Workers-Framework

A simpler way to MultiTask in MicroPython

Sharil Tumin
sharil@trimensity.tech

Contents

Dedication

To my one and only ***SBE***. You shine the brightest in my darkest night.

Preface

Hello, how are you doing? It's nice of you to stop by. In your hands is a book titled, ***Workers-Framework: A simpler way to MultiTask in MicroPython***. I discovered an easier way to multitask on microcontrollers by using the high-level language MicroPython. Yes, really. The concept is to run all of the *tasks* defined in a program on multiple *workers* at the same time. It's less complicated than using *_thread* or *asyncio*.

The `worker` module is so simple that calling it a framework seems a little pretentious. A framework, on the other hand, is a set of data and program structure in which tasks are performed and successfully completed. A framework typically compels developers to solve specific problems in a specific way. A framework, in a sense, creates a mindset for the programming problems at hand and how to solve them.

If you're curious, keep reading. I assure you that it will be worthwhile.

Multitasking is the process of performing multiple tasks at the same time while sharing a single resource, in this case, the *CPU*. Microprocessor edge programming is inherently *event-driven*, *IO-bound* multitasking. It is more difficult because of limited computing resources.

So you've learned about MicroPython[1]. That's fantastic. I'm glad.

I assumed you were familiar with MicroPython programming and its various ports and libraries. This is not a primer on MicroPython.

[1] My thanks go to Damien George, the creator of MicroPython, and Guido van Rossum, the creator of Python.

As you may be aware, MicroPython is not the same as regular *Python* on a PC. You cannot simply import Python libraries, such as `import threading`, and expect them to work. If you're expecting that, you'll be sorely disappointed. In this regard, MicroPython is inferior to Python.

In MicroPython, however, you can `import machine` but not in regular Python. If you work with microcontrollers, MicroPython is unquestionably the best high-level programming environment available[2]. MicroPython, without a doubt, outperforms Python in the realm of microcontrollers. On a *Linux* machine, MicroPython runs faster than regular Python.

MicroPython was built from the ground up, from the Python language specificatiosn, to run in a resource-constrained environment (CPU, RAM, ROM). There is no *operating system* in between MicroPython code and hardware GPIO, RTC, and peripherals. The *OS* is the MicroPython firmware.

Standard best practices for writing *Pythonic* code may not be the best way to write MicroPython programs due to resource constraints.

There are some unusual codes here, such as short variable names (a single character), variable name reuse, compact expressions, breaking normal Python code conventions, and what are normally called "poor programming techniques". Please accept my apologies. I anticipate that readers of this book will know what is best for them and will stick to their preferred programming styles. In my opinion, a good program is one that works, and 100 lines of code is far too long. By the way, I use *vim* to code all of my programs; no *auto-completion* and no *auto-correction*.

You are free to use whatever works best for you and in whatever style you prefer. Don't be afraid to break some rules in order to learn. After all, I believe programming is an art form, and we have the freedom to practice it as artists.

This book is about writing concurrent tasks in MicroPython. Things

[2] My personal opinion. I experimented with Basic, Forth, Lua, Javascript (Espruino), and C++ (Arduino). MicroPython provides the most comprehensive coverage.

happened all the time in the real world. Ready or not, they keep on coming. Writing programs for microcontrollers expose us to real concurrency of events. This is what make it fun. Learning to control concurrency and be able to reason it all out using simple principles is the main goal of this book. I've learn to do just that and I would like to share it with you.

This book teaches you how to write concurrent tasks in MicroPython. In the real world, things happen all the time. They keep coming, whether you're ready or not. Writing programs for microcontrollers exposes us to real-world events. This is what makes it interesting. The main goal of this book is to teach you how to control concurrency and how to reason about it using simple principles. I've learned how to do just that, and I'd like to share my knowledge with you.

You may be unaware of the simple method used to achieve this. You may have seen it before, perhaps in an iterator.

It is not a new discovery; in fact, the same principle was used to program the "Apollo Guidance Computer" nearly 55 years ago[3]. In 1973, the model was formally defined[4]. Older technologies that still work are easily overlooked because they are overshadowed by newer mainstream technologies supported by GB of memory and GHz multicore CPUs.

We are forced to think differently when using MicroPython. It's a good thing, in my opinion. It democratizes programming and liberates coders from the shackles of large corporations' expensive compilers, GUIs, and toolchains. The Microcontroller and MicroPython offer a low-cost entry point for anyone, anywhere who wants to learn to program.

An old laptop, a few inexpensive microcontrollers, LEDs, displays, sensors, and DC motors are all that is required. We are now prepared

[3] Robert Wills, Light Years Ahead|The 1969 Apollo Guidance Computer. https://www.youtube.com/watch?v=B1J2RMorJXM

[4] Hewitt, Carl; Bishop, Peter; Steiger, Richard (1973). "A Universal Modular Actor Formalism for Artificial Intelligence". IJCA

to have fun while learning.

A program listing is given in blue throughout the book,

```
# hello.py
print("Hello world!")
```

and the terminal input and output are given in brown.

```
$ micropython hello.py
Hello world!
```

MicroPython is constantly evolving. A new version is released on a regular basis. However, the Python runtime engine code is stable. In its implementation, the *Workers- Framework* makes use of standard *yield/send*. It will continue to work in future MicroPython versions.

This *workers-framework*, which only allows you to program concurrent tasks on a single thread, will not make your code run faster. *Concurrency* is not the same as *parallelism*. However, the concurrent multitasking framework will provide you with programming tools to easily and naturally solve concurrency problems.

For a quick learning path, read chapters **Introduction** on page 9 and **Implementation** on page 45. After that, you'll be able to experiment with *cooperative multitasking* programming in MicroPython using the *workers-framework*. I promise you, it's that easy.

For the impatient, the source code for the *workers-framework* modules can be found on the following pages: 1) `worker_lite.py` page 227, 2) `worker.py` page 229, and 3) `pipe.py` page 232. Of course, I'd be thrilled if you completed the book from beginning to end.

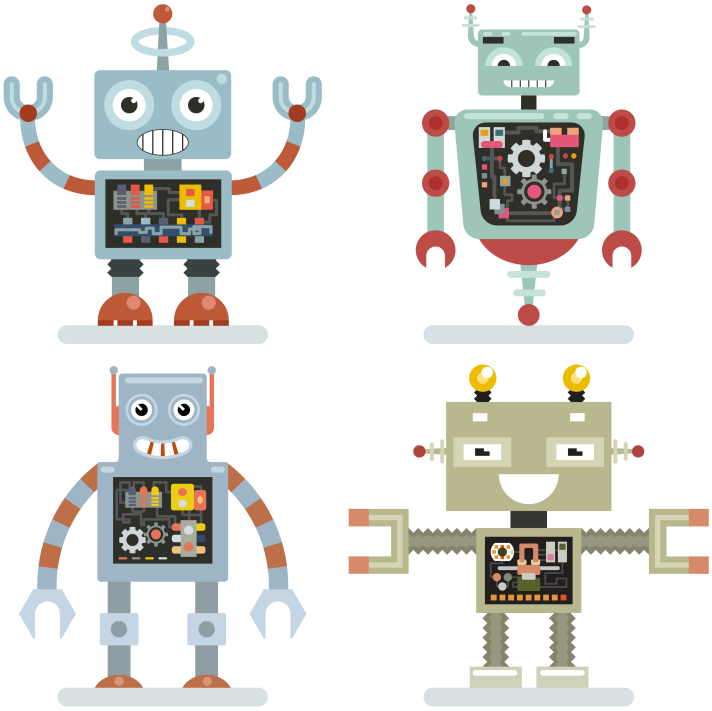
This is the first edition of this work. Please accept my apologies in advance for any errors or ambiguity in my reasoning or the example codes.

This book contains a lot of MicroPython code. Some are quite insignificant, while others can be quite useful. **Use cases** on page 139 provide some examples of *workers* use-cases.

Thank you for investing in this book. Your support is greatly appreciated. Once again, thank you.

There is a clever way and a smart way. The choice is entirely yours.

