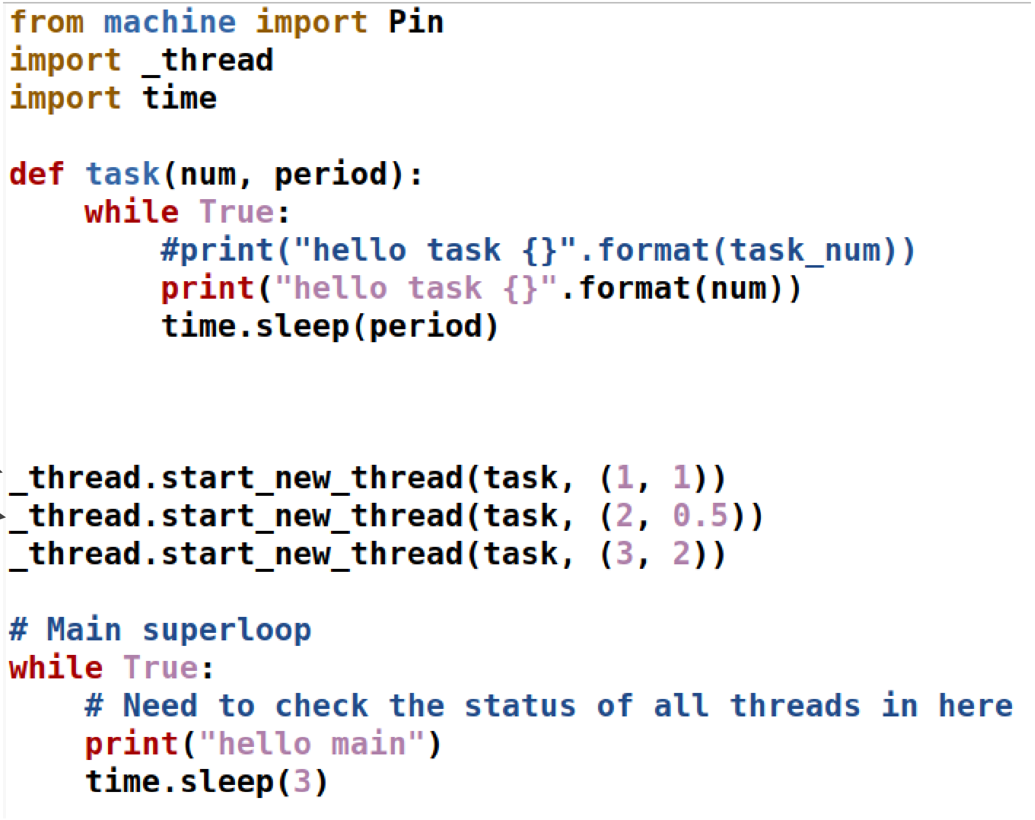
```
import _thread

def foo(arg):
    print(arg)

arg="hello"
_thread.start_new_thread(foo, (arg,))
```

New Thread “foo”

Main thread

MultiThread in MicroPython

Example



```
from machine import Pin
import _thread
import time

def task(num, period):
    while True:
        #print("hello task {}".format(task_num))
        print("hello task {}".format(num))
        time.sleep(period)



_thread.start_new_thread(task, (1, 1))
_thread.start_new_thread(task, (2, 0.5))
_thread.start_new_thread(task, (3, 2))

# Main superloop
while True:
    # Need to check the status of all threads in here
    print("hello main")
    time.sleep(3)
```

