Efficient COM Port Programming with Python: A Practical Guide

引言

在這篇文章中，我們將探討一些多工處理的概念，並將它用在雙向通訊的情境中。這些原理不僅適用於 Python 或這個雙向通訊的例子，實際上，它們可以應用於任何程式語言中與任何情境中。希望通過知曉這些原理，能夠設計出更好的程式架構。

目錄

主題討論

為實現資料的雙向傳輸，我們需以多工處理方式設計程式。其他範例皆以多線程的方式來實現多工處理。在開始這個主題之前，筆者想先指出：吳奇隆學長在他的 COM Port Programming with Python 中關於 Thread 套件的註解「可邊設定輸出時邊等待輸入數值」需謹慎看待，因為牽涉到 Python 特有的 GIL 問題，關於這點我們稍後也會討論。而至於楊永楠學長的 COM Port Programming with C# 和陳鍇瀚學長的 COM Port Programming with Visual BASIC，其對於 Thread 的使用說明可能會引起混淆。由於軟體阻塞和多工處理的概念確實不容易理解，我們首先討論第一個主題——阻塞。

阻塞

為何 Verilog 的 assignment 會區分 blocking 與 non-blocking？答案或許跟 C 語言考試經常出現的「快捷運算」有關。實際上，這都歸功於阻塞操作的影響。以開車做比喻，如果前車不走，後車就無法前進，這就是「阻塞」。

阻塞是序向邏輯電路與組合邏輯電路最明顯的差異之一，序向電路的輸出不只取決於當前輸入，也與前一次的狀態有關，因此需要區分先後順序。為了確保執行順序，就必須讓第二個操作必須等待第一個操作完成，這個「等待」的過程就是一種「阻塞」，序向電路通常使用脈波邊緣觸發來實現阻塞效果。

與硬體相反，軟體先天就是阻塞的。例如，當一行程式碼在執行完畢之前，程式不會跳到下一行，這種情況就源自於阻塞操作。因此，許多語言在邏輯運算中會使用「快捷運算」來改善執行速度（快捷運算表現出的行為與硬體邏輯閘不能說完全相似，只能說一模一樣）。

阻塞也可以視為一種「占用」，程式占用了 CPU 的線程，導致其他的程式需「等待」該程式釋放線程後才能執行。這正是楊永楠學長與陳鍇瀚學長提到的「使用 While 持續接收，會一直卡住」的原因。

多工

多工顧名思義就是處理多個工作，而

這種占用問題不僅在軟體中出現，無線通訊也因為共用介質（如空氣等）而引起占用問題。為了解決這類問題，無線通訊使用路復用作為解決方案，常見的方式包含以下四種：

1. 分時多工（TDM）：將時間切分成不同時段，輪流使用通訊媒介。
2. 分頻多工（FDM）：將頻率切分成不同頻段，同時使用通訊媒介。
3. 空間多工（SDM）：通過波束成形技術將訊號對準接收站發射。
4. 分碼多工（CDM）：在發送時添加代號，以便接收端識別特定數據。

回到我們的主題，我們不僅可以使用多線程實現多工處理，還能夠使用**分時多工**的方式！從微算機的角度來看，分時多工被稱為「半雙工」，而同時多工則自然就是「全雙工」。那麼要如何實現這個半雙工的模式呢？與其說出來不如看一下 Python 範例。

```python
from serial import *                    # 串列傳輸套件

ser = Serial('COM3', baudrate=9600) # 創建序列傳輸物件

while True:
    data = ser.readline()           # 讀取一行
    print(data.decode())            # 顯示資料

    ser.write(data)                 # 傳送資料
    ser.flush()                     # 清除緩衝區

```

（半雙工示例）

可以看出，我們透過分時多工的方法，成功避免了楊永楠學長與陳鍇瀚學長所遇到的卡住問題。不同嗎？確實如此，他們卡住的是 GUI 介面而非資料傳送。雖然不知道 C# 以及 Visual BASIC 能否將函數註冊到 GUI 的迴圈之中，但在 Python 可以透過以下的方式來達成。

```python
from serial import *                    # 串列傳輸套件
from tkinter import *                   # GUI 套件

ser = Serial('COM3', baudrate=9600) # 創建序列傳輸物件

root = Tk()                             # 創建視窗物件
root.title("UART")                      # 視窗標題

def read():
    if size:=ser.in_waiting:            # 若有資料
        data = ser.read(size)           # 讀取資料
        ser.reset_input_buffer()            # 清空緩衝區
        print(data)
    root.after(100, read)               # 重複執行

root.after(100, read)                   # 每 100 毫秒讀取一次

def write():
    ser.write(entry.get().encode()) # 寫入資料


# ------------------省略 GUI 部分------------------ #

root.mainloop()                         # 執行視窗物件