Introduction



Tasks

Assume we have four long-running tasks: `a`, `b`, `c`, and `d`. For the time being, let us assume they are logically independent of one another. They are *separated*. We normally execute these tasks in a predetermined order.

```
a()
b()
c()
d()
```

Typically, these tasks are long-running loops within an application. Two tasks, for example, read input from two input pins (switches) and two write to output pins (LEDs).

```
while do_tasks: # loop main
  a()
  b()
  c()
  d()
```

Regardless, with respect to timing, the execution steps will follow the order of:

$$t_0 < t_a < t_b < t_c < t_d < t_1$$

i.e t_0=start time, t_1=done time, and t_a=start a() time,

$$\begin{aligned} t_1 - t_0 &= (t_a - t_0) + (t_b - t_a) + (t_c - t_b) + (t_d - t_c) + (t_1 - t_d) \\ &= T_0 + T_a + T_b + T_c + T_d \\ &= T_s \end{aligned}$$

The total amount of time required to complete all tasks is T_s, which is the sum of setup time T_0 and the time required to complete each task in sequence.

We can carry out these tasks in *parallel* or *concurrently*. A parallel system can support two or more tasks running on multiple CPUs at the same time. A concurrent system, on the other hand, can support two or more tasks in progress at the same time in overlapping periods on a single CPU.

The T_p total execution period for parallelism is

$$T_p = T_0 + \min(T_a, T_b, T_c, T_d)$$

assuming the system has four cores, whereas the T_r total execution

period for concurrency is

$$T_r = T_0 + T_s + T_w$$

Following the T_0 setup period, we are actually time multiplexing the T_s period among a, b, c, and d tasks. Each small portion of a task receives its own slice of CPU time. Consider this: divide each task into three logical parts, such as

$$t_a = t_a^1 < t_a^2 < t_a^3$$

Interleaving our a, b, c, and d tasks will result in the following execution sequence:

$$t_0 < t_a^1 < t_b^1 < t_c^1 < t_d^1 < t_a^2 < t_b^2 < t_c^2 < t_d^2 < t_a^3 < t_b^3 < t_c^3 < t_d^3 < t_1$$

where each portion of a task advances at the same rate.

The overhead time T_w is the price we must pay for concurrency. The total overhead time period $T_0 + T_w$ will vary depending on the system and execution model.

$$T_p < T_s < T_r$$

Parallelism with disjointed tasks on multiple cores yields the best results. When possible, sequential execution outperforms concurrent execution. Aside from the overhead costs, we also have a significant problem with concurrency. What if one of the tasks becomes stalled while waiting for an external event, such as a signal from a GPIO pin? A blocked task will halt the entire loop, preventing us from progressing through the program and leaving us to do nothing but wait. If the anticipated event does not occur, our program is effectively halted.

Why would we want concurrency in the first place? This is due to the fact that edge or embedded system programming is *event-driven* by signals transmitted via the *input/output* interface to real-time and real-world events. Our systems will waste a lot of time waiting on various asynchronous signals if we don't have concurrency.

As a result, it is critical to recognize that these four independent tasks

can be represented as four logically independent loops:

```
while do_a: # loop a
  a() # e.g. read pin1 (switch 1)
while do_b: # loop b
  b() # e.g. read pin2 (switch 2)
while do_c: # loop c
  c() # e.g. write pin3 (LED 1)
while do_d: # loop d
  d() # e.g. write pin4 (LED 2)
```

Of course, such a simplistic arrangement will not work. Because it is too busy doing `a()`, our program will become stuck in *loop a*, while `b()`, `c()`, and `d()` will never be executed. We definitely need a way to break out of each loop so that each task has a chance to run. What we need is a way to run all loops *concurrently*. In general, what we require is:

```
par:
   while do_a: # loop a
     a() # e.g. read pin1 (switch 1)
   while do_b: # loop b
     b() # e.g. read pin2 (switch 2)
   while do_c: # loop c
     c() # e.g. write pin3 (LED 1)
   while do_d: # loop d
     d() # e.g. write pin4 (LED 2)
```

Unfortunately, the `par:` block does not exist in MicroPython syntax (not yet, though it would be nice to have it). However, in standard MicroPython, we can write multitasking programs by using:

1. `_thread` - preemptive scheduling,
2. `uasyncio` - cooperative scheduling.

But we're not going to go into detail about them. Only one example of each will be shown later. Instead, we will create a simple *workers-framework* that will provide us with a lightweight *cooperative scheduler* to support multitasking runtime.

Tasks Scheduling

In a multitasking system, shared resources include the central processing unit (CPU) and random-access memory (RAM). The MicroPython firmware decides how long a task should run before allowing another task to use the CPU during preemptive scheduling. In cooperative scheduling, programmers control the context switch from one task to another.

In the case of preemptive scheduling, the scheduling is fully controlled by the firmware, whereas in cooperative scheduling, the scheduling is fully controlled by the programmers. As the examples below demonstrate, two logically similar programs will behave differently.

Preemptive Scheduling

With the preemptive scheduling method, we will use the standard `_thread` to run four simple tasks. Please be aware that `_thread` is not supported by all MicroPython ports.

```
# thread_sched_test.py
from time import ticks_ms, ticks_diff
import _thread as th

cnt=0;tot=0
def test(w,n):
    global cnt,tot
    for i in range(n):
        cnt+=1
        print(cnt, w, i)
    tot+=ticks_diff(ticks_ms(),now)/1000.0
    print('CNT', cnt, w, "finished after", tot)

now=ticks_ms()
t0=th.start_new_thread(test, ("WWWW", 100))
t0=th.start_new_thread(test, ("XXXX", 100))
t1=th.start_new_thread(test, ("YYYY", 100))
t1=th.start_new_thread(test, ("ZZZZ", 100))
```

The results of the test are shown below:

```
>>> import thread_sched_test
>>> 1 ZZZZ2  0XXXX3
 04YYYY
 560  WWWW
ZZZZXXXX07
 11YYYY8

  91 WWWW10
ZZZZ  11 1XXXX 2
 YYYY
122 13
2 WWWW14
ZZZZ  15 2XXXX 3
 YYYY
163 17
3 WWWW18
ZZZZ  19 3XXXX 4

.... more output here ....

 YYYY39339498
97395WWWWXXXX
   396ZZZZ9798  YYYY

99 397398
98
WWWWCNTXXXX 399  98398 99
 YYYY
 400ZZZZ99 CNT
WWWW finished afterCNT 400  99 0.006400
XXXX
  CNTYYYYfinished after   400finished after0.012
WWWW0.018
finished after 0.024

>>>
```

We see output from different tasks interfering with each other as a result of preemption. This problem can be solved using *lock*; 1) `lok = th.allocate_lock()`, 2) `lok.acquire()`, and 3) `lok.release()`. A *lock* is used to restrict access to a program's *critical region*, such as a `print()` statement.

It's useful to know that a `time.sleep()` statement can force a context switch to occur.

A *multi-threaded* program will require more RAM than a *single-threaded* program. A 10-thread program will require `10*stack_size` of RAM. In MicroPython, the `stack_size` is 2KBytes.

Cooperative Scheduling

Now is a good time to show how to use the *workers-framework* in a program. We will use a cooperative scheduling method to have `workers` complete four simple tasks concurrently.

```
# worker_sched_test.py
from time import ticks_ms, ticks_diff
from worker import task, MT

cnt=0;tot=0
@task
def test(pm):
    global cnt,tot
    w,n=pm
    c=(yield)
    for i in range(n):
        cnt+=1
        print(cnt, w, i)
        if i%10==0: yield
    tot+=ticks_diff(ticks_ms(),now)/1000.0
    print('CNT', cnt, w, "finished after", tot)
    return

mt=MT(4)
mt.worker(test, ("WWWW", 100))
mt.worker(test, ("XXXX", 100))
mt.worker(test, ("YYYY", 100))
mt.worker(test, ("ZZZZ", 100))
now=ticks_ms()
mt.start()
```

This is the result of the test.

```
>>> import worker_sched_test
1 WWWW 0
2 XXXX 0
3 YYYY 0
4 ZZZZ 0
5 WWWW 1
6 WWWW 2
```

```
7 WWWW 3
8 WWWW 4
9 WWWW 5

.... more output here ....

CNT 391 YYYY finished after 0.02
392 XXXX 91
393 XXXX 92
394 XXXX 93
395 XXXX 94
396 XXXX 95
397 XXXX 96
398 XXXX 97
399 XXXX 98
400 XXXX 99
CNT 400 XXXX finished after 0.027
>>>
```

Context switching occurs at all yields in the program. A task will run until it reaches a `yield` point, at which point the scheduler will resume a suspended task from the *workers* queue. Simple applications do not necessitate the use of locks.