Thread 1

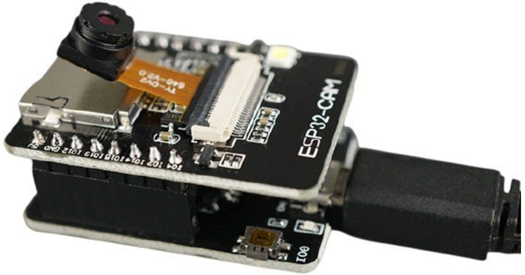
Thread 2

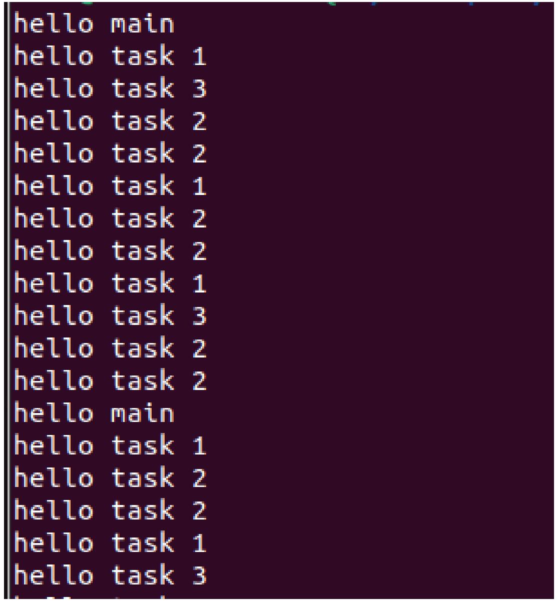
Thread 3

Thread 0 (Main)

MultiThread in MicroPython

Demo on ESP32

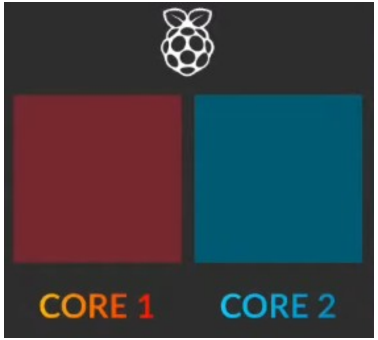




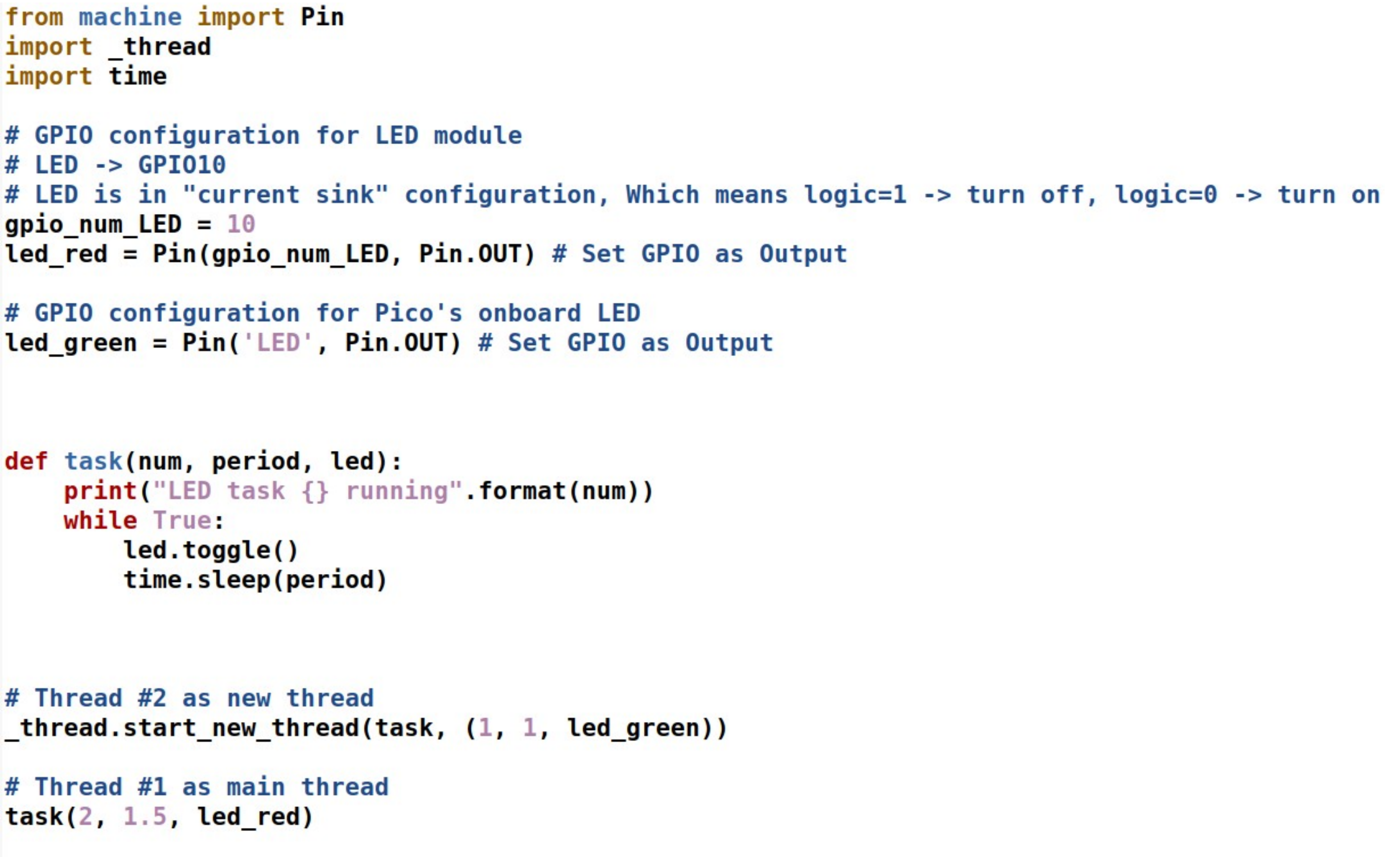
```
hello main
hello task 1
hello task 3
hello task 2
hello task 2
hello task 1
hello task 2
hello task 2
hello task 1
hello task 3
hello task 2
hello task 2
hello main
hello task 1
hello task 2
hello task 2
hello task 1
hello task 3
```

