Threading bei der Erfassung von Messdaten

Jean-claude.feltes@education.lu

Problemstellung

In diesem Beispiel werden Daten über die serielle Schnittstelle eingelesen von einem Messgerät eingelesen (z.B. http://staff.ltam.lu/feljc/school/asserlogger/asserlogger_hardware.pdf)
und in bestimmten Zeitintervallen angezeigt. Das Programm soll auch auf Benutzereingaben reagieren (z.B “q” = Quit zum Verlassen des Programms, die anderen Eingaben sollen zur Mess-Schaltung geschickt werden).

Die seriellen Daten kommen als Tabellenzeilen an:

```
1   4.87   4.27   1.42   1.36   0.000
2   4.88   4.28   1.42   1.36   0.000
3   4.88   4.28   1.42   1.36   0.000
4   4.88   4.28   1.42   1.36   0.000
…
```

Am einfachsten werden sie mit der Funktion Serial.readline eingelesen, so dass man automatisch pro Einlesevorgang eine vollständige Zeile hat.
Diese Funktion blockiert allerdings, bis eine Zeile von Werten verfügbar ist.

Genauso wirkt die input-Funktion für die Tastatur blockierend, da sie nicht einzelne Zeichen einliest, sondern auf ein Betätigen der <Enter>-Taste wartet.

Diese beiden Vorgänge müssen also in separaten Threads laufen, damit das Hauptprogramm nicht blockiert wird.

Die Ausgabe der Daten könnte man im Hauptprogramm vornehmen, oder man spendiert auch für diese einen eigenen Thread, so dass das Programm sehr übersichtlich wird, und man Anpassungen leichter vornehmen kann.

Alle Threads sind als eigne Klassen aufgebaut, jede dieser Klassen hat eine Variable self.data für die Daten, eine Funktion self.startthread zum Starten des Threads und eine Funktion self.stopthread zum Stoppen des Threads.

Der Thread für die serielle Schnittstelle hat zusätzlich noch eine Funktion self.write_serial zum Ausgeben von Kommandos über die Schnittstelle.

So ergibt sich für das Hauptprogramm in einer Kommandozeilen-Variante folgendes:

```
s=Getseriallines("/dev/ttyACM0",115200)
s.startthread()

o=Outputthread(s, 1)
o.startthread()

k=Keyboardthread()
k.startthread()
```

```
    while True:

        if k.data == "q":              # Programm beenden
            break

        if k.data:
            s.write_serial(k.data)     # Andere Eingabe → seriell ausgeben
            k.data=""                  # Ausgewertete Daten löschen

        time.sleep(0.001)              # Zeit für andere Threads lassen

    k.stopthread()
    s.stopthread()
    o.stopthread()
```

Zunächst werden die drei Threads als Objekte initialisiert und gestartet.
Dann folgt eine Schleife, in der die Daten des Keyboard-Threads abfragt werden.
Bei “q” verlassen wir die Schleife, alle anderen Daten werden über die serielle Schnittstelle ausgegeben.
Zu beachten ist, dass k.data nach der Auswertung gelöscht wird. Im anderen Fall würde, wenn kein neuer Input erfolgt, ein Kommando wiederholt über die Schnittstelle ausgegeben werden.

Die time.sleep Zeile sorgt dafür, dass Zeit für die anderen Threads gelassen wird.

Die drei letzen Zeilen dienen zum Stoppen der Threads. Sie sind nicht unbedingt nötig da dies beim Verlassen des Hauptprogramms automatisch erfolgt.

Der Ouputthread

Dieser Thread ist der einfachste, da er nur Daten lesen und ausgeben soll:

```
class Outputthread():
    def __init__(self,producer, dt, **kwargs):
        self.stopevent=threading.Event()
        self.producer=producer
        self.data=producer.data
        self.dt=dt

    def stopthread(self):
        self.stopevent.set()


    def startthread(self):
        self.thread = threading.Thread( target=self.outputloop,
                        name="outputloop")
        self.thread.daemon = True
        self.stopevent.clear()
        self.thread.start()


    def outputloop(self):
        while True:
            print(self.producer.data)


            if self.stopevent.isSet():
```

```
                break

            time.sleep(self.dt)
```

Die Funktion __init__
Die Instanziierung o=Outputthread(s, 1) im Hauptprogramm ruft die __init__ Funktion auf.

Der __init__ Funktion werden die Parameter producer und dt übergeben.
Producer ist die Instanz des Daten produzierenden Threads, in unserem Fall des Threads Getseriallines, während dt das Zeitintervall zwischen zwei Outputs darstellt, in unserem Fall 1 Sekunde.

Es wird ein stopevent definiert, das ausgelöst werden kann um den Thread zu stoppen.

Die Funktion startthread
Diese Funktion wird im Hauptprogramm mit o.startthread() aufgerufen.
Mit

```
self.thread = threading.Thread( target=self.outputloop,
                  name="outputloop")
```

wird ein neuer Thread erzeugt.
Wichtig ist die Targetfunktion, dies ist die Funktion die im Thread ausgeführt wird.

Die Eigenschaft `.daemon = True` sorgt dafür, dass der Thread nur solange läuft wie das Hauptprogramm, und es keine vagabundierenden Threads gibt wenn das Programm unterbrochen wird. Diese Zeile ist je nach Python-Version eventuell überflüssig, sie ist aber zur Sicherheit vorhanden.

`stopevent.clear()` setzt das Stopevent zurück, so dass der Thread laufen kann.

