Concurrent Programming in Constrained Devices

01219335 Data Acquisition and Integration

Chaiporn Jaikaeo

Department of Computer Engineering
Kasetsart University

Revised 2021-09-15

Outline

- Handling concurrency
- Example of concurrent tasks
- Event-driven programming
- Multithreading
- Coroutines
- Concurrency and MQTT

Handling Concurrency

- IoT applications tend to get too complex to be implemented as a sequential program
- E.g., the app needs to
  - Monitor various sensor status
  - Wait for and respond to user switch
  - Wait for and respond to requests from the network

Example: Concurrent Tasks

- Create an application that performs the following subtasks concurrently
  - Subtask #1

    repeatedly blinks the red LED for a short time period (100 ms) every 2 seconds
  - Subtask #2

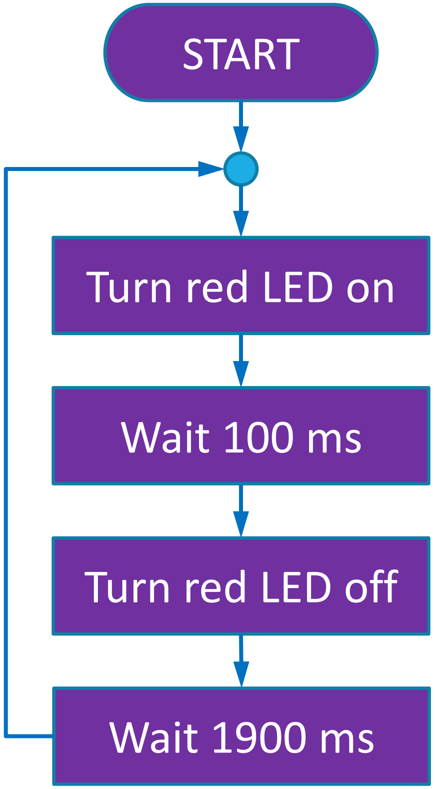
    when switch S1 is pressed, toggles the green LED

Subtask #1 Only (No Concurrency)

blinks red LED shortly (100 ms) every 2 seconds

```
from machine import Pin
import time

led_red = Pin(2, Pin.OUT)
while True:
    led_red.value(0)  # turn LED on
    time.sleep_ms(100)
    led_red.value(1)  # turn LED off
    time.sleep_ms(1900)
```



Subtask #2 Only (No Concurrency)

Toggle green LED when S1 is pressed

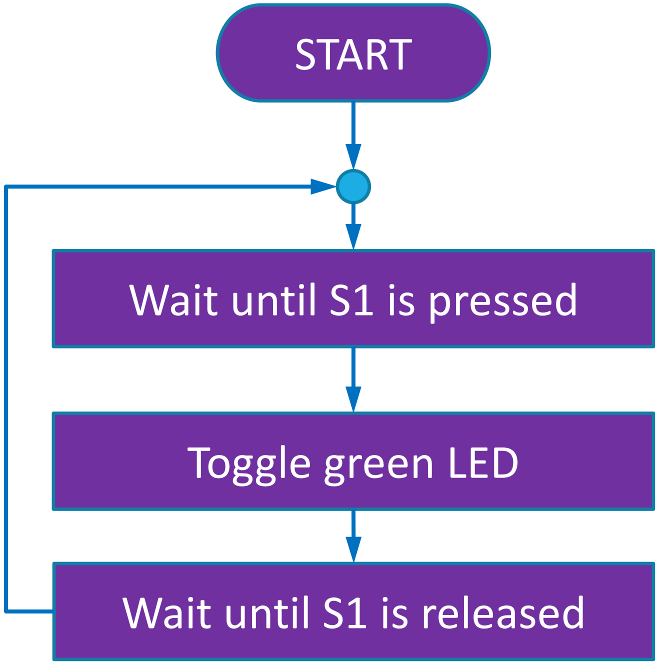
```python
from machine import Pin

led_green = Pin(12, Pin.OUT)
sw1 = Pin(16, Pin.IN, Pin.PULL_UP)

while True:
    # wait until S1 is pressed
    while sw1.value() == 1:
        pass

    # toggle LED
    led_green.value(1-led_green.value())

    # wait until S1 is released
    while sw1.value() == 0:
        pass
```