MultiThread in RP2040 (Pico)

- Sadly, For the current RP2040 micropython firmware, We can create only 2 Threads (1 Thread/Core)
- No thread switching…
- Still in development



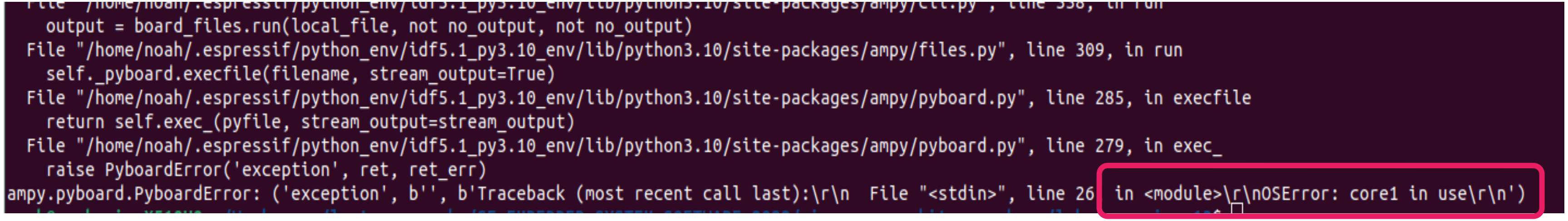
MultiThread in RP2040 (Pico)



```
from machine import Pin
import _thread
import time

# GPIO configuration for LED module
# LED -> GPIO10
# LED is in "current sink" configuration, Which means logic=1 -> turn off, logic=0 -> turn on
gpio_num_LED = 10
led_red = Pin(gpio_num_LED, Pin.OUT) # Set GPIO as Output

# GPIO configuration for Pico's onboard LED
led_green = Pin('LED', Pin.OUT) # Set GPIO as Output



def task(num, period, led):
    print("LED task {} running".format(num))
    while True:
        led.toggle()
        time.sleep(period)



# Thread #2 as new thread
_thread.start_new_thread(task, (1, 1, led_green))

# Thread #1 as main thread
task(2, 1.5, led_red)
```

