Práctica 3

Uso de interrupciones para leer valores de un sensor

Grado en Ingeniería en Robótica Software
GSyC, Universidad Rey Juan Carlos

1. Introducción

Las interrupciones suponen un mecanismo muy empleado en programación y un tema muy interesante en cómo las manejan los sistemas operativos. Por definición, una interrupción es un evento que ocurre de forma asíncrona (en cualquier momento) respecto a la ejecución del programa. A continuación, se expone un breve repaso al funcionamiento de un programa cuando este se ejecuta.

Durante la ejecución de un programa se van sucediendo saltos entre las distintas líneas de código. Por ejemplo, cuando se llama a una función, se produce un salto de ejecución de la línea actual del programa a dicha función. La dirección actual se almacena en la pila de ejecución del programa; dicho más apropiadamente, del proceso asociado al programa, aunque esto es otra historia sobre la relación programa-proceso que establece un sistema operativo. En UNIX, un proceso corresponde a un programa en ejecución. Almacenada esa dirección en la pila del proceso, se salta a la función y se ejecuta la rutina de instrucciones que engloba esta. Una vez termina de ejecutarse la función (puede haber más llamadas anidadas a funciones), se salta a la dirección que indique el puntero de pila, que será la última *apilada* y que corresponderá con la dirección de la instrucción que llamó a la función. El procesador seguirá ejecutando el proceso a partir de esta dirección; esto es, el programa sigue su curso...

1.1. Interrupciones o eventos

Hasta aquí todo normal, el programa sigue estrictamente las instrucciones según se haya diseñado este. Pero esta armonía del programa se verá interrumpida si introducimos el manejo de eventos en este. La principal novedad que se presenta al tratar eventos es que estos se dan de forma asíncrona; esto es, mientras que el programa está haciendo *sus cosas*, ocurre un evento, y hay que tratarlo. Cuando el SO, según su planificación, decida que es momento de tratar el evento, entonces para la ejecución *normal* del programa para ejecutar la rutina de tratamiento de dicho evento, también denominado *callback*. Sintácticamente, un *callback* es una función especial, pues es una función que se pasa a otra como argumento. Ejemplo de *callback*:

```
def botonPresionado ():
    print("El boton ha sido presionado")

GPIO.add_event_detect(BUTTON_GPIO, GPIO.FALLING,
  callback=botonPresionado, bouncetime=100)
```

1.2. Programación concurrente

En este punto adquiere especial relevancia la programación concurrente; esto es, la gestión de varios hilos de ejecución en un programa. Mientras que en programación secuencial solo tratamos con un hilo de ejecución, el hilo principal o *main*, y las instrucciones se van sucediendo una tras otra, en programación concurrente lanzamos conscientemente diferentes hilos, o *threads*, que nacen siempre a partir del hilo principal el programa; se produce lo que se denomina un *fork*, cuyo nombre, tenedor, se debe a su forma: el mango sería el hilo principal y luego se bifurca en varios hilos...

Si juntamos el tratamiento de interrupciones con la programación concurrente, tendremos programas muy eficientemente diseñados. Una buena práctica es, por ejemplo, asociar un hilo diferente a cada evento, para que el programa se pueda ejecutar a la vez que se puedan estar tratando los eventos. En realidad, esto es posible si el ordenador dispone de varios procesadores o núcleos, algo que en la actualidad es casi imposible no encontrar. Concretamente, en la Raspberry Pi, se pasó de uno a cuatro núcleos, siempre de la compañía *Broadcom*, a partir de su modelo 2B (2014).

2. Comportamiento de un pin GPIO de entrada

Hasta ahora hemos trabajado con los GPIO en su configuración de salida (OUT), para manejar convenientemente un LED. Es una configuración sencilla, porque somos nosotros los que variamos a nuestro antojo la señal de salida del pin. Pero al leer de un pin GPIO hemos de tener en cuenta cómo es su señal de entrada, y es que esta oscilará entre 0 y 1 si no está conectado a un voltaje, por ejemplo cuando leemos de un sensor; típicamente, este dará una señal cuando *sense* algo. Este hecho, como cualquier evento de los que hemos hablado en la sección anterior, se produce de forma asíncrona, en cualquier momento. Dicho de otro modo, si no leemos en el momento oportuno, no podremos distinguir la señal dada por el sensor de la propia señal oscilante entre 0-1 del pin; nunca podremos saber cuándo ha ocurrido el evento externo que se está sensando.