```

（搭配 GUI 介面的示例）

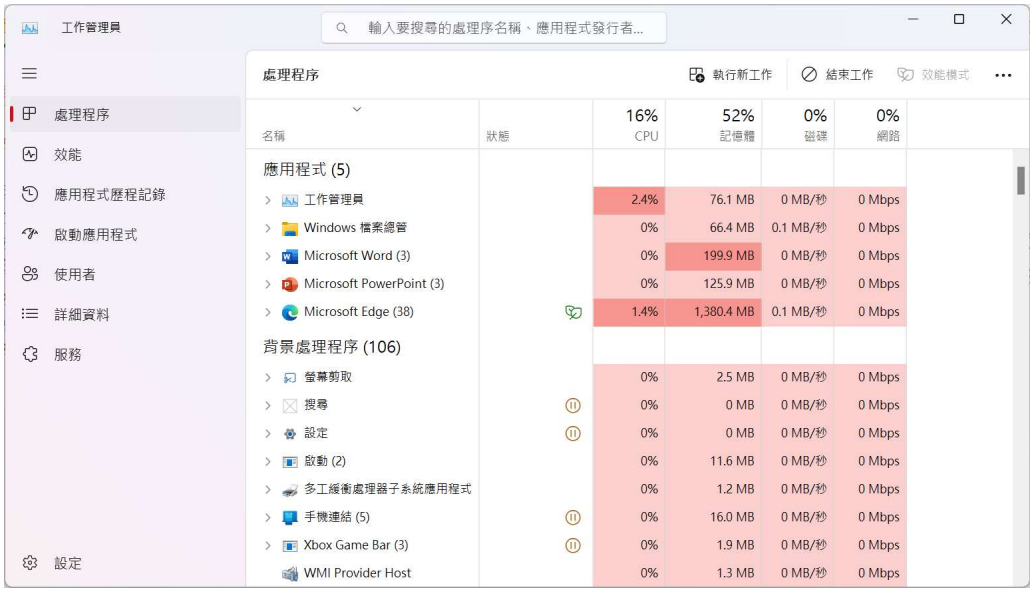
上方範例同樣使用了分時多工的方式，與先前不同的是這邊也將 GUI 加入至等待隊列。至於發送訊息的功能，只需將函數綁定到按鈕上即可，在此就略過不詳細說明。由這個範例可以得知：並行多工（Concurrency）不一定需要平行（Parallelism）。

線程與行程

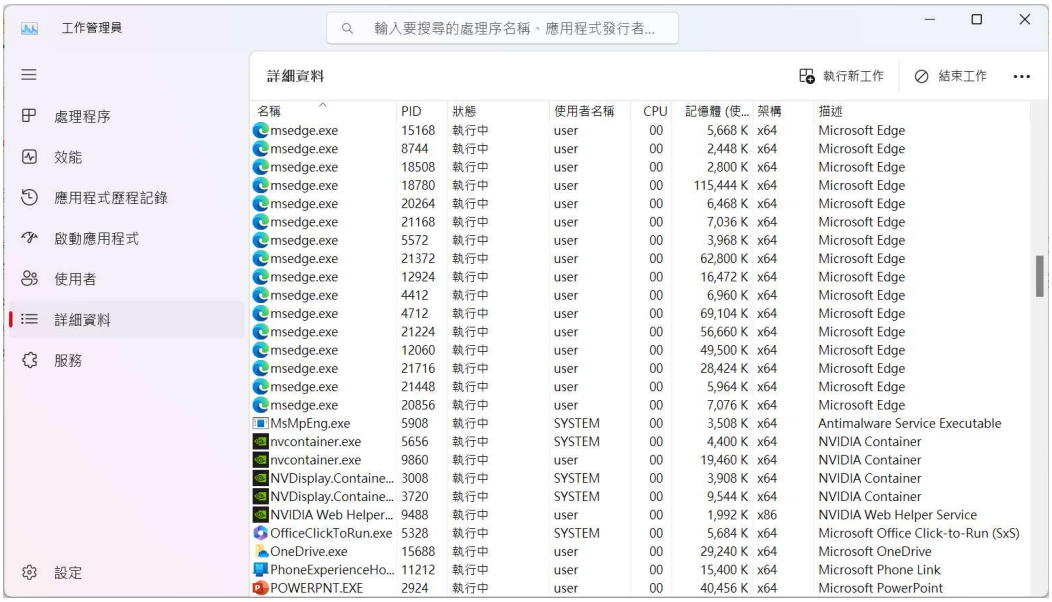
在本節中，我們將討論線程（或稱作：執行緒，英文 Thread）與行程（Process，大陸翻譯叫做進程）的差別。

