Part 85
-
Multi-Tasking
-
Multi-Processing

Version: 2021-01-14

Multiprocessing

Multiprocessing (also called parallel processing) is the method of using more than one processor/core by an application. Multiprocessing is highly suitable for heavyweight tasks such as CPU bound tasks. Python multiprocessing module provides a powerful API to develop multiprogramming applications. Processors such as the Raspberry Pi include several cores in their CPU and as a result creating multiprocessing applications on such systems are highly efficient.
In a multiprocessing application, all processes are independent of each other and have their share of the overall system resources, such as memory, processing power, etc. Processes in a multiprogramming application can share memory and communicate with each other using functions provided in the multiprocessing API.

Multiprocessing or threading

You might be confused about whether to use multiprocessing or threading (or threads) in a multitasking application. Some key differences between the two are summarized to clarify this concept:

- Multiprocessing starts with different processes that are completely independent of each other. Threading, on the other hand, launches threads which are dependent on the parent process
- The processes in a multiprocessing system have their own CPU and memory spaces which are unique to each process. Threading applications, on the other hand, utilise the same CPU and memory present in the parent process
- In a multiprocessing system, if a process fails due to an error or exception, the other processes continue to run. On a thread-based system, however, if a thread fails then all other threads terminate
- Sharing objects (e.g. data) in a thread-based system is very easy because such objects are global to all the threads in the system. Sharing data in a multi-processing system however is more difficult because special synchronisation and software functions are required to share objects between different processes.
- Threading is best suited to input-output based applications. Multiprocessing, on the other hand, is more suited to CPU intensive applications

How many cores?

Raspberry Pi is a multi-core CPU based computer. The following Python statements can be used to find out how many cores our CPU has:

```
import multiprocessing
cores = multiprocessing.cpu_count()
print("Core count = %d" % cores)
```

Output is

```
Core count = 4
```

Process calls
Python offers a multiprocessing module that can be used to start parallel processes. The function `Process` is used to start functions as processes. The `Process` call is similar to threads but in a `Process` call, the function runs in a process and not in a thread. An example program using the `Process` call is given in Figure 9.1 (program: multiproc .py).

```
from multiprocessing import Process

def NewProcess():
   print("Hello from the new process...")

pr = Process(target = NewProcess, args=())
pr.start()
print("Hello from the creator...")
```

Notice the program above could have been written by importing the whole multiprocessing module as shownbelow.

```
import multiprocessing
def NewProcess():
   print("Hello from the new process…")

p = multiprocessing.Process(target = NewProcess, args=())
p.start()
print("Hello from the creator...")
```

Events
Events are useful tools to synchronise processes in multiprocessing applications. A process can be programmed to wait for an event flag. If the event flag is cleared, then the process will block. If on the other hand the event flag is set, then the thread will continue. The basic event calls are `set()`, `wait()`, and `clear()`.
The multiprocessing Event has the following methods (assuming e is the created event flag):

- `e.wait(t)` : wait t seconds for the event flag to be set. If not set within the specified timeout, continue. Here, t is optional
- `e.set()` : set the event flag
- `e.clear()` : clear the event flag
- `e.is_set()` : check if the event flag is set

Conditions
A condition is similar to an event flag. A condition allows a process to wait and then be signaled by another process based on some condition becoming true. Conditions are usually used in producer-consumer type applications where the consumer waits until an item becomes available by the producer, and then consumes it. Conditions support the following methods:

- `c.acquire()` : obtain an internal lock. The process is blocked until the lock is available
- `c.notify()` : wake up one of the processes waiting (if there is one waiting)
- `c.notifyAll()` : wake up all waiting processes
- `c.release()` : release the internal lock
- `c.wait()` : release the lock and then block until awakened by notify()

Queues
The multiprocessing `Queue` module provides a first-in-first-out (FIFO) type queue structure so that different processes can exchange data. In a queue, data is put from one end is removed from the other end. For example, a process can put data into a queue and another process can extract and use this data. Queues can contain any type of data, including strings, integer, floating-point numbers, lists, dictionaries, and so on.
Queues must be created before they are used. Function `put(data)` puts data into the queue. Function `get()` gets data from the queue.

The size of a Queue can be specified as an argument when the queue is created. Queues have the following methods

- `qsize()` : returns the size of the queue
- `empty()` : returns True if the queue s empty
- `full()` : returns True if the queue is full
- `put(obj[,block[,timeout]])` : put obj into the queue. If option block is True (default) and timeout is Node (default), the queue will block until a free slot is available. If timeout is a positive number, the queue will block at most timeout seconds and raise the queue.Full exception if no free slot was available within that time.
- `put_nowait(obj)` : put obj into queue (same as above when block is False)
- `get([block[,timeout]])` : remove and return an item from the queue. If block is True (default) and timeout is None (default), the queue will block until an item is available. If timeout is a positive number, the queue blocks at most timeout seconds and raises the queue.Empty exception if no item was available within that time.
- `get_no_wait()` : remove and return an item from the queue (same as above when block is False)

Additionally, SimpleQueue is supported by the following basic methods

- `put(item)` : put item into queue
- `get()` : remove and return item from queue
- `empty()` : return True if the queue is empty

Sharing data using Value and Array
The multiprocessing module offers shared memory variables called `Value` and `Array`. Data stored in these variables can be made to be common to all processes running in the system.