Our `worker` module is a *single-threaded* library that uses less memory than `_thread`. On a similar runtime environment, we can thus start more *workers* than *threads*.

Concurrency Models

The `_thread` implementations are determined by the threading libraries that are supported by a specific operating system and hardware environment. Our `worker` modules and the standard `uasyncio` are written entirely in Python. The `uasyncio` may require more memory than the target microcontroller provides. Our `worker` module is significantly smaller and can be run on any MicroPython port.

A quick glance at the `uasyncio` files reveals the number of resources required:

```
$ ls micropython/extmod/uasyncio
core.py   funcs.py     lock.py      stream.py
event.py  __init__.py  manifest.py  task.py

$ wc micropython/extmod/uasyncio/*.py
  300  1010  9583 micropython/extmod/uasyncio/core.py
   61   237  1910 micropython/extmod/uasyncio/event.py
  126   530  4277 micropython/extmod/uasyncio/funcs.py
   30    84   709 micropython/extmod/uasyncio/__init__.py
   53   204  1767 micropython/extmod/uasyncio/lock.py
   13    28   313 micropython/extmod/uasyncio/manifest.py
  182   552  5026 micropython/extmod/uasyncio/stream.py
  177   683  5607 micropython/extmod/uasyncio/task.py
  942  3328 29192 total
```

This is in contrast to our much simpler `worker` module:

```
$ wc worker.py
  86  193 2070 worker.py
```

These will provide us with straightforward comparisons between `uasyncio` and `worker`:

1. number of lines 942//96=10
2. number of words 3328//223=17
3. number of bytes 29192//2390=14

The `uasyncio` is ten times larger than the `worker`. The `uasyncio`, of course, is far more sophisticated and complex than our simple `worker` module. Nonetheless, we will see later that the `worker` can support *cooperative multitasking* concurrency in a manner similar to `uasyncio`. In fact, we can improve performance by using the `worker` module.

The `uasyncio` module is included with the MicroPython distribution. Let us take a look at what is supported by *REPL*[5]:

```
>>> import uasyncio
>>> uasyncio.<TAB>
sleep          sleep_ms       sys            ticks_add
ticks_diff      core           funcs          wait_for
gather          event          Event          Lock
stream         __version__    _attrs        wait_for_ms
```

[5] <TAB> means entering a tab. REPL (Read-Eval-Print Loop) has a useful auto-complete feature.

```
ThreadSafeFlag  open_connection             start_server
StreamReader    StreamWriter    select         TaskQueue
Task          task          CancelledError  TimeoutError
SingletonGenerator              IOQueue         Loop
create_task     run_until_complete              run
get_event_loop  current_task    new_event_loop  ticks
>>> loop=uasyncio.Loop()
>>> loop.<TAB>
close          stop           __dict__       create_task
run_until_complete              call_exception_handler
run_forever    set_exception_handler        _exc_handler
get_exception_handler          default_exception_handler
```

Wait until you see the standard `asyncio` module for Python 3.8 if you think `uasyncio` is complicated.

```
>>> import asyncio
>>> asyncio.<TAB><TAB>
asyncio.ALL_COMPLETED
asyncio.AbstractChildWatcher(
asyncio.AbstractEventLoop(
asyncio.AbstractEventLoopPolicy(
asyncio.AbstractServer(
asyncio.BaseEventLoop(
asyncio.BaseProtocol(
asyncio.BaseTransport(
asyncio.BoundedSemaphore(
asyncio.BufferedProtocol(
asyncio.CancelledError(
asyncio.Condition(
asyncio.DatagramProtocol(
asyncio.DatagramTransport(
asyncio.DefaultEventLoopPolicy(
asyncio.Event(
asyncio.FIRST_COMPLETED
asyncio.FIRST_EXCEPTION
asyncio.FastChildWatcher(
asyncio.Future(
asyncio.Handle(
asyncio.IncompleteReadError(
asyncio.InvalidStateError(
asyncio.LifoQueue(
asyncio.LimitOverrunError(
asyncio.Lock(
asyncio.MultiLoopChildWatcher(
asyncio.PriorityQueue(
```

```
asyncio.Protocol(
asyncio.Queue(
asyncio.QueueEmpty(
asyncio.QueueFull(
asyncio.ReadTransport(
asyncio.SafeChildWatcher(
asyncio.SelectorEventLoop(
asyncio.Semaphore(
asyncio.SendfileNotAvailableError(
asyncio.StreamReader(
asyncio.StreamReaderProtocol(
asyncio.StreamWriter(
asyncio.SubprocessProtocol(
asyncio.SubprocessTransport(
asyncio.Task(
asyncio.ThreadedChildWatcher(
asyncio.TimeoutError(
asyncio.TimerHandle(
asyncio.Transport(
asyncio.WriteTransport(
asyncio.all_tasks(
asyncio.as_completed(
asyncio.base_events
asyncio.base_futures
asyncio.base_subprocess
asyncio.base_tasks
asyncio.constants
asyncio.coroutine(
asyncio.coroutines
asyncio.create_subprocess_exec(
asyncio.create_subprocess_shell(
asyncio.create_task(
asyncio.current_task(
asyncio.ensure_future(
asyncio.events
asyncio.exceptions
asyncio.format_helpers
asyncio.futures
asyncio.gather(
asyncio.get_child_watcher(
asyncio.get_event_loop(
asyncio.get_event_loop_policy(
asyncio.get_running_loop(
asyncio.iscoroutine(
asyncio.iscoroutinefunction(
asyncio.isfuture(
```

```
asyncio.locks
asyncio.log
asyncio.new_event_loop(
asyncio.open_connection(
asyncio.open_unix_connection(
asyncio.protocols
asyncio.queues
asyncio.run(
asyncio.run_coroutine_threadsafe(
asyncio.runners
asyncio.selector_events
asyncio.set_child_watcher(
asyncio.set_event_loop(
asyncio.set_event_loop_policy(
asyncio.shield(
asyncio.sleep(
asyncio.sslproto
asyncio.staggered
asyncio.start_server(
asyncio.start_unix_server(
asyncio.streams
asyncio.subprocess
asyncio.sys
asyncio.tasks
asyncio.transports
asyncio.trsock
asyncio.unix_events
asyncio.wait(
asyncio.wait_for(
asyncio.wrap_future(
>>> loop=asyncio.get_event_loop()
>>> loop.<TAB><TAB>
loop.add_reader(
loop.add_signal_handler(
loop.add_writer(
loop.call_at(
loop.call_exception_handler(
loop.call_later(
loop.call_soon(
loop.call_soon_threadsafe(
loop.close(
loop.connect_accepted_socket(
loop.connect_read_pipe(
loop.connect_write_pipe(
loop.create_connection(
loop.create_datagram_endpoint(
```

```
loop.create_future(
loop.create_server(
loop.create_task(
loop.create_unix_connection(
loop.create_unix_server(
loop.default_exception_handler(
loop.get_debug(
loop.get_exception_handler(
loop.get_task_factory(
loop.getaddrinfo(
loop.getnameinfo(
loop.is_closed(
loop.is_running(
loop.remove_reader(
loop.remove_signal_handler(
loop.remove_writer(
loop.run_forever(
loop.run_in_executor(
loop.run_until_complete(
loop.sendfile(
loop.set_debug(
loop.set_default_executor(
loop.set_exception_handler(
loop.set_task_factory(
loop.shutdown_asyncgens(
loop.slow_callback_duration
loop.sock_accept(
loop.sock_connect(
loop.sock_recv(
loop.sock_recv_into(
loop.sock_sendall(
loop.sock_sendfile(
loop.start_tls(
loop.stop(
loop.subprocess_exec(
loop.subprocess_shell(
loop.time(
```

A list of attributes and methods is three and a half pages long. However, two things are clear here:

1. `asyncio` - create and manage tasks,
2. `loop` - schedule and manage tasks execution.

All *input/output* must be *non-blocking* in order for the *cooperative scheduler* in both `uasyncio` and `asyncio` to function properly.