行程

行程（Process）是執行中的程式實例，每一個行程有各自的 ID，且互相獨立。百聞不如一見，直接上圖！在工作管理員中，可以看到筆者此時開啟的應用程式，而應用程式後面括號的數字就是該應用程式開啟了多少個實例，比方說 Edge 瀏覽器就開了 38 個實例。



當我們點進詳細資料，確實看到了 Edge 正在執行超級多的實例，而每個實例都有自己的 PID（Process ID），所以 Microsoft Edge (38) 表示 Edge 開啟了 38 個行程。



（可以看到 Edge 開了許多行程）

多行程

電腦同時有許多的行程需要執行，而作業系統通過上下文交換（Context Switch）的方式在不同的行程之間不斷切換，給使用者造成一種多個行程同時執行的假象。在搶占式多工的系統中，每一個行程將會輪流執行不定長度的時間。到執行時間到達限制時，作業系統會產生一個中斷，將行程執行期間使用的所有暫存器儲存起來，並安排下一個行程執行。因此同時打開太多應用程式，各個程式平均能分配到的資源就會下降，讓使用者感覺電腦變慢。

線程

線程（Thread）究竟是什麼呢？以蓋房子來比喻，應用程式向建築公司（作業系統）下訂單，作業系統安排不同的訂單，分配給工人（Thread）來施工。Thread 是真正執行工作的實體，也是程式執行的最小單元。雙核心是指將兩個 CPU 核心集成在同一顆晶片上，因此搭載這顆晶片的電腦具有 2 個處理單元。然而，自從 Intel 在 2002 年推出超執行緒技術（Hyper-Threading），現今的一顆處理單元能夠模擬兩個線程，所以通常將 CPU 核心數乘以二就是線程的數量。不過 Intel 第 12 代後的 Alder Lake 架構 CPU，其中混合了效能核心 P core 和效率核心 E core，因此在 12 代以後的 CPU 上，這個方法不再適用。

多線程

Intel 的超執行緒技術並不限制一個行程只能使用兩個線程。實際上，線程（Thread）分為三個不同的層次：

- Hardware Thread：物理上能夠同時運作的執行單元數量（例如 i7-10510U 有 8 個線程）
- Kernel Thread：由作業系統管理（OS）的 thread，何時放到 hardware thread 執行由 OS 決定
- User Thread：由程式透過 Library 所建，OS 會將這些 thread 綁定到 kernel thread 上

Windows 作業系統的線程數量通常沒有限制，在 N:1 的 Threading Model 當中，程式可以創建 N 個 User Thread，但這些 User Thread 最終只會綁定到同一個 Kernel Thread。（在 Linux 與 macOS 對線程的創建數量有限制，更多資訊可參考：How Many Threads Can a Java VM Support? | Baeldung）

不同的程式語言可能使用不同的線程模型，例如 Go 語言的 goroutine 設計讓 Golang 支援高效的平行處理。而 Python 與 Java 使用的是 1:1 model，即每個程式執行的 thread 都對應到一個 kernel thread。（註：早期的 Java 使用 Green thread 進行多線程處理，這些 Green thread 是在 User Level 層面進行管理。後來 Java 才將多線程執行轉移到 native thread，即 Kernel Thread。）

由於 Python 的預設直譯器 CPython 的內存管理不具備線程安全性，所以引入了全域直譯器鎖（Global Interpreter Lock，縮寫為 GIL）來避免多個 Kernel Thread 同時執行 Python bytecode。儘管 GIL 在某種程度上提供了線程安全性，卻也導致 Python 在執行時限制只允許一個線程運行，這成為 Python 在多線程處理上常受批評的原因。

