eventio_tour

December 1, 2018

1 EventIO Tutorial

The need to perform several actions concurrently arises frequently in microcontroller applications. Sensor outputs need to be read, motors and LEDs controlled, and buttons or touchpads checked for user input.

While it is possible in principle to perform all these actions sequentially in a loop, except in simple applications such solutions are difficult to get correct and even harder to maintain when additional capabilities are introduced.

This tutorial introduces `eventio`, a *partial* implementation of `curio`, a library for concurrent processing.

1.1 Blinking one LED

Let's start the tour with "blink", the proverbial "Hello World" application for microcontrollers (change the name of the LEDs to match your board).

```
In [2]: from board import LED_R, LED_G, LED_B
        import digitalio
        import time

        def blink(color, pin, period):
            p = digitalio.DigitalInOut(pin)
            p.direction = digitalio.Direction.OUTPUT
            for i in range(10):
                time.sleep(period/2)
                p.value = not p.value
            p.deinit()

        blink("red", LED_R, 0.5)
```

The code blinks the LED the specified number of times and then stops. Nothing unusual.

1.2 Blinking multiple LEDs simulataneously

Now suppose we wanted to blink several LEDs at individual rates, say 0.7ms and 0.4ms periods. Simply calling calling `blink` twice with different parameters won't produce the desired effect:

```
In [3]: blink("red",   LED_R, 0.7)
        blink("green", LED_G, 0.4)
```

This blinks the red light followed by the green light, one after the other. To blink them simultaneously but at different rates, modify `blink` as follows:

```
In [8]: import eventio

        async def blink(color, pin, period):
            p = digitalio.DigitalInOut(pin)
            p.direction = digitalio.Direction.OUTPUT
            for i in range(10):
                await eventio.sleep(period/2)
                p.value = not p.value
            p.deinit()
```

The differences are importing `eventio` and peceeding the function definition with `async`. This marks it as a `coroutine`, a piece of code that can run concurrently with other operations.

The call to `time.sleep` is replaced by `await eventio.sleep`. The keyword `await` tells the Python interpreter to perform other activity, turing on and off other LEDs in this case, simultaneously.

The following code blinkes the red, green, and blue LED simultaneously:

```
In [11]: async def main():
             r = await eventio.spawn(blink, "red  ", LED_R, 0.7)
             g = await eventio.spawn(blink, "green", LED_G, 0.4)
             b = await eventio.spawn(blink, "blue ", LED_B, 0.5)

         eventio.run(main)
```

The calls to `eventio.spawn` create three separate coroutines which all run in parallel. The call to `eventio.run` starts the process.

In addition to the visual confirmation, we can add printed output to convince us that the coroutines indeed run concurrently:

```
In [12]: %softreset

         from timer import Chronometer
         from board import LED_R, LED_G, LED_B
         import eventio
         import digitalio

         chrono = Chronometer()

         async def blink(color, pin, period):
             global chrono
             p = digitalio.DigitalInOut(pin)
             p.direction = digitalio.Direction.OUTPUT
             last_time = 0
             for i in range(10):
                 await eventio.sleep(period/2)
                 p.value = not p.value
```

```
                elapsed = chrono.elapsed_time
                delta = elapsed-last_time
                extra_delay = delta - period/2
                print("{} at {:6.0f} ms, {:4.0f} ms since last call, {:6.0f} ms extra".format
                    color, 1000*elapsed, 1000*delta, 1000*extra_delay))
                last_time = chrono.elapsed_time
            p.deinit()
            print("{} is done".format(color))

        async def main():
            r = await eventio.spawn(blink, "red  ", LED_R, 0.3)
            g = await eventio.spawn(blink, "green", LED_G, 0.7)
            b = await eventio.spawn(blink, "blue ", LED_B, 0.5)
            print("all LEDs blinking ...")
            await g.join()   # wait for coroutine g to terminate
            # print some statistics
            kernel = await eventio.get_kernel()
            print("Program ran for {} seconds with {} % CPU utilization".format(
                kernel.uptime(), kernel.load_average()
            ))

        eventio.run(main)
```

```
all LEDs blinking ...
red   at    155 ms,  155 ms since last call,      5 ms extra
blue  at    260 ms,  260 ms since last call,     10 ms extra
red   at    307 ms,  151 ms since last call,      1 ms extra
green at    357 ms,  357 ms since last call,      7 ms extra
red   at    459 ms,  151 ms since last call,      0 ms extra
blue  at    511 ms,  250 ms since last call,     -0 ms extra
red   at    611 ms,  151 ms since last call,      0 ms extra
green at    709 ms,  351 ms since last call,      0 ms extra
red   at    763 ms,  151 ms since last call,      1 ms extra
blue  at    765 ms,  252 ms since last call,      2 ms extra
red   at    915 ms,  151 ms since last call,      1 ms extra
blue  at   1018 ms,  251 ms since last call,      0 ms extra
green at   1060 ms,  350 ms since last call,     -0 ms extra
red   at   1067 ms,  151 ms since last call,      1 ms extra
red   at   1219 ms,  151 ms since last call,      1 ms extra
blue  at   1270 ms,  251 ms since last call,      0 ms extra
red   at   1371 ms,  151 ms since last call,      1 ms extra
green at   1412 ms,  350 ms since last call,     -0 ms extra
blue  at   1522 ms,  251 ms since last call,      0 ms extra
red   at   1524 ms,  152 ms since last call,      2 ms extra
red   is done
green at   1765 ms,  351 ms since last call,      0 ms extra
blue  at   1774 ms,  251 ms since last call,      1 ms extra
blue  at   2026 ms,  251 ms since last call,      1 ms extra
```

```
green at   2117 ms,  351 ms since last call,      1 ms extra
blue  at   2277 ms,  250 ms since last call,      0 ms extra
green at   2469 ms,  351 ms since last call,      1 ms extra
blue  at   2530 ms,  251 ms since last call,      0 ms extra
blue  is done
green at   2821 ms,  351 ms since last call,      1 ms extra
green at   3173 ms,  351 ms since last call,      1 ms extra
green at   3525 ms,  351 ms since last call,      1 ms extra
green is done
Program ran for 3.528 seconds with 3.089569 % CPU utilization
```