Our *Workers Framework* as implemented by the `worker` module, is far more straightforward.

```
>>> import worker
>>> worker.<TAB>
gc              utime           K               MT
nop             task            WS
>>> mt=worker.MT()
>>> mt.<TAB>
start           __dict__        log             s
x               purge           worker
>>> mt.s.<TAB>
get             __dict__        lock            put
V               v               unlock          mbox
delay
>>> worker.K.<TAB>
size            L               M               A
W               tail
```

In most cases, we only import `tasks` and `MT` into our codes.

```
from worker import task, MT
mt=MT(10) # create multitasking environment for
          # maximum 10 workers

@task
def fun(p): # must be a generator function (i.e with yield)
   n=p[0]
   s=yield  # s exposes multitasking related methods
   n+=2
   w = s.delay(10) # non-blocking delay
   while w(): yield
   print('done', n)
   return
mt.worker(fun, (2,)) # add 1.st worker
mt.worker(fun, (3,)) # add 2.nd worker
mt.start()
```

The `worker.MT` class manages *workers*, and the `worker.WS` class provides communication methods between workers via *message passing* using shared memory. The `worker` module focuses on cooperative multitasking rather than async multitasking. When using workers, the programmers are responsible for all workers' non-blocking and cooperative behavior. This means that a single bad worker can adversely affect the entire program. Every now and then, a running worker

must relinquish execution to the `worker.MT` scheduler so that other workers can progress. *Workers-Framework* operates at a lower level than `asyncio`. The `worker` module gives programmers more flexibility and freedom.

Most programmers prefer to believe that tasks are assigned in a sequential manner. Real-world tasks are rarely performed sequentially. The topic of *async multitasking* is extremely complex. Many interesting talks on the subject have been given at Python conferences[6][7][8]. It should come as no surprise that many different *async concurrency* modules[9], in addition to the standard asyncio, are already being developed. We must keep in mind that all of the underlying input and output functions implemented in these modules must be *cooperative* and *non-blocking*, which adds to the complexity.

The output from the REPL sessions above shows the difference in the number of attributes and methods between MicroPython `uasyncio`, Python 3.8.x `asyncio`, and our `worker` modules. Appendix 1 starting on page 227 contains the complete source code for the worker module. You may freely copy, modify, and reuse them in your own projects. It's absurdly simple, but extremely effective.

Generator

To use `uasyncio` modules, you must be well-versed in `async`/`await` programming constructs. The same is true for the `worker` module. We need to understand how `yield`/**`next`** and `yield`/`send` work. The Python *generator* is hidden beneath these modules.

We can define a function that behaves like an *iterator* using generator

[6] David Beazley. Python Concurrency From the Ground Up: LIVE! PyCon 2015. https://www.youtube.com/watch?v=MCs5OvhV9S4

[7] David Beazley. Fear and Awaiting in Async: A Savage Journey to the Heart of the Coroutine Dream. https://www.youtube.com/watch?v=E-1Y4kSsAFc

[8] Nathaniel J. Smith. Trio: Async concurrency for mere mortals PyCon 2018. https://www.youtube.com/watch?v=oLkfnc_UMcE

[9] twisted, tornado, gevent, greenlet, curio, trio …

functions. When called, such a function will return an iterator that can be used in a `for` loop. To clarify, consider the following generator function example:

```
>>> def countdown():
...     print("Commencing countdown")
...     n=yield
...     while n>=0:
...         yield n
...         n-=1
...     yield "Blast off!"
...
>>> c=countdown()
>>> next(c)
Commencing countdown
>>> c.send(5)
5
>>> for i in c:
...     print(i)
...
4
3
2
1
0
Blast off!
>>>
```

We can begin building a simple cooperative scheduler for our multitasking concurrency framework once we understand how Python `yield` works.

To begin, calling a generator function returns a generator object:

```
>>> c=countdown()
>>> c
<generator object 'countdown' at 3ffef560>
>>> c.<TAB>
close      send       throw      pend_throw
```

A value assignment `yield` is the first yield in the `countdown()`. The `n=yield` statement is linked to the `c.send()` statement, so the variable `n` will receive a value from `c.send()`. We must first start the generator object before we can `send()`. We can use either `next(c)`, as in the preceding example, or `c.send(None)`. Only after that can we begin

the `countdown()` loop with `c.send(5)`. The `for i in c:` loop executes `i=next(c)` automatically.

Second, it is obvious that a `yield` suspends the execution of a task. In our example, the generator object `c` create by `c=countdown()` is waiting for a `next(c)` or a `c.send(None)` to resume execution before continuing.

To avoid errors, we must first use `next(c)` and then `c.send(5)`. This is the result of using `next(c)` instead of `c.send(5)`.

```
>>> c=countdown()
>>> next(c)
Commencing countdown
>>> next(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in countdown
TypeError: unsupported types for __ge__: 'NoneType', 'int'
```

Because `next(c)` sends no value, the `NoneType` error occurs. Following the `c.send(5)`, the `while n>=0` begins counting down for each `i` from `for i in c`. The `for i in c` statement executes `i=next(c)` automatically. It is critical to understand that a `yield` suspends the current execution and allows values to be passed in and out at the yield point. Consider the following example:

```
>>> def f():
...     s=0
...     while s!=6:
...         n,m=yield s
...         s=n+m
...         print(s)
...     yield "Bye!"
...
>>> c=f()
>>> print(c.send(None))
0
>>> print(c.send((1,2)))
3
3
>>> print(c.send((4,1)))
5
5
```

```
>>> print(c.send((3,3)))
6
Bye!
>>>
```

We can clearly send and receive data from a generator. Our goal in *workers* is not a *coroutine* type of concurrency. Our *workers* will communicate using shared memory. We will not use the generator to pass values. This will make `worker` implementation easier.

The generator is also used by the official MicroPython `uasyncio` to perform its magic. A quick `grep` reveals that `yield` is used in `event.py`, `funcs.py`, `lock.py`, and `stream.py`.

Multitasking

Returning to our original issue of running multiple looping tasks in the form:

```
while True: # loop a
  a()
while True: # loop b
  b()
while True: # loop c
  c()
while True: # loop d
  d()
```

Keep in mind that this is the type of program we want to run on a microcontroller. We now know how to implement those four tasks as generators so that they can all run concurrently while sharing CPU and RAM resources, thanks to our new understanding of the power of a generator that can suspend itself and read/write values by `yield` and can be resumed by `c.send()` or `next(c)` as long as we know the generator object `c`. Later on, we will convert a loop (*loop a*, for example) to a cooperative function called a *task*.

We will begin with sequentially running functions and convert them into tasks running concurrently in *workers*. We will create a benchmark from sequentially running functions to compare sequential and concurrent invocation. If our implementation is correct, we should

get the same results from these runs. Furthermore, it is reasonable to anticipate that the concurrent execution period will be longer than the sequential run. Why? Because we added some overhead to support concurrency. Concurrency will not speed up a process with only sequential functions. To make it run faster, we must add parallelism by running it on multiple CPUs.

Sequential Functions

We can run functions sequentially without having to worry about concurrency. Our preliminary sequential procedure is as follows:

```
from time import ticks_ms,ticks_diff,ticks_us

def f(p):
   global cnt
   # initialization phase
   N,a,b=p
   q=0
   # execution (work loop) phase
   while True:
      if a<b:
         print(N, "doing something")
         q+=a*b
         a+=1
      else:
         print(N, "done")
         break
   # finalization phase
   print(N, 'gives', q)
   cnt+=1

def main(pm):
   while cnt<4: next # four tasks
   print("All done")

cnt=0
a=b=c=d=f
t0=ticks_us()
a(('A',0,5))
b(('B',-3,0))
c(('C',5,10))
d(('D',-10,-5))
```

```
main(())
t1=ticks_us()
ran=ticks_diff(t1,t0)/1000000.0
print("Test run for", ran, "secs")
```

The four non-main functions are all the same. We used various call parameters to achieve various results. The `main()` function will complete last and act as a guard, which means that all other functions must complete before it is terminated. The following are the results of a sequential run with MicroPython on a Linux system:

```
A doing something
A doing something
A doing something
A doing something
A doing something
A done
A gives 50
B doing something
B doing something
B doing something
B done
B gives 0
C doing something
C doing something
C doing something
C doing something
C doing something
C done
C gives 350
D doing something
D doing something
D doing something
D doing something
D doing something
D done
D gives 200
All done
Test run for 0.000234 secs
```

The “done” time sequence is,

$$done_A < done_B < done_C < done_D < done_{main}$$

With concurrency enabled invocations, we expect to get,

1. 5 counts of “A doing something” and “A gives 50”
2. 3 counts of “B doing something” and “B gives 0”
3. 5 counts of “C doing something” and “C gives 350”
4. 5 counts of “D doing something” and “D gives 200”

in the later results. We also anticipate that the time periods will be slightly longer due to overhead. We’d like to know how much overhead we’ll incur as a result of concurrency.