多線程 vs 多行程

最初我們期望透過平行計算（Parallel Computing）來充分發揮多核心處理器（例如 i7-10510U 就有 4 個核心）的性能，然而由於全域直譯器鎖的存在，Python 的 Multithreading 往往無法達到我們預期的效果。在這種情況下，我們可以借助第三方套件如 Numba 或是 Dask 這類第三方套件。Numba 不僅支援 CPU 上的平行運算，還能運行在 NVIDIA CUDA 和 AMD ROCm 等 GPU 環境上。如果曾使用 Python 進行機器學習（Machine Learning），可能聽說 Dask，它過支援 pandas、Numpy、scikit-learn、XGBoost 等套件，Dask 不但能在本地執行，也能擴展到叢集

上，實現「分散式運算」（Distributed computing）。在本節中，我們將比較多線程與多行程的概念，並明白各自的優點和缺點以及如何在這兩方式中選擇。

如果想在原生 Python 中平行運算，可以使用內建的 multiprocessing 庫，透過建立行程（Process）的方式來讓 Python 以平行的方式執行。這種方式的原理是各行程的 GIL 是相互獨立的。在類 Unix 系統下，也可以使用 os.fork() 達到相同的效果。（註：更多關於 os.fork() 與 multiprocessing 的差異可以看看這篇討論：Python multiprocessing process vs. standalone Python VM - Stack Overflow。）

由於 CPython 的 GIL 在進行阻塞 I/O 操作時會釋放，因此對於 I/O 密集型任務，多數文章建議使用 multithreading，而對於計算密集型的任務使用 multiprocessing。而在沒有 GIL 的語言中，要使用 Multithreading 還是 Multiprocessing？可以參考文章 Multithreading vs. Multiprocessing: What's the Difference? | Indeed.com。和多行程類似，當 kernel thread 的數量大於 hardware thread 的數量時， 線程也會進行上下文交換。更多資訊可以看這篇文章：Difference between Thread Context Switch and Process Context Switch - GeeksforGeeks。

但使用 multithreading 的效能不如預期，也會造成除錯上的困難，因此 multithreading 已經逐漸被下個章節所介紹的非同步、事件驅動這類方式取代[1]。

非同步與事件驅動

在本節中，我們將討論非同步與事件驅動的概念和影響。非同步與事件驅動是實現多工的另外兩種方式，它們可以提高程式的反應速度和靈活性。網路上有個例子可以幫助我們理解阻塞非阻塞與同步非同步的概念[2]。

老張燒開水的故事（故事來源網路）。
出場人物：老張，水壺兩把（普通水壺，簡稱水壺；會響的水壺，簡稱響壺）。

老張愛喝茶，時常二話不說便煮起了開水。

第一次：老張置水壺於火上，立等水開（同步阻塞）。
> 水開了，但老張覺得自己這樣乾等有點傻。

第二次：老張置水壺於火上，到客廳看電視，時不時看看廚房的水開了沒有（同步非阻塞）。
> 老張依舊覺得自己有點傻，於是買了把會響笛的水壺。水開之後，能發出嘀~~的噪音。

第三次：老張置水壺於火上，立等水開。（非同步阻塞）
> 老張就這樣傻等壺響，覺得意義不大。

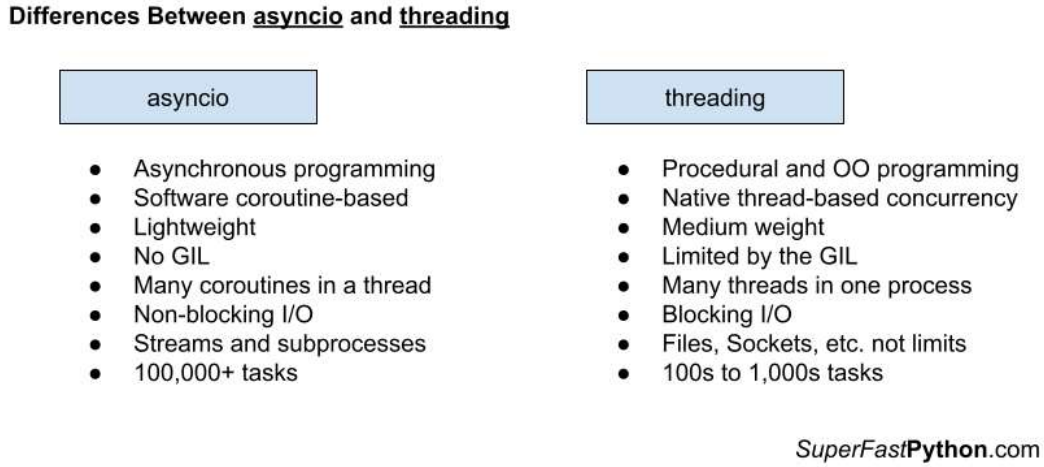
第四次：老張把響水壺放在火上，去大廳看電視，直到響了再去拿壺。（異步非阻塞）
> 老張覺得自己聰明了。