Anonymous Pipes
Anonymous pipes are used to establish interprocess communication and data exchange between processes. A pipe is like a shared memory buffer where one process puts data from one end and another process gets this data. Pipes are blocking. For example, a call to a pipe to read data will block the calling process until data is available.

Named Pipes
Named pipes are like files where they are opened by their names and data is written to or read from them as if they are files. A named file is created using the `os.mkfifo()` system call.

Signals
Signals are used to trigger handlers in user programs. When a signal occurs the handler can be programmed to be activated automatically. The function `signal(number, handler)` is used to create a handler object. Here, number is the handler number assigned by the programmer. Handler is the function to be activated when the signal occurs. The program can be forced to sleep and wait for the signal to occur by calling to function `signal.pause()`.

Examples

Basic processes : Two LEDs flashing at different rates
Two LEDs are connected to the Raspberry Pi. One of the LEDs flashes every second while the other one flashes every 250 milliseconds.
The LEDs are connected to Raspberry Pi GPIO 2 and GPIO 3 through 470 Ohm current limiting resistors.

After importing the modules used in the program, LED1 and LED2 are assigned to 2 and 3 which correspond to GPIO 2 and GPIO 3. Process `Flash1000` flashes LED1 every second, while process `Flash250` flashes LED2 every 250 milliseconds. Notice that processes `Flash1000` and `Flash250` are given the names `Flash1000` and `Flash250` respectively when they are created. Process `Flash250` displays its name and process ID as soon as it runs. The two processes are started with function call `start()`.

```
import multiprocessing
import os
import RPi.GPIO as GPIO
import time

# Disable warnings
GPIO.setwarnings(False)
# init LEDs
LED1 = 2 # LED1 at GPIO 2
LED2 = 3 # LED2 at GPIO 3

GPIO.setmode(GPIO.BCM)
GPIO.setup(LED1, GPIO.OUT)
GPIO.setup(LED2, GPIO.OUT)

# Process Flash100
def Flash1000():
  # Flash the LED
  while True:
    GPIO.output(LED1, 1) # LED ON
    time.sleep(1)        # Wait 1 second
    GPIO.output(LED1, 0) # LED OFF
    time.sleep(1)        # Wait 1 second

# Process Flash250
def Flash250():
  myname = multiprocessing.current_process().name
  print("Process name=%s" % myname)
  myid = os.getpid()
  print("Process id=%d" %myid)
  while True:
    GPIO.output(LED2, 1) # LED ON
    time.sleep(0.25)     # Wait 250ms
    GPIO.output(LED2, 0) # LED OFF
    time.sleep(0.25)     # Wait 250ms

p = multiprocessing.Process(name="Flash1000", target = Flash1000, args = ())
q = multiprocessing.Process(name="Flash250",  target = Flash250,  args = ())
p.start()
q.start()
```

Using queue : Setting the LED flashing rate from the keyboard

At the beginning of the program, a Queue with the name q is created. The LED is assigned to GPIO 2 and this port is configured as an output. The flashing rate is set to 1 second by sending 1 to the queue using function `put(1)`. Process `Flash` checks whether the queue is empty and if not the new flashing rate is read from the queue using function `get()`. The LED then flashes at this rate. The main program creates process `Flash` and starts it. The input function is then used to read the required flashing rate from the keyboard which is then sent to the queue.

```
import multiprocessing
import RPi.GPIO as GPIO
import time

# Create queue
q = multiprocessing.Queue()

# Disable warnings
GPIO.setwarnings(False)
LED = 2 # LED1 at GPIO 2
GPIO.setmode(GPIO.BCM)
GPIO.setup(LED, GPIO.OUT)

# Default rate
q.put(1)

# Process Flash
def Flash():
  while True:
    if not q.empty():   # If queue not empty
      rate = q.get()    # Get flashing rate
    GPIO.output(LED, 1) # LED ON
    time.sleep(rate)    # Wait 1 second
    GPIO.output(LED, 0) # LED OFF
    time.sleep(rate)    # Wait 1 second

# Start process Flash
p = multiprocessing.Process(target = Flash, args = ())
p.start()

# Input the flashing rate
while True:
   flash_rate = float(input("Enter flashing rate: "))
   q.put(flash_rate)
```

Sometimes we may want to check whether the queue is full or empty, or to detect if an error occurs when we want to put items to a full queue, or to read items from an empty queue. The following exception can be used to check when the queue is full and take the required actions:

```
try:
q.put(item) # Put item into queue
except q.Full:
# code to take action if the queue is full # Queue is full
```

or,

```
try:
d = q.get() # Get item from queue
except q.Empty:
# code to take action if the queue is empty # Queue is empty
```

Using Event : Reaction timer

This is a reaction timer project which makes use of an LED and a push-button switch. The user is expected to press the push-button switch as soon as the LED is turned ON. The time between the LED being turned ON and the user pressing the button is measured and displayed on an LCD in seconds. The LED is turned ON again after a random delay, ready for the next measurement.