Workers Multitasking

Let’s look at a typical *worker-based* program before we convert the sequential program example above into a multitasking program:

```
from time import ticks_ms,ticks_diff,ticks_us
from worker import task, MT

@task
def f(p):
   # initialization phase
   N,a,b=p
   c=yield
   # execution (work loop) phase
   while True:
      if a<b:
         print(N, "doing something")
         a+=1
         yield
      else:
         print(N, "done")
         break
   # finalization phase
   return

mt=MT(2)
mt.worker(f, ('A',0,2))
mt.worker(f, ('B',1,2))
mt.start()
```

The following is the result of the above codes:

```
A doing something
B doing something
A doing something
```

```
B done
A done
```

In our application, all *tasks* will have three basic `yield` points related to the `worker.MT` scheduler:

1. initialize - `yield`/`send`
2. execution loop - `yield None`/**`next`**
3. finalize - **`return`**

Let's add concurrency to our sequential program by using *worker* and the three guidelines mentioned above.

For the time being, let us assume that all four *tasks*: `a`, `b`, `c`, and `d` are independent and identical. Only the parameter values at call time will cause them to behave differently. In our program, we will use our `worker` module to manage these four *tasks* and one for the `main`. The resulting code is as follows for a cooperative multitasking application:

```
from time import ticks_ms,ticks_diff,ticks_us
from worker import task, MT

@task
def f(p):
   global cnt
   # initialization phase
   N,a,b=p
   q=0
   c=yield
   # execution (work loop) phase
   while True:
      if a<b:
         print(N, "doing something")
         q+=a*b
         a+=1
         yield
      else:
         print(N, "done")
         break
   # finalization phase
   print(N, 'gives', q)
   cnt+=1
   return
```

```
@task
def main(pm):
   c=yield
   while cnt<4: yield # four tasks
   print("All done")
   return

cnt=0
a=b=c=d=f
mt=MT(5)
mt.worker(a, ('A',0,5))
mt.worker(b, ('B',-3,0))
mt.worker(c, ('C',5,10))
mt.worker(d, ('D',-10,-5))
mt.worker(main, ())

t0=ticks_us()
mt.start()
t1=ticks_us()
ran=ticks_diff(t1,t0)/1000000.0
print("Test run for", ran, "secs")
```

Running this program from the command line with the MicroPython port for unix on a Linux PC will result in the following output:

```
$ upy test_worker.py
A doing something
B doing something
C doing something
D doing something
A doing something
B doing something
C doing something
D doing something
A doing something
B doing something
C doing something
D doing something
A doing something
B done
B gives 0
C doing something
D doing something
A doing something
C doing something
D doing something
```

```
A done
A gives 50
D done
D gives 200
C done
C gives 350
All done
Test run for 0.000755 secs
```

The `upy` is a short name for the soft-linked `micropython`. MicroPython (running on a Linux PC) must have an empty module path in its `sys.path` list in order to recognize a *frozen module*[10]. If we run the script from the command line, we must include `sys.path.insert(0, '')` in our program. The REPL, on the other hand, does this automatically.

```
import sys
sys.path.insert(0, '')
from worker import task, MT
```

Tasks are jobs for *workers*. Every *task* is a *generator* function. They must all contain two or more `yield` statements. To function properly, a *task* must have one `c=yield` in the *initialization phase* and at least one **`return`** statement as the last statement in the *finalization phase*. A **`return`**, also known as a *termination point*, ends a *worker* and removes it from the run queue.

All other `yield` statements in between `c=yield` and **`return`** must return `None`. These `yield` statements are the *cooperative* part of a *worker*, where execution is suspended and control is returned to the `worker.MT` scheduler, also known as a *swapout point*. All of these *workers* are competing with one another. The currently running worker must `yield` so that other workers can have their fair share of CPU time in order for the program to progress. The *swapout point* is an important part of the logic of *workers* programming.

As in the statement **`while`** `cnt<4:yield`, a `yield` associated with a *wait* is referred to as a *synchronization point*. In this case, the *synchronization point* is waiting for the condition `cnt>=4` to be met.

[10] There is no longer a need to include `''` in the `sys.path` for *frozen module* after MicroPython v1.18. That is handled by '.frozen'.

Multitasking processes with completely independent and unrelated tasks are uninteresting (aside from the possibility of parallelism). An asynchronous multitasking module requires *synchronization points* in order to be useful as a programming tool.

In our example, tasks `a`, `b`, `c`, and `d` ran independently, and no *synchronization point* was required, while the `main` task is timed to the event `cnt>=4`. The `main` task will pause execution at its `cnt<4` *synchronization point* until the *synchronization event* occurs.

Here are the four fundamental *yielding* concepts for *workers*:

1. swapout point
2. synchronization point
3. synchronization event
4. termination point

If we want to use *workers* successfully in our programs, we must first understand these four *yielding* types. They also apply to any *cooperative multitasking* system. Let's call them 1) *swapout*, 2) *sync-point*, 3) *sync-event*, and 4) *exit*.

Take note of how *workers* alternate "doing something". The `worker.MT` scheduler intersperses the "doing something" among the *workers*.

The following is the time sequence of "done" produced by the "workers",

$$done_B < done_A < done_D < done_C < done_{main}$$

The above order is determined by the parameter values and how the scheduler handles scheduling. The last *worker* in the run queue is moved to the slot vacated by the terminated *worker* in our `worker` module. When the `b` *worker* exits at **`return`**, the `main` *worker* (last task registered) is moved to the slot previously held by `b` (terminated *worker*).

If you care to count the results obtained,

1. 5 counts of "A doing something" and "A gives 50"
2. 3 counts of "B doing something" and "B gives 0"

3. 5 counts of “C doing something” and “C gives 350”
4. 5 counts of “D doing something” and “D gives 200”

were the same as the results from the sequential test run. The difference was that these *tasks* were being executed at the same time. Readers with keen eyes will notice that the `main` task wasted a large number of *CPU* cycles in its pooling for its *sync-event* of `cnt>=4`. The bulk of overhead time was spent at *sync-point*s.

We can’t expect the tasks to be completed in the order they were added to the *workers* list. This invariant is not a requirement for concurrency. Concurrent tasks are loosely coupled and can be executed in any order. A concurrent task is not affected by the relative speed of other concurrent tasks or the order in which they are executed. The order of execution is determined by how the tasks collaborate via *swapout*, *sync-events*, *workers* scheduling, and system interrupt handlers.

The above test script and its corresponding output demonstrated that all five *tasks* were running concurrently and cooperatively. The amount of CPU timesharing is proportional to the number of *swapout*s in each worker’s execution loop. A worker sits at its *sync-point*, waiting for a specific *sync-event* to occur. While the program was running, many *workers* were being added, scheduled, and purged.

We’ll take a closer look at the worker module and go over its implementation in detail in the following chapter, beginning on page 45.

Async/Await Coroutines

Can we achieve a similar result with `uasyncio`? Of course it’s a possibility. After all, `asyncio` is included in the MicroPython standard library. The following codes can be used to achieve a similar result:

```
from time import ticks_ms,ticks_diff,ticks_us
import uasyncio as asy

async def f(p):
   global cnt
   # initialization phase
```

```
    N,a,b=p
    q=0
    # execution (work loop) phase
    while True:
      if a<b:
        print(N, "doing something")
        q+=a*b
        a+=1
        await asy.sleep(0)
      else:
        print(N, "done")
        break
    # finalization phase
    print(N, 'gives', q)
    cnt+=1
    return  # MUST have a return,
          # so that the task can be deleted from run queue.

async def main():
    while cnt<4: # four tasks
       await asy.sleep(0.01)
    print("All done")
    return  # MUST have a return

cnt=0
a=b=c=d=f
loop = asy.get_event_loop()
loop.create_task(a(('A',0,5)))
loop.create_task(b(('B',-3,0)))
loop.create_task(c(('C',5,10)))
loop.create_task(d(('D',-10,-5)))
t0=ticks_us()
loop.run_until_complete(main())
t1=ticks_us()
ran=ticks_diff(t1,t0)/1000000.0
print("Test ran for", ran, "secs")
```

As you can see, the resulting output was very similar to multitasking with `worker`.

```
$ upy test_async.py
A doing something
B doing something
C doing something
D doing something
A doing something
```

```
B doing something
C doing something
D doing something
A doing something
B doing something
C doing something
D doing something
A doing something
B done
B gives 0
C doing something
D doing something
A doing something
C doing something
D doing something
A done
A gives 50
C done
C gives 350
D done
D gives 200
All done
Test ran for 0.002239 secs
```

You may notice some unusual constructs in the codes. An async task must call `await uasyncio.sleep(0)` in order to *swapout*. `uasyncio.sleep()` is a sleep function that is *non-blocking*. The running task will yield to the *event loop* at `await`. The *event loop* runs the next task in its execution queue. Reschedule the suspended task once the sleep period is over.