[1] 曾涉獵. (2015, May 17). 為什麼 thread (執行緒、線程)越少越好?. Goodspeedlee.blogspot.com. https://goodspeedlee.blogspot.com/2015/05/thread.html
[2] Hejianhui. (2020, October 10).Netty 学习前基本知识 — BIO 、NIO 、AIO 总结. Weixin.qq.com. https://mp.weixin.qq.com/s/7DrH3vdl0xVJp97Q-fjTAA

同步與非同步的差異是在任務完成前需要持續關注該任務是否結束。會響的水壺代表一種事件通知的機制。而阻塞與非阻塞是在於等待的期間老張有沒有去做其他事情。

依據 Amdahl's law，就算可以無限提升平行化運算的速度，程式的總運算時間仍會被無法平行化運算加速的部分所侷限[3]。Multicore Scalability Through Asynchronous Work (vt.edu) 提到以非同步的方式將不影響數據一致性的任務從關鍵路徑中移除，並移動到後台非同步執行，這樣可以減少關鍵路徑的大小，從而提高系統的性能和可擴展性。

需要注意的是 Multithreading 與 Asyncio 並不一樣，在有 GIL 的情況下，當 Multithreading 用於阻塞型的 I/O 任務時，GIL 就會釋放，這時 Python 就會去執行其他線程。而 Asyncio 卻永遠都只有單個線程，且 Asyncio 只在處理非阻塞型的 I/O 上才有效果。計算密集型的任務無法使用。



（Asyncio 與 Threading 的比較[4]）

我們有在 Process 的章節提及搶占式多工（Preemptive multitasking）的概念，而 Asyncio 是使用一種稱為協作式多工（Cooperative Multitasking）的方式。相較於搶佔式多工（Preemptive multitasking）由作業系統決定任務切換時機。協作式多工要求程式定時放棄（yield）自己的執行權，轉讓給下一個程式執行。這種透過程式之間合作而達到的多工方式，就稱作協作式多工。

適時放棄執行權也能提升程式整體的執行效率，非同步程式設計的技巧就是在進行阻塞操作時（也就是等待期間）將執行權轉交出去，在程式等待的同時，讓其他程式進行操作。若要觀看更多與非同步程式設計的討論可以閱讀「程式設計該同步還是非同步？ | iThome」這篇文章。

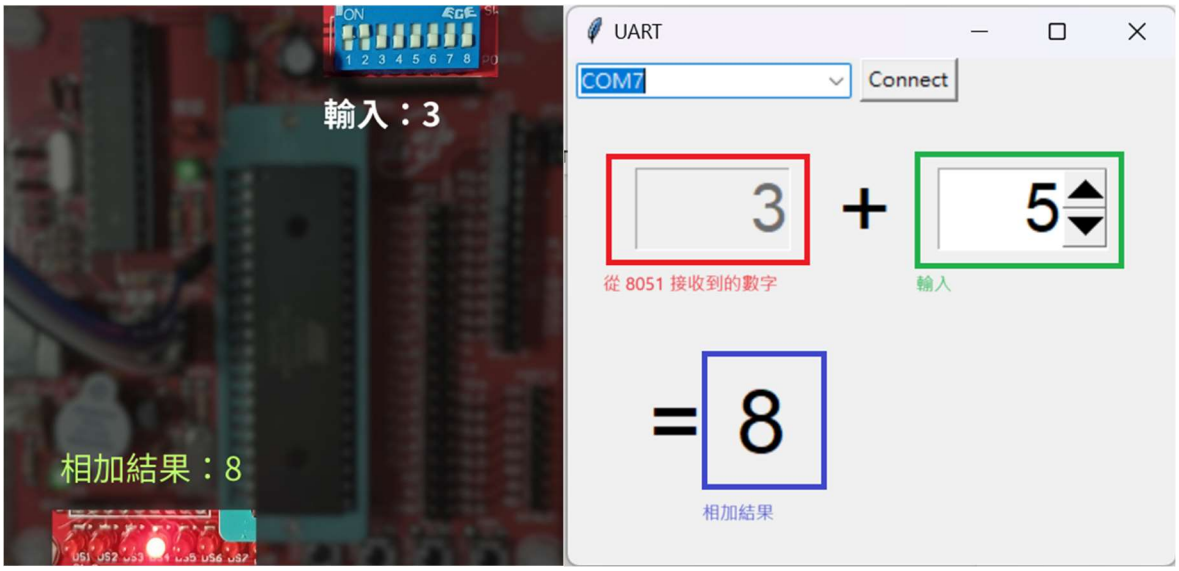
下個章節將會演示如何在 8051 上實作非同步程式設計。期望讀者在明白軟硬體相似時，能有醍醐灌頂的感受。