MultiThread in RP2040 (Pico)

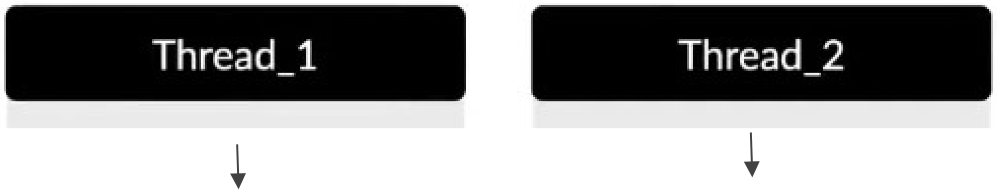
- Error in RP2040 when tried to create more than 2 threads.



```
  File "/home/noah/.espressif/python_env/idf5.1_py3.10_env/lib/python3.10/site-packages/ampy/cli.py", line 338, in run
    output = board_files.run(local_file, not no_output, not no_output)
  File "/home/noah/.espressif/python_env/idf5.1_py3.10_env/lib/python3.10/site-packages/ampy/files.py", line 309, in run
    self._pyboard.execfile(filename, stream_output=True)
  File "/home/noah/.espressif/python_env/idf5.1_py3.10_env/lib/python3.10/site-packages/ampy/pyboard.py", line 285, in execfile
    return self.exec_(pyfile, stream_output=stream_output)
  File "/home/noah/.espressif/python_env/idf5.1_py3.10_env/lib/python3.10/site-packages/ampy/pyboard.py", line 279, in exec_
    raise PyboardError('exception', ret, ret_err)
ampy.pyboard.PyboardError: ('exception', b'', b'Traceback (most recent call last):\r\n  File "<stdin>", line 26, in <module>\r\nOSError: core1 in use\r\n')
```

MultiThread in RP2040 (Pico)

Lab1 : Blinking 2 LEDs with different period.

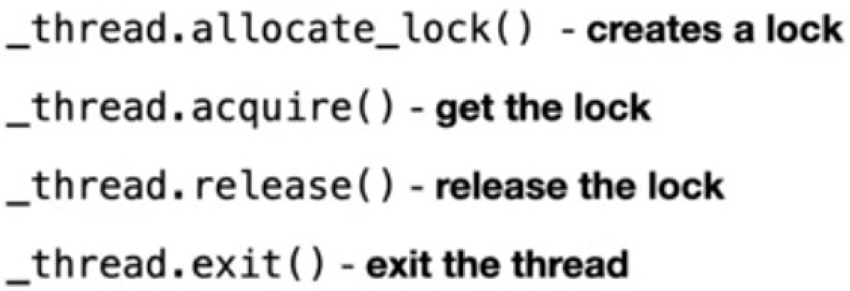


```
led_green.toggle()
time.sleep(2)
```

```
led_red.toggle()
time.sleep(3)
```

Mutex

- Using **lock** in micropython



_thread.allocate_lock() - **creates a lock**

_thread.acquire() - **get the lock**

_thread.release() - **release the lock**

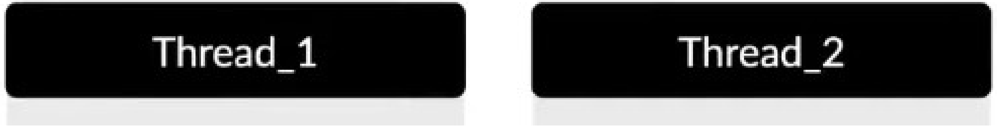
_thread.exit() - **exit the thread**

When/Where should MUTEX be used?