The statement `uasyncio.sleep(0.001)` in `main()` means to suspend execution and resume `main()` as soon as possible after 0.001 seconds, while other active tasks continue to run. Repeat until all `f` tasks are completed.

A **`return`** statement is required for an async task. The completed task will not be removed from the run queue unless a **`return`** statement is used. The global `cnt` variable assists `main()` in tracking the termination of `f` tasks. In this case, `main()` will only return if `cnt` is four.

There is one important caveat. In a cooperative multitasking concurrency, all tasks must be *non-blocking*. A program that uses a coopera-

tive scheduler relies on task swapping within its code. Any obstructive condition will cause the system to hang.

Multithreading Multitasking

Can we have similar multitasking program that uses `_thread`? Yes, we can. Provided that the MicroPython firmware we are using supports multithreading, i.e. time-multiplex multitasking with preemptive scheduling.

One possible way to do it is shown below:

```
import _thread as th
from time import ticks_ms,ticks_diff,ticks_us, sleep

def f(N,a,b):
   global cnt
   # initialization phase
   q=0
   # execution (work loop) phase
   while True:
     if a<b:
       with plock: print(N, "doing something")
       q+=a*b
       a+=1
       sleep(0.001)
     else:
       with plock: print(N, "done")
       break
   # finalization phase
   with plock: print(N, 'gives', q)
  with clock: cnt+=1 # lock.acquire();cnt+=1;lock.release()

def main():
   while cnt<4: # four tasks
      sleep(0.001)
   with plock: print("All done")

cnt=0
clock=th.allocate_lock()
plock=th.allocate_lock()

a=b=c=d=f
t0=ticks_us()
```

```
th.start_new_thread(a, ('A',0,5))
th.start_new_thread(b, ('B',-3,0))
th.start_new_thread(c, ('C',5,10))
th.start_new_thread(d, ('D',-10,-5))
main()
t1=ticks_us()
ran=ticks_diff(t1,t0)/1000000.0
print("Test ran for", ran, "secs")
```

We obtained similar results as the other two methods discussed above, but the execution order was slightly different.

```
$ upy test_thread.py
D doing something
B doing something
C doing something
A doing something
D doing something
B doing something
C doing something
A doing something
D doing something
B doing something
C doing something
A doing something
D doing something
B done
B gives 0
A doing something
C doing something
D doing something
A doing something
C doing something
D done
D gives 200
A done
A gives 50
C done
C gives 350
All done
Test ran for 0.006682 secs
```

The order in which tasks are executed is determined by the threading library used to implement the `_thread` module. The code contains no `yield` or `await` statements. The `_thread` employs preemptive scheduling, which means that the programmer has no control over when and

where a task is suspended or resumed.

Due to this preemption, we need to protect critical regions in our codes using locks. We make use of two locks:

1. `clock`, to safeguard the updating of global variable `cnt`,
2. `plock`, to protect serial channel during a print statement. The output from **`print`**`()` statements will interleave if `plock` is not used.

Even if one of the tasks is blocked, a thread-based multitasking system will continue to function. It will be preemptively suspended after a predetermined time period. Other tasks will continue to function normally. The underlying `_thread` library, not the application program, controls the allocation of CPU time slices.

Event/Callback Multitasking

So far, our thought process has been linear, moving from the past to the future; if this, then that. The event/callback prompts us to consider the alternative; if that occurs, then this. There is no pause or blocking wait in our logic flow. Everything that is possible will occur at any point in time.

Our brains, on the other hand, like to finalize things.

```
v=get_data(io)
w=comp(v)
```

What we normally expect is the value `v` is finalized by `get_data(io)` when we call `comp(v)` to get value `w`. In *event/callback* programming system (e.g. node.js) this assumption lead to many problems. The `comp(v)` will not wait for the `get_data(io)` to finish. The correct way is to use callback. In the case above we use `get_data(io, comp)`, where the `comp` is the callback function to `get_data`.

We normally expect `get data(io)` to finalize the value `v` when we call `comp(v)` to get value `w`. This assumption leads to a slew of issues in *event/callback* programming systems (for example, *javascript*). The `comp(v)` will not wait for `get data(io)` to complete. The proper

method is to use a callback. In the preceding example, we use `get data(io, comp)`, where `comp` is the callback function to `get data`.

This is well and good, but many tasks are inherently sequential. Simple tasks like cooking, eating, washing dishes, and relaxing can become a complex chain of callbacks with the *event/callback* system.

```
cook(ingredinces, eat(food, wash(dishes, relax)))
```

Using sequential tasks makes it much easier to understand.

```
food=cook(ingredinces)
dishes=eat(food)
wash(dishes)
relax(tv)
```

Our programming codes are linear-time instructions. The asyncrous effects of sequential codes are hidden by the *event/callback* programming framework. When an object, such as a network socket, listens to multiple events, the programs become more difficult to understand. Consider `socket('disconnect', callback)`. We need a way to determine the cause of this 'disconnect', which could be due to client-side `socket.close()`, server-side `socket.close()`, timeout, or network outage. Each callback is associated with a specific event. A disconnected error in blocking sockets is related to the action we want to take with the socket. We will not, for example, attempt to read or write from a socket after calling `socket.close()`.

Multitasking is well supported by the *event/callback* framework. In event-oriented programming, there is no *main loop*. Each callback is, in fact, an independent task that is ready to take action when the event associated with it occurs. The framework's *main-event loop* and *task scheduler* control the *event/callback* actions, which are not accessible to programmers.

Speed and Efficiency Comparison

We are performing a simple speed comparison of the execution time periods[11] of the four test scripts running on a Linux PC. The fastest time was when we ran the tasks sequentially, without concurrency.

Python2 was included in the comparison. Python2 was decommissioned on January 1, 2020. Python2 is paving the way for the Python language to gain popularity. There are still systems using Python 2. Python3 is more sophisticated and feature-rich than Python2. Python3 is more powerful than Python2. Because Python2 is simpler, it can implement some language constructs more efficiently.

We cannot guarantee that the calculated time periods are accurate because they are based on different time resolutions. Table 1 shows the outcome, for what it's worth.

Table 1: Speed comparison.

python/period	thread	asyncio	worker	seq
MicroPython	0.006682 sec	0.002239 sec	0.000755 sec	0.000234 sec
Python3	0.006691 sec	0.001934 sec	0.001646 sec	0.000269 sec
Python2	0.006662 sec		0.000318 sec	0.000213 sec

Let's use a simple formula to define overhead,

$$CPU_{overhead} = CPU_{con} - CPU_{seq}$$

The time spent running each concurrency model less the time spent running the sequential program is the amount of CPU time used to implement concurrency (i.e., the overhead time). The overhead for MicroPython, for example, is `0.000755-0.000234=0.000521`. Overheads for three different Pythons and three concurrency models[12] are shown in Table 2.

[11] We use `time.clock()` for Python2, `time.time_ns` for Python3, and `ticks_ms` and `ticks_diff` for MicroPython.

[12] The `asyncio` is not supported by Python2. For Python2, we must slightly mod-

Table 2: Efficiency.

python/overhead	thread	asyncio	worker
MicroPython	0.006448 sec	0.002005 sec	0.000521 sec
Python3	0.006422 sec	0.001665 sec	0.001377 sec
Python2	0.006449 sec		0.000105 sec

These numbers are merely indicative. These tests were carried out on a Linux PC. Linux is a computer operating system that supports multitasking and multiuser. It is part of the Unix family. The results obtained are dependent on how the underlying operating system scheduled our test scripts in relation to other currently active processes. The numbers obtained can differ between invocations.