La solución a lo anterior es fijar un valor al pin y, de este modo, podremos detectar cuándo ocurre un cambio de valor, por ejemplo cuando el sensor envía señal porque se ha producido un evento. Para fijar un determinado valor a un pin GPIO, existen las denominadas resistencias *pull-up* y resistencias *pull-down*. Estas son unas resistencias específicas de la librería `rPi.GPIO` que suministran, vía software, un voltaje fijo para que el pin GPIO tenga un valor estático hasta que sea anulado por una señal más fuerte.

Se establece una resistencia *pull-down* (en 0) para un determinado pin GPIO cuando se espera que la señal conectada al mismo lo eleve a 1 cuando ocurra un evento. Y viceversa, se debe establecer un *pull-up* (en 1) si se espera que la señal conectada lo baje a 0 cuando ocurra el fenómeno que está siendo sensado.

3. Ejercicios prácticos de manejo de eventos

A continuación veremos diferentes ejemplos prácticos sobre cómo manejar eventos. En todos ellos usaremos un botón o pulsador como sensor o dispositivo de entrada, y el objetivo será siempre saber cuándo se ha presionado el pulsador. El esquema de conexión seguido para estos ejemplos es el que aparece en la Figura 1.

Nótese que el cable rojo está conectado al pin de lectura GPIO 16 (modo BCM), y el cable negro al sexto pin (GND-BOARD o toma de tierra). Por otro lado, téngase en cuenta que el pulsador ha de colocarse en la zona central de la protoboard, donde —recordemos— el circuito está discontinuado; es el pulsador el que une (o no) esas filas para precisamente cerrar el circuito (o no).

3.1. Método 0. Pidiendo el valor continuamente

En este primer ejemplo, cuyo código tenemos en `sinInterrupciones.py` (y cuyo *snippet* vemos a continuación), comprobamos continuamente el estado del pin de lectura del LED. Esto, como ya hemos adelantado, es poco eficiente. Además, y lo que es peor, siguiendo esta dinámica de *comprobación exhaustiva* se puede