- Shared memory (Global variable) that “Race Condition” can be occured.
- Shared I/O. Example, GPIO, Communication Peripheral (UART, I2C, SPI etc)

MultiThread in RP2040 (Pico)

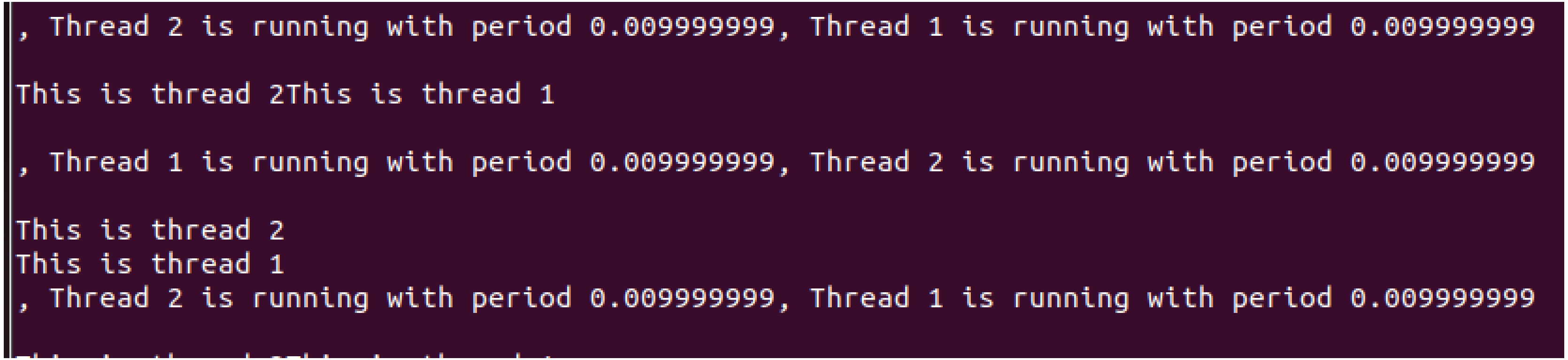
Lab2 : Print thread.



```
print("This is thread {}".format(thread_num))
print(", Thread {} is running with period {}".format(thread_num, period))
time.sleep(period)
```

MultiThread in RP2040 (Pico)

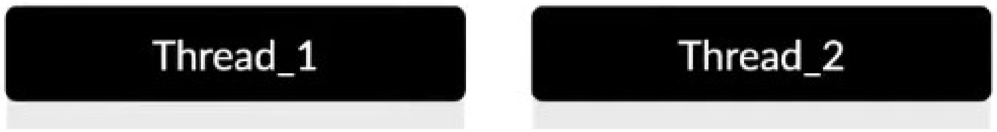
Lab2 : Print thread (Result)



```
, Thread 2 is running with period 0.009999999, Thread 1 is running with period 0.009999999

This is thread 2This is thread 1

, Thread 1 is running with period 0.009999999, Thread 2 is running with period 0.009999999

This is thread 2
This is thread 1
, Thread 2 is running with period 0.009999999, Thread 1 is running with period 0.009999999
```

MultiThread in RP2040 (Pico)

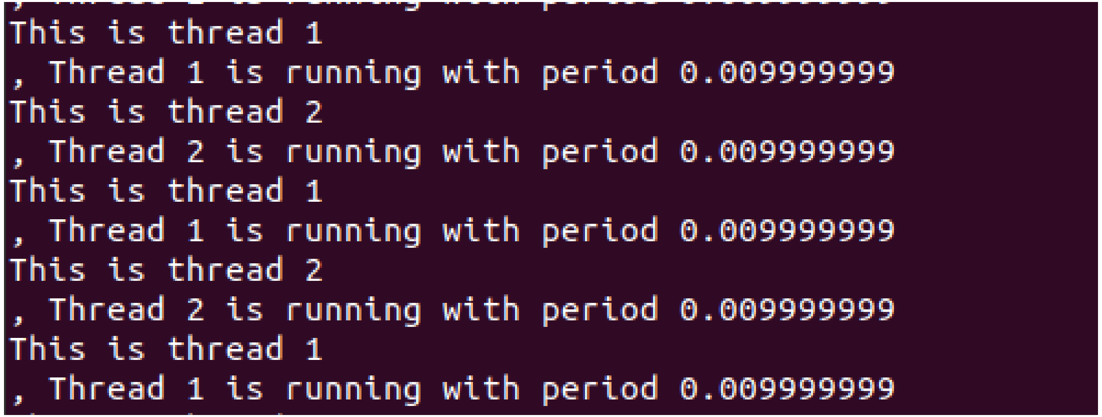
Lab2 : Print thread with MUTEX lock



```
lock.acquire()
print("This is thread {}".format(thread_num))
print(", Thread {} is running with period {}".format(thread_num, period))
lock.release()

time.sleep(period)
```