Attempt to Combine Subtasks

- Cannot just put one subtask after another

This will not work! Why?

```
from machine import Pin
import time

led_red = Pin(2, Pin.OUT)
led_green = Pin(12, Pin.OUT)
sw1 = Pin(16, Pin.IN, Pin.PULL_UP)

while True:
    # subtask #1
    led_red.value(0)  # turn LED on
    time.sleep_ms(100)
    led_red.value(1)  # turn LED off
    time.sleep_ms(1900)

    # subtask #2
    while sw1.value() == 1: # wait until S1 is pressed
        pass
    led_green.value(1-led_green.value())
    while sw1.value() == 0: # wait until S1 is released
        pass
```

blocking

blocking

blocking

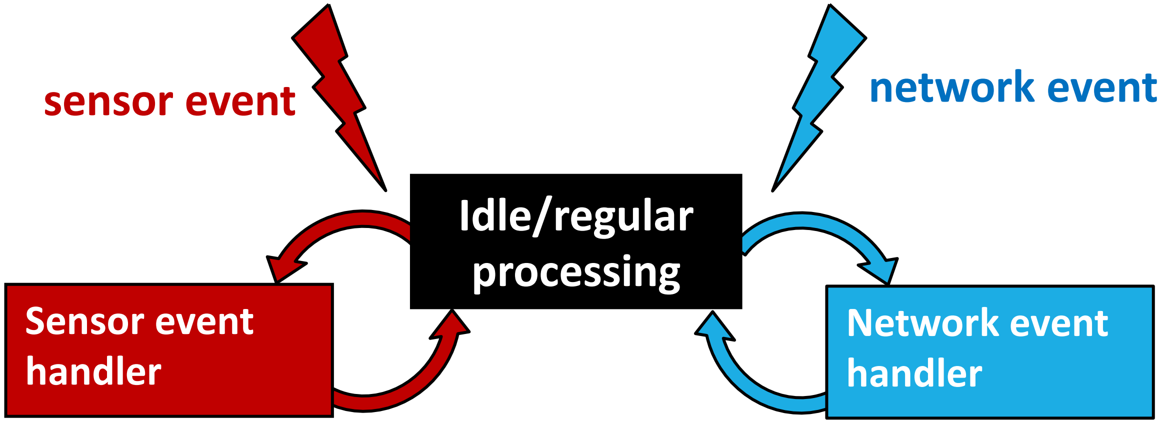
blocking

Concurrency Programming Models

- Event driven
- Multithreading
- Coroutines

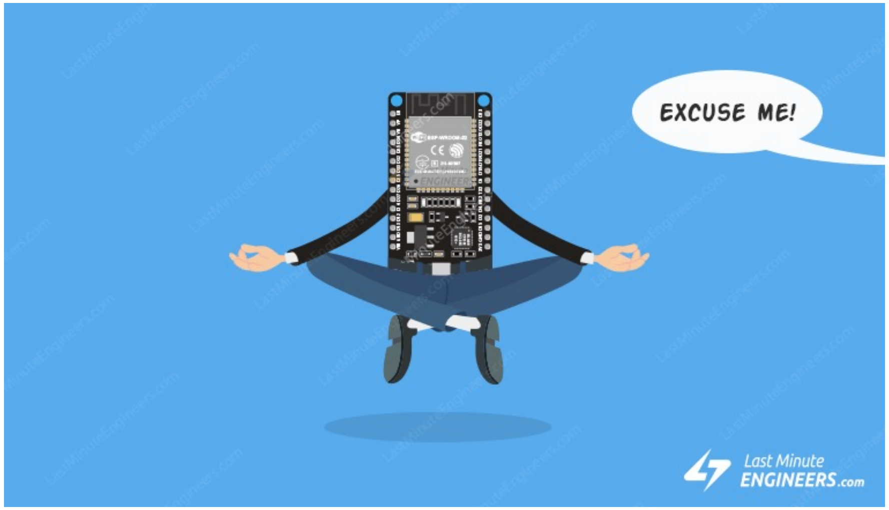
(1) Event-Driven Model

- Also known as **reactive programming**
- Perform regular processing or be idle
- React to events when they happen immediately
- To save power, CPU can be put to sleep during idle