In terms of efficiency and speed, the `worker` module is comparable to `_thread` and `asyncio`. Concurrency models provide us with programming tools for solving more complex problems that sequential programming cannot solve easily (if at all). Direct comparisons are thus limited to stating that 'concurrency models incurred overhead'. We saw that the significant proportion of the overhead time was spent waiting for the *sync-event* to occur at the *sync-point*. An external (real-world) event can be a *sync-event*.

MultiCore

Not all microcontroller ports support the `_thread` module. `_thread` is implemented differently in different ports. The *Raspberry Pi Pico RP2040* board's `_thread` module only supports two threads for its two cores. The two threads run on real physical cores and do not use time-multiplex preemptive scheduling. Instead, I believe the module should be called `_core`.

If you try to run the test code on page 37 on an RP2040, you will see

ify `worker.py`. Generators are implemented more efficiently in Python2 than in Python3.

the following error:

```
Traceback (most recent call last):A
doing something
  File "<stdin>", line 6, in <module>
  File "<stdin>", line 3, in main
OSError: core1 in use
>>> A doing something
A doing something
A doing something
A doing something
A all done
A gives 50
```

What caused this to occur? This is due to the fact that we only have two cores to run threads on. The REPL is using the first core and is currently executing `main()`. The second core will be assigned to `a()` by `th.start_new_thread(a,('A',0,5)`. The following `th.start_new_thread(b,('B',-3,0))` statement, will fail with `"OSError: core1 is in use"` error, and the `main()` function is aborted as a result, while the active task `a()` will continue to run to completion.

The *rp2* port, on the other hand, provides us with an intriguing opportunity to run two independent cooperative schedulers of workers on two different cores. This possibility will be explored further later, beginning on page 131.

Appendix 1

Source code for worker_lite.py

```
/*
 * The MIT License (MIT)
 *
 * Copyright (c) Sharil Tumin
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

from gc import enable, collect
from utime import ticks_ms, ticks_diff
def task(m,g,p):return m.worker(g(p))
class MT:
   def __init__(my,s=20):
      my.A=[b'0000']*s;my.size=s;my.tail=-1
      my.M={};my.x=''
      enable()
```

```
def worker(my,g):
   if my.tail+1<my.size:
      next(g)
      my.tail+=1;my.A[my.tail]=g
      return True
   else:
      return False
def mbox(my,w,t=0):
   my.M[w]=None
   if t==0:
      def d():
         return my.M[w]==None
   else:
      n=ticks_ms()
      def d():
          return my.M[w]==None and ticks_diff(ticks_ms(),n)<t
   return d
def get(my,w,k=False):
   v=my.M[w] if w in my.M else None
   if not k: my.M[w]=None
   return v
def put(my,w,v):
   if w in my.M: my.M[w]=v
def delay(my,t=0):
   n=ticks_ms()
   def d():
       return ticks_diff(ticks_ms(),n)<t
   return d
def start(my):
   i=0
   while True:
      C=my.A[i]
      try:
         w=C.send(my)
      except StopIteration:
         w=True # Done
      except Exception as x:
         w=False # Error
         my.x=str(C).split()[2]+' '+str(x)
      if w!=None: # worker done
         C.close();del C;collect()
         my.A[i]=my.A[my.tail];my.A[my.tail]=b'0000'
         my.tail-=1
         if my.tail<0:
            return
      i+=1
      if i>my.tail:i=0
```

Source code for worker.py

```
/*
 *
 * The MIT License (MIT)
 *
 * Copyright (c) Sharil Tumin
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

import gc
import utime
def task(g):
   def job(pm):
       return g(pm)
   return job
class MT:
   def __init__(my, n=20):
      my.s = WS(n); my.x=''
   def worker(my, f, p):
      if repr(f).find('closure')>=0:
         if K.tail<K.size:
            try:
               g=f(p)
               next(g)
            except Exception as x:
               my.x=str(x)
            else:
               K.A[K.tail]=g; K.tail+=1
               return K.tail
```

```
      # all else
      return 0
   def purge(my):
      for i in range(0,K.tail): K.A[i]=nop
      K.tail=0; gc.collect()
   def start(my):
      i=0
      while K.tail>0:
         C=K.A[i]
         try:
            w=C.send(my.s)
         except StopIteration:
            w=True # Done
         except Exception as x:
            w=False # Error
            my.x=str(C).split()[2]+' '+str(x)
         if w!=None:
            C.close();del C
            #gc.collect()
            K.tail-=1
            if i==K.tail: i=0
            else: K.A[i]=K.A[K.tail]
            K.A[K.tail]=nop
         else:
            i+=1
         if i>=K.tail:i=0
      return
   def log(my):
      return my.x
class K:
   A=[];tail=0;size=0;M={};L={}
def nop():pass
class WS:
   class V: pass
   def __init__(my,s):
      K.A=[nop]*s; K.size=s
      my.v=WS.V()
      gc.enable()
   def lock(my,l,w=''):
      if l in K.L: return False
      else: K.L[l]=w; return True
   def unlock(my,l,w=''):
      if l in K.L and w==K.L[l]: K.L.pop(l); return True
      else: return False
   def mbox(my,w,t=0):
      K.M[w]=None
      if t==0:
```

```
      def d():
         return K.M[w]==None
   else:
      n=utime.ticks_ms()
      def d():
         return K.M[w]==None and utime.ticks_diff(utime.ticks_ms(),n)<t
   return d
def get(my,w,k=False):
   v=K.M[w] if w in K.M else None
   if not k: K.M[w]=None
   return v
def put(my,w,v):
   if w in K.M: K.M[w]=v
def delay(my,t=0):
   if t<=0:
      def d(): return False
   else:
      n=utime.ticks_ms()
      def d():
          return utime.ticks_diff(utime.ticks_ms(),n)<t
   return d
```

Source code for pipe.py

```
/*
 *
 * The MIT License (MIT)
 *
 * Copyright (c) Sharil Tumin
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

class Pipe():
    def __init__(my, s=100):
       my.a=bytearray([0]*s);my.s=s
       my.h=my.t=my.n=0
    def put(my, w):
       if my.n>=my.s:
          return 0
       else:
          m=my.n
          for v in list(w):
             my.a[my.t]=v;my.t+=1;my.n+=1
             if my.t==my.s:my.t=0
             if my.n>=my.s or my.t==my.h: return my.n-m
          return my.n-m
    def get(my, c=0):
       if my.n==0 or c<0:
          return ''
       else:
          if c>my.n or c==0: c=my.n
          if my.h+c<=my.s:
```

```
            w=my.a[my.h:my.h+c]
            my.h=(my.h+c)%my.s
        else:
            w=my.a[my.h:]+my.a[0:c-(my.s-my.h)]
            my.h=c-(my.s-my.h)
        my.n-=c
        if my.n==0:
            my.t=my.h
        return w
```

Source code for worker.py for Python3

```
/*
 *
 * The MIT License (MIT)
 *
 * Copyright (c) Sharil Tumin
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

import gc
from datetime import datetime
def task(g):
   def job(pm):
       return g(pm)
   return job
class MT:
   def __init__(my, n=20):
      my.s = WS(n); my.x=''
   def worker(my, f, p):
      if repr(f).find('task')>=0:
         if K.tail<K.size:
            try:
               g=f(p)
               next(g)
            except Exception as x:
               my.x=str(x)
            else:
               K.A[K.tail]=g; K.tail+=1
               return K.tail
```

```
      # all else
      return 0
   def purge(my):
      for i in range(0,K.tail): K.A[i]=nop
      K.tail=0; gc.collect()
   def start(my):
      i=0
      while K.tail>0:
         C=K.A[i]
         try:
            w=C.send(my.s)
         except StopIteration:
            w=True # Done
         except Exception as x:
            w=False # Error
            my.x=str(C).split()[2]+' '+str(x)
         if w!=None:
            C.close();del C
            #gc.collect()
            K.tail-=1
            if i==K.tail: i=0
            else: K.A[i]=K.A[K.tail]
            K.A[K.tail]=nop
         else:
            i+=1
         if i>=K.tail:i=0
      return
   def log(my):
      return my.x
class K:
   A=[];tail=0;size=0;W={};M={};L={}
def nop():pass
class WS:
   class V: pass
   def __init__(my,s):
      K.A=[nop]*s; K.size=s
      my.v=WS.V()
      gc.enable()
   def lock(my,l,w=''):
      if l in K.L: return False
      else: K.L[l]=w; return True
   def unlock(my,l,w=''):
      if l in K.L and w==K.L[l]: K.L.pop(l); return True
      else: return False
   def mbox(my,w,t=0):
      K.M[w]=None
      if t==0:
```

```
      def d():
         return K.M[w]==None
   else:
      n=datetime.now()
      def d():
         return K.M[w]==None and (datetime.now()-n).total_seconds()<t
   return d
def get(my,w,k=False):
   v=K.M[w] if w in K.M else ''
   if not k and v!='': K.M[w]=None
   return v
def put(my,w,v):
   if w in K.M: K.M[w]=v
def delay(my,t=0):
   if t<=0:
      def d(): return False
   else:
      n=datetime.now()
      def d():
          return (datetime.now()-n).total_seconds()<t
   return d
```

Copying and pasting directly from a *pdf* file is probably not going to work. You will lose all your page identations. To copy Python code from a *pdf*, open the file with **okular**. Once the file is loaded, you can then export it as 'Plain Text'. Go to 'File' → 'Export As' → 'Plain Text'. You can then copy and paste from the text file with minimal corrections.