MultiThread in RP2040 (Pico)

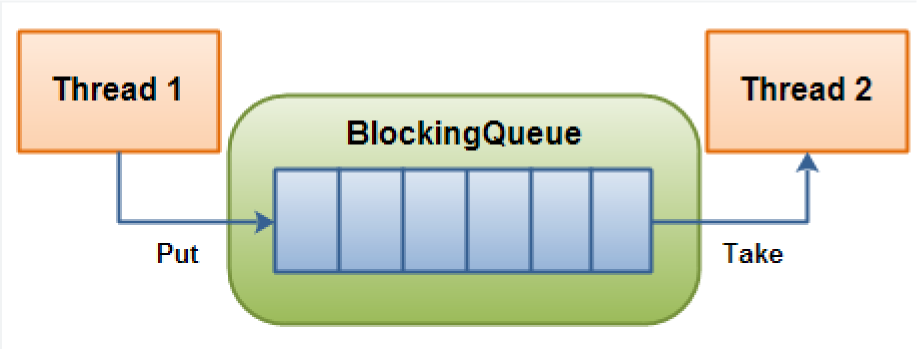
Lab2 : Print thread with MUTEX lock (Result)

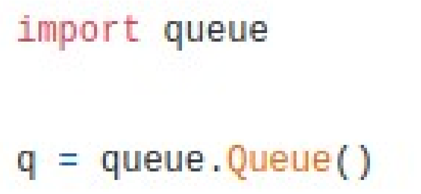


```
This is thread 1
, Thread 1 is running with period 0.009999999
This is thread 2
, Thread 2 is running with period 0.009999999
This is thread 1
, Thread 1 is running with period 0.009999999
This is thread 2
, Thread 2 is running with period 0.009999999
This is thread 1
, Thread 1 is running with period 0.009999999
```

Messaging between threads

- **Global variable** can be used. But It needs **MUTEX**!
- **Queue** is thread-safe. It can be used and eliminates the race condition.

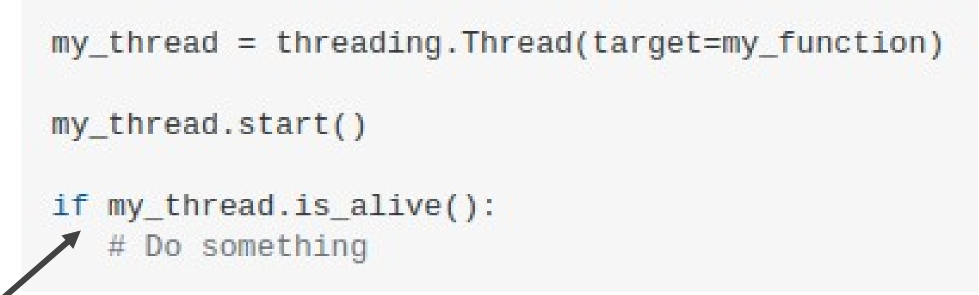




```
import queue

q = queue.Queue()
```