[3] https://blog.maxkit.com.tw/2023/04/amdahls-law.html
[4] https://superfastpython.com/asyncio-vs-threading/

範例演示

範例功能

這個示例演示了如何實現一個應用程式，允許電腦和 8051 通過 UART 介面互相傳輸 8 位元的數字。兩方將接收到的數字與本地數字相加，然後顯示結果。



（功能展示，左圖為 8051、右圖為電腦端）

接線方式這裡就不多做說明，只提醒一點：作業繳交區有提供對應型號 USB to UART Bridge 的驅動程式，要安裝電腦才能夠找到 COM Port。

8051 端（組合語言）

範例將會展示三種不同的寫法。讀者可以比較這三者之間的差異。

```
1   ORG     0000h
2           MOV     SCON,#01010000b     ;設定串列傳輸工作於模式1, 可接收資料
3           MOV     TMOD,#00100000b     ;設定TIMER1工作於模式2(自動載入模式)
4           MOV     TH1,#253            ;設定自動載入值為253(在石英震盪器為11.059MHz時, 鮑率為9600bps)
5           SETB    TR1
6   RECEIVE:
7           JNB     RI,$                ;判斷是否接收完畢(RI=1), 否則停在此行
8           CLR     RI                  ;清除接收完畢旗號
9           MOV     A,SBUF
10          MOV     R7,A
11          SUBB    A,#030h
12          CPL     A
13          MOV     P1,A
14          CPL     A
```

（8051 阻塞範例程式片段）

上方的程式片段是來自其他的項目。放在這裡主要是讓讀者對阻塞的寫法有一定程度的了解。閱讀這個程式片段可以發現，當程式在得到輸入之前（RI 等於 1 之前），程式會一直停（阻塞）在第七行。當然這只代表程式會等待輸入而已，不代表無法實現多工，依照不同的需求，可以設計成在輸出一個資料後等待輸入，並在輸入完成後進行輸出。如此往復也能稱作多工（雙向傳輸）。

```
1   ORG 0000h
2
3       MOV     TMOD,#00100001b ; Timer1 in Mode 2, Timer0 in Mode 1
4       MOV     SCON,#01010000b ; UART in Mode 1
5       MOV     TH1,#0FDh       ; Baud Rate = 9600 bps at 11.0592MHz
6       SETB    TR1             ; Start Timer 1
7
8       SETB    TI              ; 將 TI 設為 1，表示 UART 可以傳送資料
9   LOOP:
10      JB      RI, RECV        ; 有輸入資料時，跳到 RECV
11      JB      TI, SEND        ; 能傳送資料時，跳到 SEND
12      SJMP    LOOP            ; 無資料時繼續迴圈
13
14  RECV:
15      CLR     RI
16      MOV     A, SBUF         ; Read UART data
17
18      ; 將 DIP Switch 的值與 UART 接收到的資料相加
19      CPL     A
20      ADD     A, P0
21      INC     A
22
23      MOV     P1, A           ; 顯示結果在 LED 燈上
24      LJMP    LOOP
25
26  SEND:
27      CLR     TI
28      MOV     A, P0
29      CPL     A
30
31      MOV     SBUF,A          ; 傳送 DIP Switch 的值
32      LJMP    LOOP
33  END
```