The printed output confirms that the coroutines indeed run simultataneously. Also shown is the time (in milli-seconds) at which the LED is turned on or off. Analyzing them you will notice that each LED is called at multiples of half it's period, plus a small "processing delay". Try to program all of this in a single loop ... And this is just for blinking LEDs; many real applications have more complex requirements!

Unlike calling `time.sleep`, which litterally instructs the processor to do `nothing` except dissipate power and heat up the planet, a call to `eventio.sleep` checks for other work to be done, which in this example amounts to checking if one of the `blink` coroutines is ready to run. If no coroutine is ready to execute, `eventio.sleep` powers the processor down, which reduces the supply current by more than three orders-of-magnitude.

Indeed, while the entire program ran for several seconds, the processor was in the high-power `awake` state less than 2 percent of the time, corresponding to a 50 times reduction of the energy used. Given the populatity of programs that blink LEDs, `eventio` has the potential of making a significant contribution towards saving the planet!

1.3 Input

In the current implementation the LEDs blink for a fixed number of times. The code below stops when button `MODE` (adapt to your board) is pressed.

```
In [13]: %softreset

         from timer import Chronometer
         from board import LED_R, LED_G, LED_B, MODE
         import eventio
         import digitalio

         chrono = Chronometer()

         sw1_event = eventio.PinEvent(MODE)

         async def blink(color, pin, period):
             global chrono
             p = digitalio.DigitalInOut(pin)
             p.direction = digitalio.Direction.OUTPUT
             try:
```

```
            while True:
                await eventio.sleep(period/2)
                p.value = not p.value
                chrono.reset()
        except eventio.CancelledError:
            print("{} cancelled".format(color))
            p.value = False   # LED off
            p.deinit()
            raise

    async def main():
        r = await eventio.spawn(blink, "red  ", LED_R, 0.3)
        g = await eventio.spawn(blink, "green", LED_G, 0.7)
        b = await eventio.spawn(blink, "blue ", LED_B, 0.5)
        print("All LEDs blinking ... press button to stop!")
        await sw1_event.wait()
        print("Button pressed! Cancelling blinkers ...")
        await r.cancel()
        await g.cancel()
        await b.cancel()
        # print some statistics
        kernel = await eventio.get_kernel()
        print("Program ran for {} seconds with {} % CPU utilization".format(
            kernel.uptime(), kernel.load_average()
        ))

    eventio.run(main)
```

```
All LEDs blinking ... press button to stop!
Button pressed! Cancelling blinkers ...
red   cancelled
green cancelled
blue  cancelled
Program ran for 5.186 seconds with 2.757424 % CPU utilization
```

1.4 Concurrency and Parallel Processing

Although it appears as if coroutines run in parallel, only one is active at any given time. This is similar to other computers (laptops, desktops, etc) which give the appearance of running several programs simultaneously but are in fact switching rapidly between different tasks (multi-core processors do run more than one program concurrently). If the processor alternates rapidly e.g. between displaying a movie and a spreadsheet, it appears as if both were running simultaneously, just as the different colored LEDs are blinking concurrently.