Monitoring other threads in main thread

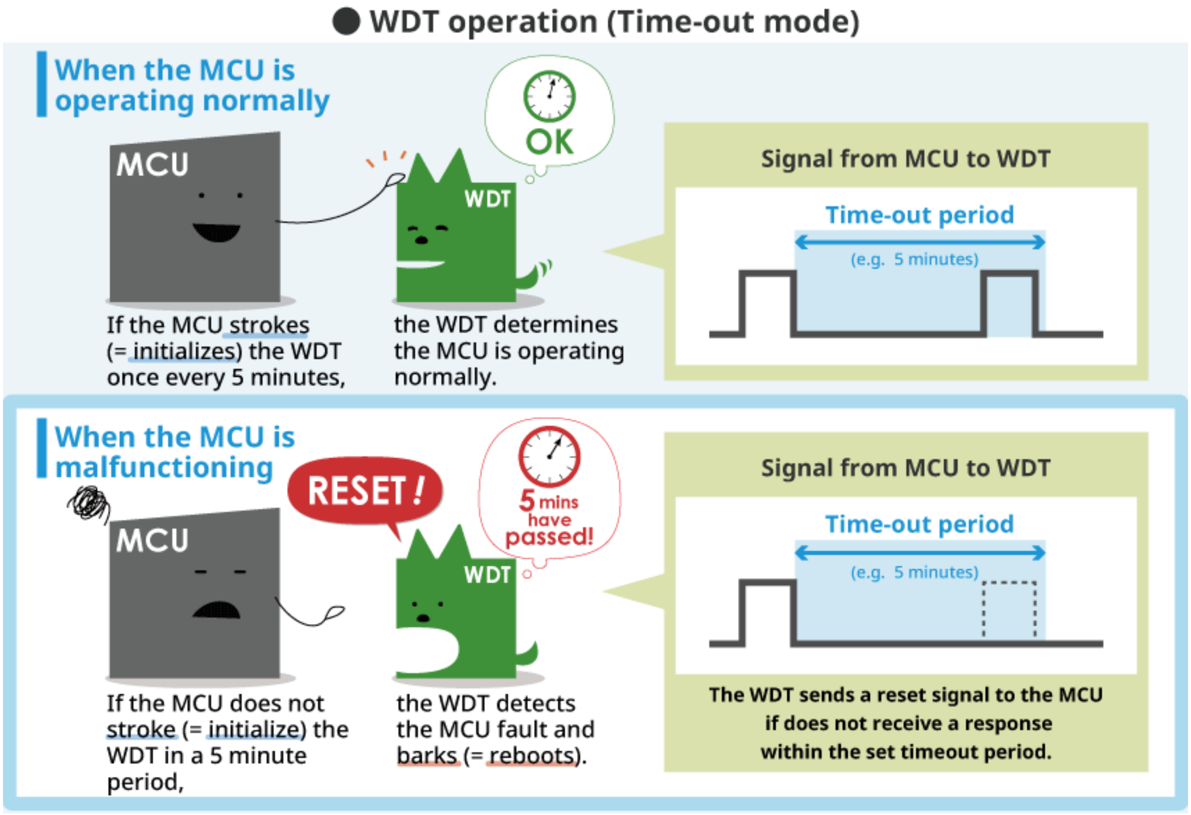
- **Main thread** can run the status check on each threads
- **Watchdog Timer/Timeout Timer** technique can be used.



```
my_thread = threading.Thread(target=my_function)

my_thread.start()

if my_thread.is_alive():
  # Do something
```

Threading is not available in micropython, Need to implement this concept by yourself

About Watchdog



● WDT operation (Time-out mode)

When the MCU is operating normally

If the MCU strokes (= initializes) the WDT once every 5 minutes,

the WDT determines the MCU is operating normally.

When the MCU is malfunctioning

If the MCU does not stroke (= initialize) the WDT in a 5 minute period,

the WDT detects the MCU fault and barks (= reboots).

The WDT sends a reset signal to the MCU if does not receive a response within the set timeout period.

Pro/Con

Pro

- Python threading is optimized for I/O bound tasks.
- Scale-up Task more easier.

Con

- Get overhead from context switching.
- Difficult to debug when dealing with a lot of threads.
- Problem with shared resource between threads (MUTEX, Semaphore is needed)