（8051 同步非阻塞範例）

而這個範例程式是同步非阻塞的寫法，以 Java 語言的角度來看就是 NIO 模型。程式以輪詢的方式不斷檢查（或者說等待）輸入或輸出事件，在事件發生之前我們的程式將持續運行，所以它是非阻塞的。以老張煮水的故事來說，老張（程式）會不斷的檢查（LOOP 迴圈）開水有沒有滾（是否輸入 RI/是否輸出 TI）。由於我們將讀取輸入與進行輸出的工作交給底層，在這裡就是 8051 的硬體電路，因此我們不用處理輸入與輸出的部分。若是使用軟體，那 I/O 的部分就會交給外部進行處理，例如作業系統（Operating System，縮寫 OS）或在網路程式設計上的話有可能是將資料丟給伺服器處理，直到伺服器處理完畢再傳送回來。（畢竟工作終究得要有人做）而在這段期間進行其他的任務就能稱為「非阻塞」。很明顯能知道，這個程式能夠處理輸入也能處理輸出，達到多工處理的功能（雙向傳輸）。

```
1   PROG  EQU   0000h
2         ORG   PROG+0000h
3         SJMP  START
4
5   ; 中斷向量表
6         ORG   PROG+0023h
7         LCALL UART_ISR
8         RETI
9   ;
10        ORG   PROG+0030h
11  START:
12        MOV   TMOD,#00100001b   ; Timer1 in Mode 2, Timer0 in Mode 1
13        MOV   SCON,#01010000b   ; UART in Mode 1
14        MOV   TH1,#0FDh         ; Baud Rate = 9600 bps at 11.0592MHz
15        SETB  ES                ; Enable UART Interrupt
16        SETB  EA                ; Enable Interrupt
17        SETB  TR1               ; Start Timer 1
18
19        SETB  TI                ; 將 TI 設為 1，表示 UART 可以傳送資料
20        SJMP  $                 ; 停留在這裡
21
22  UART_ISR:
23        JB    RI, RECEIVED      ; 有輸入資料時，跳到 RECEIVED 進行讀取
24  TRANSMITTED:
25        CLR   TI
26        MOV   A, P0
27        CPL   A
28
29        MOV   SBUF, A           ; 傳送 DIP Switch 的值
30        RET
31  RECEIVED:
32        CLR   RI
33        MOV   A, SBUF
34        ; 將 DIP Switch 的值與 UART 接收到的資料相加
35        CPL   A
36        ADD   A, P0
37        INC   A
38
39        MOV   P1, A             ; 顯示結果在 LED 燈上
40        RET
41        END
```

（8051 異步非阻塞範例）

這個程式以 Java 來看就是 AIO 模型，程式碼使用中斷的方式（事件驅動）處理 UART 通訊。它設置了 UART 接收中斷（RI）的中斷服務程式 UART_ISR。當 RI 為 1 時，將觸發中斷並呼叫 UART_ISR 處理接收，輸出同理。以老張煮水的故事來說，我們發送訊息（煮水）後並沒有等待 TI 變成 1，而是可以進行其他的事情（雖然這邊是直接停在第 20 行）。而我們並沒有不斷的檢查水是否已經煮好（以 LOOP 檢查 TI），而是讓水壺好的時候發出響聲（觸發中斷），這個方法似乎又比剛才的方式聰明了。

在 Linux 可以使用 Epoll 的方式來監聽多個描述符的狀態，當其中一個描述符變更時通知程式。而 8051 的 RI 與 TI 已經 OR 在一起，所以不能將輸入與輸出拆成兩個事件，實屬可惜。