A big difference between the parallel processing of laptop and desktop computers and `eventio` is the approach taken to switching between tasks. The former use a technique known as **"preemptive multitasking"**. In this case, the operating system (e.g. Linux or Windows) uses a timer to rapidly (e.g. every 50ms) switch between tasks: The currently running program is temporarily

suspended, it's state saved, and a different task is permitted to execute. This process repeats, with each task getting a turn. If a task is current blocked, e.g. waiting for input, it is skipped until it is ready again.

With preemptive multitasking, tasks are not aware when execution is interrupted and have no control over when this happens. By contrast, `eventio` uses a different form of task switching called **"cooperative multitasking"**. In this scenario, task have full control over when they relinquish the CPU, namely when calling `await`. In other words, a coroutine can be assured that no other task will interrupt it until it signals its "consent" with `await`. The duration of the suspension depends on the statement called and the tasks that are waiting. E.g. `await eventio.sleep(1)` suspends the task for at least one second, but could also suspend it for longer if other tasks need the CPU.

Preemptive and cooperative multitasking have both advantages and drawbacks, and which solution is better depends on the application. For small microcontrollers with limited memory and processing speed, cooperative multitasking is attractive since it generally uses less memory. More significantly, however, is the fact that task switching happens only at well defined instances marked with the keyword `await`.

One of the most challenging aspect of parallel processing is ensuring proper use of shared resources. For example, imagine you want to read the value from a sensor over `I2C`. The driver first sends the address of the sensor to get its attention. Then asks the sensor send its value over the bus.

If this sequence is interrupted at an unfortunate moment, by another task, the program may not function as intended. For example, if after addressing the sensor a different task gets control and adresses a different sensor, once control returns to the first task the request to read the sensor data goes to the wrong sensor producing the wrong or no answer.

Such errors are difficult to detect, as they may occur only infrequently and are difficult to reproduce. With cooperative multitasking a task can be assured that it will not be interrupted unless calling `await`. While this simple solution does not address all situations, it substantially reduces the opportunities for errors in concurrent programs.

The main drawback of cooperative multitasking is that a (rouge) task can prevent all others from running. E.g. if in the above example, the call to `eventio.sleep` is replaced by `time.sleep` (and the `await` keyword removed), the first coroutine will run exclusively until it finishes. In a desktop environment this obviously would not be practical - a simple programming mistake could freeze the entire computer. On a microcontroller, however, the entire application is usually under control of a single user who can make sure that this does not happen.

1.5 Caveats

In addition to the situation where a coroutine fails to call `await`, either due to a programming error or maliciously, thus preventing other tasks to run, there are a few other errors to avoid.

The first, and most notrious (the kind where everything looks right but still refuses to "work") is forgetting the `await` keyword, as in the example below:

```
In [15]: %softreset

         from board import LED_R
         import digitalio
         import eventio

         async def blink(color, pin, period):
```

```
            p = digitalio.DigitalInOut(pin)
            p.direction = digitalio.Direction.OUTPUT
            for i in range(100):
                # ERROR: no await
                eventio.sleep(period/2)
                p.value = not p.value
            p.deinit()

        eventio.run(blink, "red", LED_R, 1)
        print("Done!")
```

```
Done!
```

This program runs very quickly without error, but `eventio.sleep(period/2)` does not "sleep", hence resulting in the LED blinking very rapidly (faster than discernible).

The result of `eventio.sleep(1)` is a generator (or coroutine in some versions of Python):

```
In [16]: print(eventio.sleep(1))
```

```
<generator object 'sleep' at 200081c0>
```

It is the `await` keyword that instructs the interpreter to actually perform the function. Unfortunately in Python the statements with and without `await` are syntactically correct, so watch out!

The other common error is calling `time.sleep` instead of `eventio.sleep`. The former, with `await`, gives an error and is thus easy to spot. But without `await`, `time.sleep` is of course a valid call, except that it blocks the CPU, preventing other tasks from running.

If a concurrent program does not function as expected, these are some of the things to check!