Appendix 2

Scripts code listing

1. `worker_first_example.py` on page 22.
   A simple example script demonstrating how to program *worker*s.
2. `abcd_tasks_seq.py` on page 27.
   An example of sequential execution of four tasks.
3. `worker_second_example.py` on page 29.
   An example of a typical script that makes use of *worker*s.
4. `abcd_tasks_worker.py` on page 30.
   A four-task concurrent execution example using cooperative multitasking with *workers*.
5. `abcd_tasks_async.py` on page 34.
   A four-tasks concurrent execution example using `uasyncio` module.
6. `abcd_tasks_thread.py` on page 37.
   A four-tasks multithreading execution example using preemptive multitasking with `_thread` module.
7. `worker_lite.py` on page 46.
   A simplified version of *workers-framework* designed for microcontrollers with limited run-time memory.
8. `deco_example_1.py` on page 51.
   A decorator example in the `deco(fun)(parm)` form.
9. `deco_example_2.py` on page 52.
   A decorator example in the `deco(parm)(fun)` form.

10. `simple_worker_decorator.py` on page 53.
    An example of how decorator is elegantly used to implement simple *worker*s.
11. `test_namespaces.py` on page 57.
    A script that calculates the overhead for various types of namespace variable reference.
12. `KS.py` on page 60.
    A test module to be used in **`import`** experiments containing `@property` for testing `setter` and `deleter` methods set to 'READ ONLY'.
13. `test_KS_a.py` on page 64.
    A test module for demonstrating the purpose of `sys.modules` and testing reimport in a function.
14. `test_KS_b.py` on page 66.
    A script that tests reimport in a function after deleting a module from `sys.modules` using **`del`**.
15. `test_import_func.py` on page 68.
    A script that demonstrates the use of `__import__()` to import modules.
16. `one_liner_import.py` on page 69.
    A script that demonstrates the use of `__import__()` as one-liner import statement.
17. `worker.py` on page 70.
    The complete *workers-framework* module source code, complete with line numbering
18. `test_lock.py` on page 80.
    A simple *critical region* test employing the *worker* `s.lock()` method.
19. `test_put_get.py` on page 82.
    Show usage of `s.mbox()` with `s.put()` (*producer*) and `s.get()` (*consumer*).
20. `test_wait_signal.py` on page 84.
    Simple example of `s.wait()` and `s.signal()` for events management.

21. `test_delay.py` on page 86.
    Simple example of `s.delay()` for non-blocking wait.
22. `test_pipe_1.py` on page 90.
    A simple script that demonstrates a *produser* writes one byte at a time to a *ring buffer* with a one-byte buffer size, and a *consumer* reads from it.
23. `test_lock_deadlock.py` on page 105.
    Simulate an incorrect use of `s.lock()` that results in deadlock.
24. `test_put_get_deadlock.py` on page 108.
    Simulate an incorrect use of `s.get()` and `s.put()` on `s.mbox()` that will cause deadlock.
25. `test_wait_signal_deadlock.py` on page 111.
    Simulate how nested `s.wait()` and `s.signal()` will cause deadlocka.
26. `test_wait_signal_delay.py` on page 116.
    Demonstration of well-behaved operations by `s.wait()` for multiple `s.signal()` event values.
27. `test_wait_signal_delay.py` on page 119.
    Using `s.val()` to provide a simple solution to multiple events on a single wait.
28. `test_pipe_2.py` on page 123.
    A script used to map speed versus buffer size in `pipe.py`. Table 8 shows the results.
29. `test_wait_signal_rp2.py` on page 132.
    Test script for running two *worker*s on two different cores of *RP240* and protecting critical regions with locks. The script shows how parallelism and cooperative-based concurrency can be combined.
30. `dining_philosophers.py` on page 140.
    Classic resource management problem introduced by Edsger Dijkstra. Simulate cooperative resource sharing problems to avoid deadlock and starvation.
31. `led_pulse.py` on page 145.
    LED blinking controller using two *worker*s on the *BBC micro:bit*'s *5x5 display*. Precise timing was difficult to achieve through soft-

ware control mechanisms.

32. `firefly.py` on page 146.
    Each LED of the *5x5 display* on a *BBC micro:bit* board represents a firefly. Each firefly is executed as a *worker*, where the intensity and duration of a flash are controlled.
33. `blink_rp2040.py` on page 149.
    Control the blink rate of the Raspberry Pi Pico board's onboard *LED* with *REPL* keyboard input.
34. `websoc_srv.py` on page 151.
    Websocket server.
35. `websoc_cln.py` on page 154.
    Websocket client.
36. `wifi.py` on page 156.
    WIFI utility script that saves and reads credentials from a file.
37. `websoc_echo_srv.py` on page 159.
    Websocket echo server for testing browser websocket connectivity.
38. `happy_eyeballs.py` om page 161.
    Happy eyeballs is a program that finds the fastest servers in a list of servers returned by DNS.
39. `wrepl.py` on page 163.
    Machine-to-machine (M2M) interaction through UART writing to `sys.stdout` and reading from `sys.stdin`.
40. `ble_srv_tools.py` on page 170.
    BLE server utility module for Bluetooth Low Energy beacon server.
41. `msg_beacon.py` on page 172.
    BLE message beacon server.
42. `ble_cln_tools.py` on page 174.
    BLE client utility module for Bluetooth Low Energy scanner.
43. `msg_scanner.py` on page 176.
    BLE message scanning client.
44. `auto_net.py` on page 179.
    Automatic network discovery and recovery using UDP broadcast.

45. `mpy_va.py` on page 206.
    A utility script for determining the correct `"-march="` option to `mpy-cross` of a board.

References

1. Dijkstra, E. W. (1968). *Co-operating sequential processes. In Programming languages*: NATO Advanced Study Institute: lectures given at a three weeks Summer School held in Villard-le-Lans, 1966 / ed. by F. Genuys (pp. 43-112). Academic Press Inc..

2. C. A. R. Hoare (2015) *Communicating Sequential Processes*. This document is an electronic version, first published in 1985 by Prentice Hall International.

3. M. Ben-Ari (1948). *Principle of concurrent programming*. Prentice Hall International. ISBN 0-13-701078-8

4. Michael Wooldridge (2002). *An Introduction to MultiAgent Systems*. Wiley. ISBN 0-471-49691-X

5. Ali E. Abdallah, Cliff B. Jones, Jeff W. Sanders: *Communicating Sequential Processes The First 25 Years*. Springer. Symposium on the Occasion of 25 Years of CSP London, UK, July 7-8, 2004. ISSN 0302-9743

Github

This book's GitHub repository can be found at

https://github.com/shariltumin/workers-framework-micropython

Credits

1. Cover page graphic designed by Macrovector/Freepik
   https://www.freepik.com
2. Introduction title page 9 graphic designed by Macrovector/Freepik
   https://www.freepik.com
3. Implimentation title page 45 graphic designed by Macrovector/Freepik
   https://www.freepik.com
4. Testing title page 99 graphic designed by Macrovector/Freepik
   https://www.freepik.com
5. Use-cases title page 139 graphic designed by Macrovector/Freepik
   https://www.freepik.com
6. Parting Remark title page 187 graphic designed by Macrovector/Freepik
   https://www.freepik.com

Be kind, do good.

SharilTumin © Bergen,2022