Nicht vergessen darf man, den Thread mit `thread.start()` zu starten.

Die Funktion outputloop
Dies ist die Funktion, die als Target aufgerufen wird.
Sie besteht im wesentlichen aus einer Schleife, in der die Daten ausgegeben werden, mit einer Pause dt zwischen den einzelnen Durchläufen.

Statt die Daten nur zu printen, kann man sie hier natürlich auch anders darstellen, z.B. mit matplotlib als Diagramm.

Der Keyboard-thread

Dieser ist im Prinzip ganz ähnlich aufgebaut:

```
class Keyboardthread():

    def __init__(self, **kwargs):
        self.stopevent=threading.Event()
        self.data =""
        self.lock = threading.Lock()
```

```
    def stopthread(self):
        self.stopevent.set()
    def startthread(self):
        self.thread = threading.Thread(target=self.inputloop , name="inputloop")
        self.thread.daemon = True
        self.stopevent.clear()
        self.thread.start()


    def inputloop(self):
        while True:
            inp=input()
            self.lock.acquire()
            self.data=inp
            self.lock.release()

            if self.stopevent.isSet():
                break
```

Die Funktion __init__
Sie wird bei der Instanziierung aufgerufen, im Hauptprogramm durch `k=Keyboardthread().`
Hier werden keine Parameter übergeben.
Es wird ein Stopevent und eine Variable `self.data` deklariert.
Neu im Vergleich zum Outputthread ist ein Lock-Objekt.

Die Fnktionen stopthread und startthread
sind ähnlich aufgebaut wie beim Outputthread.

Die Funktion inputloop
wird im Thread ausgeführt. Sie enthält eine Schleife, in der mit `input()` die Tastatur abgefragt wird. Zur Erinnerung: Die input()-Funktion blockiert solange bis die <Enter>-Taste gedrückt wird. Die Eingabe wird dann in die Variable self.data geschrieben.
Dabei wird die Veränderung der self.data Variablen mit `lock.acquire` und `lock.release` „thread safe“ gemacht, um korrupte Daten zu vermeiden.
(Da mehrere Threads asynchron laufen, kann es zu Problemen kommen, wenn das Schreiben einer Variablen durch einen anderen Thread unterbrochen wird. Einge Operationen sind je nach Python-Version und Variablentyp „atomic“, sie können nicht unterbrochen werden und sind so auch ohne Lock „thread safe“, es ist aber sicherer, den Lock anzuwenden.)

Der serielle Thread

```
class Getseriallines(serial.Serial):

    def __init__(self, port, baudrate, **kwargs):
        serial.Serial.__init__(self)
        self.port=port
        self.baudrate = baudrate
        self.stopevent=threading.Event()
        self.data =""
        self.lastline=""
        self.lock = threading.Lock()

    def stopthread(self):
       self.stopevent.set()

    def startthread(self):
```

```
        self.serialthread = threading.Thread( target=self.serial_loop ,
                  name="_serial_loop")
        self.serialthread.daemon = True
        self.stopevent.clear()

        if self.isOpen()==False:
            self.open()

        self.serialthread.start()

    def serial_loop(self):
        while True:
            l=self.readline()
            line=l.decode('utf-8')
            self.lock.acquire()
            self.data += line
            self.lastline=line
            self.lock.release()

            if self.stopevent.isSet():
                break

        self.close()


    def write_serial(self,s):
        b=s.encode('utf-8')
        self.write(b)
```

Die Klasse Serialreadline basiert auf der Klasse Serial.serial, so dass sie sämtliche Funktionen und Variablen dieser Klasse erbt.

Die Funktion __init__
Sie wird bei der Instanziierung aufgerufen, im Hauptprogramm durch

```
s=Getseriallines("/dev/ttyACM0",115200)
```

Dabei werden serieller Port (hier ein virtueller serieller USB Port) und Baudrate gesetzt.

Die Funktionen stopthread und startthread
sind ähnlich aufgebaut wie bei den anderen Thread Klassen.
Bei starthread wird zusätzlich der Port geöffnet, wenn dies vorher nicht geschehen war.

Die Funktion serial_loop
enthält die Schleife zum Lesen der einkommenden Datenzeilen.
Diese werden mit readline() gelesen. Da readline() einen Bytestring liefert, muss dieser mit der decode-Funktion in Unicode umgewandelt werden, wenn Python3 benutzt wird.

Wie beim Keyboard-Thread wird ein Lock verwendet, um korrupte Daten zu vermeiden.

Die vollständigen Daten sthen in der Variablen self.data, während self.lastline nur die letzte empfangene Zeile enthält.

Die Funktion write_serial

dient dazu, einen String s über die serielle Schnittstelle auszugeben.
In Python 3 ist hier eine Umwandlung von Unicode nach Byte String nötig, dies geschieht mit der encode Funktion.

Literatur

Threading:

https://www.tutorialspoint.com/python3/python_multithreading.htm

https://stackoverflow.com/questions/2846653/how-to-use-threading-in-python

https://pymotw.com/2/threading/

Serial:
https://pyserial.readthedocs.io/en/latest/shortintro.html#eol

Python interface for Arduino:

https://playground.arduino.cc/interfacing/python

https://gitlab.com/william.belanger/bridge

Serial und numpy.csv.reader():

https://bytes.com/topic/python/answers/159541-py-serial-csv

https://www.e-education.psu.edu/geog485/node/141