電腦端（PYTHON 語言）

在這個範例功能中，8051 會持續不斷的傳送資料，而我們以 Python 語言開發的客戶端也將不停的接收資料（當然你也能讓 8051 只在資料改變時才傳送資料）。很典型的 I/O 密集行任務，範例是使用 Python 的 Thread 套件。由於 GIL 在線程進行阻塞式的 I/O 操作與呼叫 time.sleep() 時都會釋放，所以這個 Python 實際上的運作模式是不斷的在 GUI 與讀取 8051 資料的線程之間切換（Context Switch）。

```
from serial import *
from serial.tools import list_ports
from tkinter import ttk
from tkinter.messagebox import *
import tkinter as tk
import threading
import time

baudrate = 9600


# ---------------------------------------------- #
ser: Serial = None

def loop() -> None:
    while(True):
        # Update port list
        port_chose['values'] = [x.device for x in
list_ports.comports()]

        if(ser is None or not ser.isOpen()):
            continue

        try:
            data = ser.read(1)
            ser.reset_input_buffer()
            print(data)
            text_a.set(f'{int.from_bytes(data)}')
            entry_a.update()
            time.sleep(0.1)
        except UnicodeDecodeError:
            # print("UnicodeDecodeError")
            pass

        ans['text'] = str(int(text_a.get()) + int(text_b.get()))


def connect():
    global ser
    if ser is not None:
        ser.close()
        del ser

    if com_port.get() == "":
        showerror("Connecting Error", "Please select a port to
connect")
        return
    ser = Serial(com_port.get(), baudrate, timeout=0,
writeTimeout=0)

def send() -> None:
```

```
    if ser is None or not ser.isOpen():
        showerror("Sending Error", "Please connect to a port first")
        return
    ser.write(int(entry_b.get()).to_bytes())

# --------------------- GUI --------------------- #
root = tk.Tk()
root.title("UART")
root.geometry('360x300')
root.minsize(360, 300)

ctrl_frame = tk.Frame(root)
ctrl_frame.pack(side=tk.TOP, fill=tk.X)

com_port = tk.StringVar()
port_chose = ttk.Combobox(ctrl_frame, textvariable=com_port) # 下拉式
清單
port_chose.pack(side=tk.LEFT, padx=5, pady=3)

connect_btn = tk.Button(ctrl_frame, text="Connect", command=connect)
connect_btn.pack(side=tk.LEFT)

frame1 = tk.Frame(root)
frame1.pack(side=tk.TOP, fill=tk.BOTH, padx=30, pady=30)

text_a = tk.StringVar()
entry_a = tk.Entry(frame1, width=4, font=('Arial', 30),
state=tk.DISABLED,
                   textvariable=text_a, justify=tk.RIGHT)
entry_a.pack(side=tk.LEFT, padx=10)

tk.Label(frame1, text=" + ", font=('Arial', 40)).pack(side=tk.LEFT)

text_b = tk.StringVar()
entry_b = tk.Spinbox(frame1, width=4, font=('Arial', 30),
textvariable=text_b,
                     justify=tk.RIGHT, from_=0, to=255,
command=send)
entry_b.pack(side=tk.LEFT, padx=10)

frame2 = tk.Frame(root)
frame2.pack(side=tk.TOP, fill=tk.BOTH, padx=30, pady=30)

tk.Label(frame2, text=" = ", font=('Arial', 40)).pack(side=tk.LEFT)
ans = tk.Label(frame2, text="0", font=('Arial', 40))
ans.pack(side=tk.LEFT)


# --------------------- Main --------------------- #
t = threading.Thread(target=loop)
t.daemon = True
```

```
t.start()

root.mainloop()
```

結語

我們已經在這個雙向通訊的例子透過 Python 和 8051 的組合語言示例來說明阻塞與非同步等概念。這個例子當中也有能再進一步的地方，比如說讓 8051 只在資料改變時才傳送資料，或是讓 Python 的介面也使用非同步的寫法。筆者當初聽到 RI/TI 就聯想到程式設計的非同步程式設計，原本短短 41 行的程式碼居然能想到 5000 字的內容。希望這篇文章能給讀者其他範例沒有提到的內容，並提供一個全新角度的切入點來觀看其他的範例。

參考資料

- Mathew, A. (2020). Multicore scalability through asynchronous work (Master's thesis). Virginia Polytechnic Institute and State University.
- Jess, A.(2017, April 18). Grok the GIL: How to write fast and thread-safe Python. Opensource.com. https://opensource.com/article/17/4/grok-gil
- Sysprog.(2023, August). 並行程式設計: 概念. Hackmd.io. https://hackmd.io/@sysprog/concurrency/%2F%40sysprog%2Fconcurrency-concepts
- 王建興.(2012, June 22). 程式設計該同步還是非同步？Ithome.com.tw. https://www.ithome.com.tw/voice/74544
- Jason Brownlee. (2023, January 20). Asyncio vs Threading in Python. https://superfastpython.com/asyncio-vs-threading/
- Pi314. (2016, Sep 19). Threading Models. Github.com. https://github.com/pi314/pi314.notes/blob/main/os-threading-models.rst
- Jack. (2021, July 15). Thread Model 介紹. Github.io. https://yu-jack.github.io/2021/07/15/thread-model/
- Yovan Li. (2019, February 10)OS Process & Thread (user/kernel) 筆記. Medium.com. https://medium.com/@yovan/os-process-thread-user-kernel-筆記-aa6e04d35002
- 跨元探索. (2022, August 4)【多工技術】OFDM 多重載波調變技術. Vocus.cc. https://vocus.cc/article/62eb1500fd8978000182a421
- Alwareadit. (2020, May 25). Difference between Thread Context Switch and Process Context Switch. Geeksforgeeks.org. https://www.geeksforgeeks.org/difference-between-thread-context-switch-and-process-context-switch/
- Indeed. (2023, February 4). Multithreading vs. Multiprocessing: What's the Difference? https://www.indeed.com/career-advice/career-development/multithreading-vs-multiprocessing
- Flaviu Cicio. (2023, September 21). How Many Threads Can a Java VM Support? Baeldung.com. https://www.baeldung.com/jvm-max-threads