Event Sources

- Hardware (external) interrupts
  - E.g., pin logic changes
  - Use `Pin.irq()` method to set up a callback
- Software interrupts
  - E.g., Timer, incoming network messages



Event-Driven Code

```
import time
from machine import Timer, Pin

led_red = Pin(2, Pin.OUT)
led_green = Pin(12, Pin.OUT)
sw1 = Pin(16, Pin.IN, Pin.PULL_UP)

def blink1(timer):
    led_red.value(0)  # turn LED on
    timer.init(period=100,
               mode=Timer.ONE_SHOT,
               callback=blink2)

def blink2(timer):
    led_red.value(1)  # turn LED off
    timer.init(period=1900,
               mode=Timer.ONE_SHOT,
               callback=blink1)
```

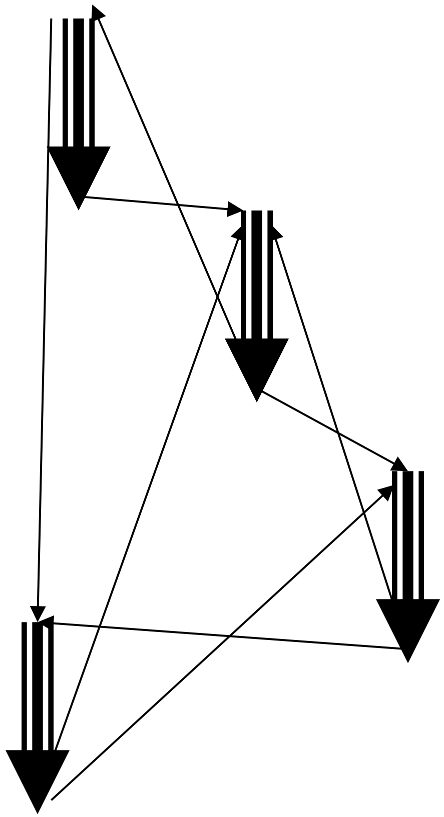
```
def switch_pressed(pin):
    led_green.value(1-led_green.value())

timer = Timer(0)
blink1(timer)
sw1.irq(trigger=Pin.IRQ_FALLING,
        handler=switch_pressed)

while True:
    time.sleep(1)
```

Problems with Event-Driven Model

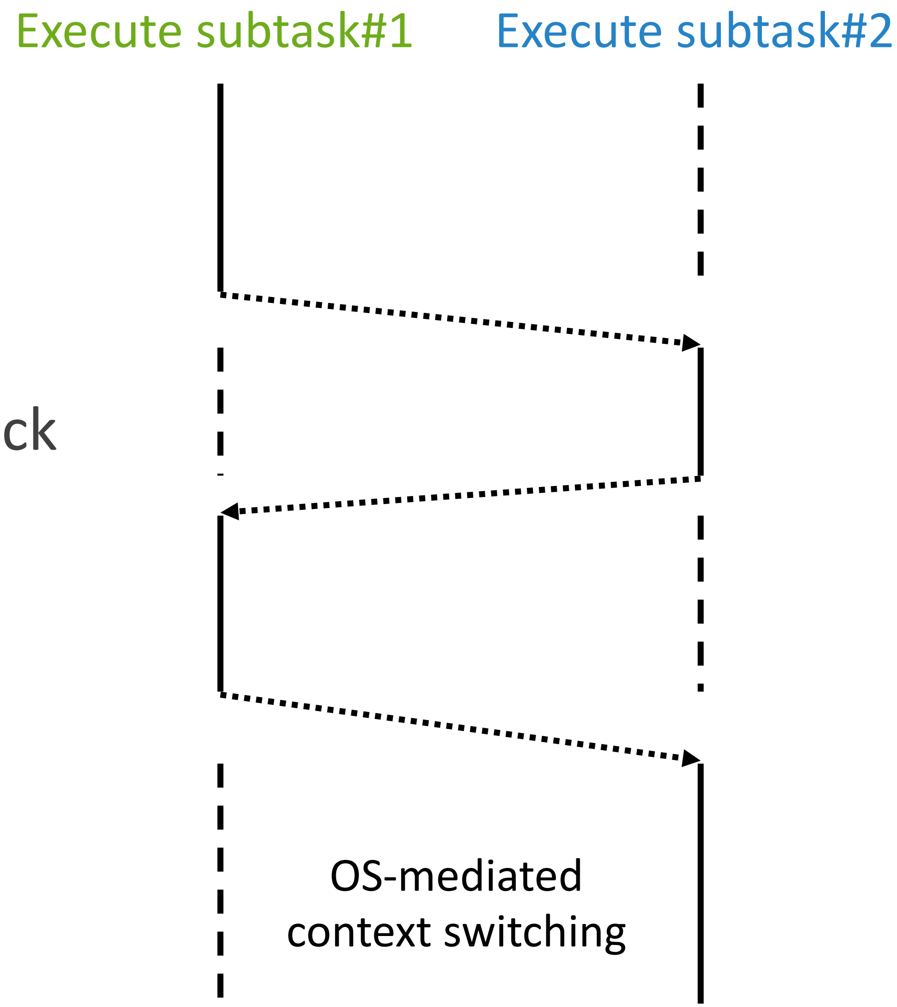
- Unstructured code flow
- Requires global variables to keep track of task's states