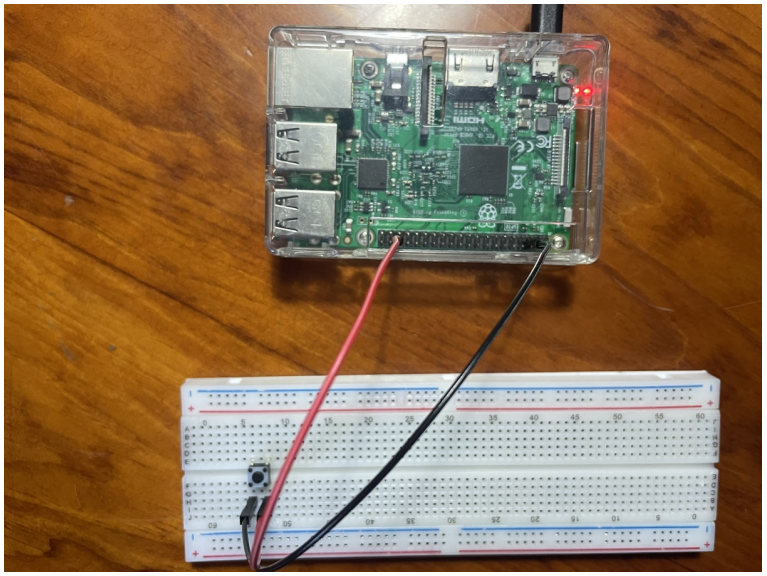

Figura 1: Esquema de conexión de pulsador empleado para estos ejemplos

llegar a saturar el procesador.

```
# Activamos resistencia pull_up_down en modo HIGH, esto es:
# - HIGH: estado por defecto del GPIO (no se ha pulsado).
# - LOW: estado del GPIO cuando se ha pulsado el boton.
GPIO.setup(pulsadorGPIO, GPIO.IN, pull_up_down=GPIO.PUD_UP)

pulsado = False

while True:
    if not GPIO.input(pulsadorGPIO): # la lectura siempre da 1 (HIGH/True) excepto al pulsar,
                                     # y solo en ese instante, que da 0 (LOW/False)
        if not pulsado: # con esta variable evitamos considerar mas de una vez una pulsacion;
                        # puede que se lea +1 vez el estado del pin antes de cambiar su estado
                        # Este fenomeno se conoce como rebote (o bounce). Algunas funciones
                        # permiten establecer un parametro denominado "bouncetime"
            print("El boton se ha pulsado")
            pulsado = True
    else:
        pulsado = False

    time.sleep(0.1) # si pulsamos rapida/ veremos que algunas se escapan...
```

Como observamos en el código, además de establecer el pin en modo lectura, fijamos un valor al mismo —como ya hemos comentado previamente—. Establecemos *HIGH* como valor por defecto (`pull_up_down=GPIO.PUD_UP`), y así cuando se pulse el botón, el estado cambiará a *LOW*. A continuación, comprobaremos continuamente el estado del pin en un bucle infinito; cuando se pulse el botón, se muestra un mensaje por pantalla.

Para no saturar al procesador, establecemos una determinada frecuencia con la instrucción `time.sleep(segundos)`. A menor frecuencia, mayor será el alivio para el procesador, pero más comprobaciones nos perdemos. Por ejemplo, con un *sleep* de $0{,}1s$, obtenemos una frecuencia de comprobación de $f = 1/t \implies f = 1/0{,}1 = 10Hz$; dicho de otro modo, todas las pulsaciones que se hagan dentro de ese intervalo contarán como una. Así pues, hemos de llegar a un valor que equilibre ambos frentes. Además, otro factor negativo de emplear este mecanismo para tratar eventos es que no sabremos nunca cuándo exactamente se ha pulsado el botón; puede ser cualquier momento del intervalo establecido.

3.2. Método 1. Interrupciones con wait_for_edge

En Raspberry Pi no existen las interrupciones hardware (como sí las hay, por ejemplo, en Arduino), pero sí se pueden simular vía software gracias a la librería de manejo de GPIO (`RPi.GPIO`). Sea de una forma o de otra, las interrupciones se activan cuando el estado del pin que estamos leyendo cambia de valor. Concretamente, se pueden activar de dos modos: cuando el pulso cambia de 0 a 1 (*rising*) o cuando cambia de 1 a 0 (*falling*). En nuestro contexto, *rising* sería sería el instante en que el pulsador va hacia arriba (se deja de pulsar) y *falling* el instante en que va hacia abajo (es pulsado).

```
while True:
    GPIO.wait_for_edge(pulsadorGPIO, GPIO.RISING)
    print("El boton se ha pulsado")
```

En este *snippet* del codigo `interrupcionEdge.py` también facilitado, vemos la aplicación de la función `wait_for_edge`. Se configure en modo *rising* o *falling*, recibiremos una notificación cuando el pin cambia de estado. Ya no tenemos que estar comprobando una y otra vez cuál es el estado del pin. También se puede configurar para recibir la notificación en cualquier caso, estableciendo `BOTH` en su lugar.

Al ejecutar este código te percatarás de dos cosas: la primera, es que el comportamiento del programa no es del todo correcto, pues hay ocasiones en que se muestra más de una vez el mensaje de *pulsado* cuando en realidad solo se ha pulsado una vez el botón; la segunda, es que no hay forma de parar el programa de la forma habitual, pulsando la combinación *CTRL+C*.

La explicación a estas dos cuestiones es sencilla. La razón de la primera es que no hemos implementado ningún mecanismo antirebote, como sí hicimos en el primer ejercicio, y así vemos los efectos. La razón de la segunda es que, en esta ocasión, como podemos observar, no damos un respiro al procesador en el bucle infinito, y el resultado es un procesador dedicado exclusivamente a dar vueltas, es por ello que ignora la la combinación *CTRL+C*.