Very much like programming with GOTOs!

(2) Multithreading Model

- Based on interrupts, context switching
- Difficulties
  - Too many context switches
  - Each process required its own stack
- Not much of a problem on modern microcontrollers



Multithreading Code

```
import _thread
import time
from machine import Timer, Pin

led_red = Pin(2, Pin.OUT)
led_green = Pin(12, Pin.OUT)
sw1 = Pin(16, Pin.IN, Pin.PULL_UP)

def blink_led():
    while True:
        led_red.value(0)
        time.sleep_ms(100)
        led_red.value(1)
        time.sleep_ms(1900)
```

Thread #1

Thread #2

```
def switch_toggle():
    while True:
        # wait until sw is pressed
        while sw1.value() == 1:
            pass

        # toggle LED
        led_green.value(1-led_green.value())

        # wait until sw1 is released
        while sw1.value() == 0:
            pass

# create and run threads
_thread.start_new_thread(blink_led, [])
_thread.start_new_thread(switch_toggle, [])

while True:
    time.sleep(1)
```

Problems with Threads

- Code employing multithreads must ensure **thread-safe** operations
- A function may be interrupted in the middle of its execution where it shouldn't be, causing **race conditions**
  - Use a **Lock** object to create a mutex

(3) Coroutines

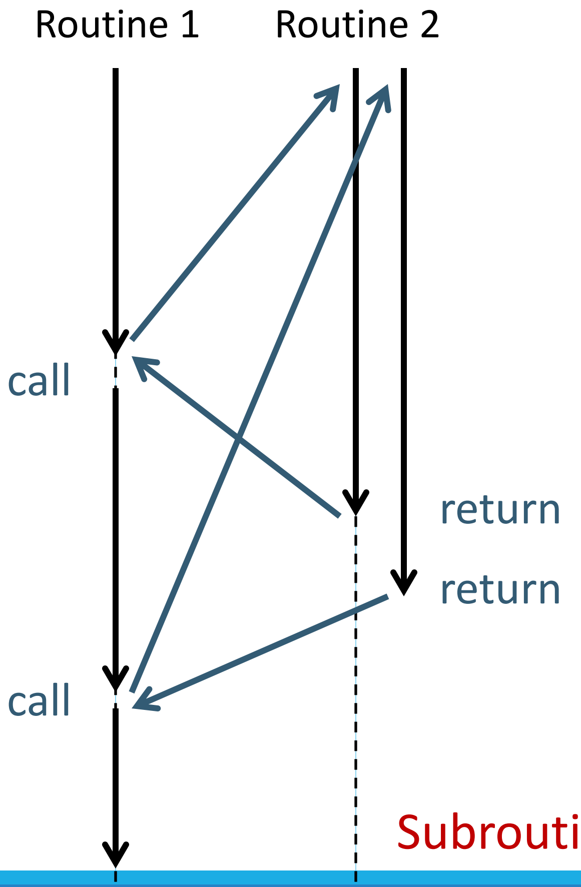
- Generalized subroutines
  - Allow multiple entry points for suspending and resuming execution at certain locations
- Can be used to implement Cooperative Multitasking
- As coroutines do not return, tasks' states can be kept inside local variables
- No worry about thread-safe operations
- May be known by different names in some languages:
  - Green threads
  - Fibers