La solución a ambos problemas también es sencilla. Para el primer problema, hemos de implementar un sistema antirebote, similar al primer ejercicio. Para el segundo problema, la solución es establecer una frecuencia de iteración en el bucle, como también hacíamos en el primer ejercicio. De momento, para salir del programa, en lugar de emplear la combinación *CTRL+C*, que se trata de una interrupción de teclado (y por ello está siendo ignorada, porque el procesador no da para más), emplea la combinación *CTRL+Z*, que lo que hace es detener, que no terminar, el proceso `python3 interrupcionEdge.py`. Es por ello que este proceso sigue ejecutándose en *background*. Puedes comprobar que está entre los procesos en ejecución con el comando `ps ax`. También puedes —y debes— terminarlo definitivamente mediante la orden `killall -9 python3` (o bien con la orden `kill -9 PID`, PID es el ID del proceso).

Otra solución más digna para el segundo problema es emplear un *thread* aparte dedicado a tal evento, pero esto es otro cantar... Veamos la solución plausible, sin necesidad de más hilos de ejecución en el siguiente apartado.

3.3. Método 2. Interrupciones con add_event_detect

La principal y radical diferencia entre este y el anterior método es el uso de *callbacks*. Esta será la función que se invoque cuando ocurra el evento; esto es, cuando se haya pulsado el botón. En realidad, un *callback* tiene asociado un hilo de ejecución aparte, pero no lo hemos de implementar nosotros, por eso el código resulta más claro, más sencillo y —sobre todo, si no hemos trabajado con *threads*— más entendible.

```
GPIO.add_event_detect(pulsadorGPIO, GPIO.FALLING,
  callback=callbackBotonPulsado, bouncetime=500) # expresado en ms.

signal.signal(signal.SIGINT, callbackSalir) # callback para CTRL+C
signal.pause() # esperamos por hilo/callback CTRL+C antes de acabar
```

Como vemos en este *snippet* perteneciente al código facilitado en `interrupcionEvent.py`, la función `add_event_detect` admite ese tercer parámetro de *callback*, además de un cuarto donde expresar el tiempo de rebote (*bouncetime*) que se ajuste a nuestras necesidades. El valor establecido ($500ms$) ha demostrado funcionar convenientemente. Con estos dos parámetros, estas dos mejoras respecto al mecanismo anterior, ya tenemos cazados de una forma muy sencilla los dos problemas que se planteaban en el anterior ejemplo.

Por otro lado, estamos usando la librería signal, que nos permite tratar con interrupciones de teclado, como es el caso de la combinación $CTRL+C$ para terminar la ejecución. En este caso hemos de usar este mecanismo porque, de lo contrario, al tratar con varios hilos de ejecución (aunque no los veamos), si pulsamos $CTRL+C$ solo afectaría al hilo principal, que en este caso de hecho no está haciendo nada después de invocar el *callback*. Y ya que tratamos mediante otro *callback* la terminación del programa (función `callbackSalir`), aprovechamos a limpiar los recursos empleados de los puertos GPIO en dicha función mediante la instrucción `GPIO.cleanup()`.

Ejercicios

Ejercicio 1

Mejora la implementación del código facilitado en `interrupcionEdge.py` para que no ocurran los problemas descritos del mismo. Al fichero resultante denomínalo `interrupcionEdgeBueno.py`

Ejercicio 2

Rediseña el esquema de conexión, así como la implementación de los tres códigos facilitados para dar soporte a dos botones que a su vez operan sobre dos LEDs de forma independiente. Así, al pulsar uno de los botones, se encenderá un LED rojo (y se apagará al soltar dicho botón); y, al pulsar el otro botón, lo mismo pero con el otro LED, que será verde.

Recuerda que deberás ofrecer esa funcionalidad bajo las tres implementaciones que se han expuesto aquí. Los códigos resultantes estarán en ficheros con estas denominaciones:

1. `sinInterrupcionesMejorado.py`
2. `interrupcionEdgeMejorado.py`
3. `interrupcionEventMejorado.py`