Subroutines vs. Coroutines

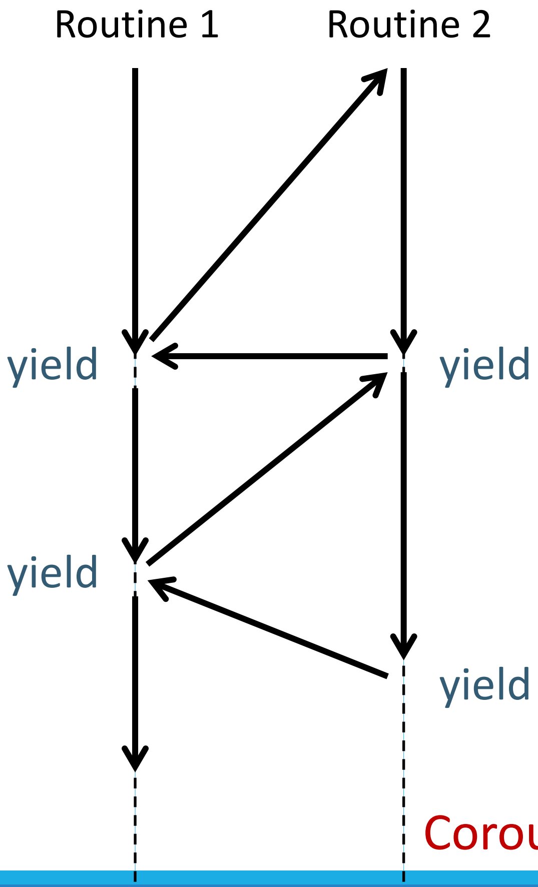
"Subroutines are a special case of coroutines."

--Donald Knuth

Fundamental Algorithms. The Art of Computer Programming



Subroutines



Coroutines

Coroutines in MicroPython

- Can be achieved using the `uasyncio` module with async/await pattern
- The uasyncio module needs to be installed separately
  - Already built into the course's MicroPython firmware

Coroutines: Code

```
import uasyncio as asyncio
import time
from machine import Timer, Pin

led_red = Pin(2, Pin.OUT)
led_green = Pin(12, Pin.OUT)
sw1 = Pin(16, Pin.IN, Pin.PULL_UP)
```

Coroutine #1

```
async def blink_led():
    while True:
        led_red.value(0)
        await asyncio.sleep_ms(100)
        led_red.value(1)
        await asyncio.sleep_ms(1900)
```

Coroutine #2

```
async def switch_toggle():
    while True:
        # wait until sw is pressed
        while sw1.value() == 1:
            await asyncio.sleep_ms(0)

        # toggle LED
        led_green.value(1-led_green.value())

        # wait until sw1 is released
        while sw1.value() == 0:
            await asyncio.sleep_ms(0)

# create and run coroutines
loop = asyncio.get_event_loop()
loop.create_task(blink_led())
loop.create_task(switch_toggle())
loop.run_forever()
```

Problems with Coroutines/asyncio

- Usually run on a single thread and not employ multiple cores
- Due to their cooperative nature, long-running function calls render all other coroutines unresponsive
  - May miss some hardware events

MQTT and Concurrency

- To handle incoming messages, `umqtt` requires `check_msg()` method to be called regularly
- A subtask (thread or coroutine) can be created to ensure `check_msg()` is continuously called
    - In case of coroutines, do not forget to yield, e.g.,

```
async def check_mqtt():
    while True:
        mqtt.check_msg()
        await asyncio.sleep_ms(0)
```

Conclusion

- Complex tasks often require concurrency
- Three concurrency programming models
  - Event-driven
  - Multithreading
  - Coroutines

Assignment 5.1: IoT Light Control

- Make a device to allow lamp control from (1) local S1 switch, and (2) the Internet
  - When S1 is pressed, the lamp's state is toggled
  - Publish lamp state to `ku/daq2021/<sid>/lamp/2` only when it is changed
    - Payload 0 → Lamp is off
    - Payload 1 → Lamp is on
  - Also subscribe to `ku/daq2021/<sid>/lamp/2` to allow controlling the LED from the Internet

http://clipart-library.com/clipart/1168222.htm