The LED and the button are connected to GPIO 21 and GPIO 20 respectively. The I2C LCD is connected to GPIO 2 and GPIO 3 SA and SCL pins of the Raspberry Pi as in the previous LCD projects.

At the beginning of the program, the modules used in the program are imported. Notice module random is used to generate random numbers which are then used to generate random delays. Two events are created in the program with names e and t . Button (PB) is assigned number 20 and this port is configured as input.
The program consists of a process called LED_ON. This process configures the LED port as output and turns OFF the LED. The remainder of this process is executed in an endless loop. The LCD is controlled in the main program. Here the program waits until the LED is turned ON and then starts a timer. When the button is pressed, the timer reading is read and the elapsed time is calculated and displayed as the reaction time of the user.

```
import RPi.GPIO as GPIO
import multiprocessing
import random
import time
import RPi_I2C_driver

LCD = RPi_I2C_driver.lcd()

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BCM)
e = multiprocessing.Event()
t = multiprocessing.Event()

PB = 20 # Button at GPIO 20
GPIO.setup(PB, GPIO.IN)


# Process to turn ON the LED
def LED_ON():
 LED = 21                  # LED at GPIO 21
 GPIO.setup(LED, GPIO.OUT) # LED is output
 GPIO.output(LED, 0)
 while True:
   GPIO.output(LED, 0)       # LED OFF
   r = random.randint(1,10) # Generate random no
   time.sleep(r)             # Wait random seconds
   GPIO.output(LED, 1)       # LED ON
   e.set()                   # Set efn e
   t.wait()                  # Wait for efn t
   t.clear()                 # Clear efn t
   GPIO.output(LED, 0)       # LED OFF
   time.sleep(5)             # Wait and repeat

# Create the process
p = multiprocessing.Process(target = LED_ON, args = ())
p.start()
```

```
# LCD Display control. Display the reaction time in seconds
LCD.lcd_clear()                                 # Clear LCD
LCD.lcd_display_string("REACTION TIMER", 1) # Heading
while True:                                     # DO forever
  e.wait()                                      # Wait for event flag
  e.clear()                                     # Clear event flag
  TimeStart = time.time()                       # Start time
  while GPIO.input(PB) == 1:                    # Button not pressed
    pass

TimeEnd = time.time()                           # End time
t.set()
ReactionTime = str(TimeEnd - TimeStart)[:6] + " secs"
LCD.lcd_display_string(ReactionTime, 2)     # Display reaction time
```

Setting the flashing rate of an LED with keypad

In this project, an LED, keypad, and LCD are connected to the Raspberry Pi. The flashing rate of the LED is set using the keypad. This project aims to show how a keypad can be used in a multitasking environment. Like the 7-segment displays, keypads are ideal applications for multitasking.

Keypads are used in many microcontroller-based applications since they are small, portable, and do not require any external power supplies.

The Keypad: Several types of keypads can be used in microcontroller based projects. In this project, a 4x4 keypad is used. This keypad has keys for numbers 0 to 9 and letters A,B,C,D,*, and #. The keypad is interfaced to the processor with 8 wires with the names R1 to R4 and C1 to C4, representing the rows and columns respectively of the keypad.

The operation of the keypad is very simple: the columns are configured as outputs and the rows as inputs. The key pressed is identified by using column scanning. Here, a column is forced low while the other columns are held high. Then the state of each row is scanned, and if a row is found to be low, the key at the intersection of the row (which is low) and this column is the key pressed. This process is repeated for all rows.
The I2C LCD is connected to the Raspberry Pi as in the previous projects using the LCD, where GPIO 2 and GPIO 3 are used as the SDA and SCL pins respectively. The LED is connected to GPIO 21 of the Raspberry Pi. The 4x4 keypad is connected to the following GPIO pins of the Raspberry Pi. The row pins are held high using 10K pull-up resistors to +3.3V. Notice that Raspberry Pi has internal pull-up resistors when a pin is used as an input

Keypad pin Raspberry Pi pin

```
R1 GPIO 14
R2 GPIO 15
R3 GPIO 12
R4 GPIO 23
C1 GPIO 24
C2 GPIO 2
C3 GPIO 8
C4 GPIO 7
```

Test program
Before writing the project program we will, first of all, develop the code to read keys from the keypad. The basic steps to read a key are as follows

```
Configure all columns as outputs
Configure all rows as inputs
Set all columns to 1
DO for all columns
  Set a column to 0
  DO for all rows
    IF a row is 0 THEN
      Return the key at this column and row position
    ENDIF
  ENDDO
ENDDO
```

At the beginning of the program the keypad keys are defined after importing the required modules to the program. The keypad row and columns connections are defined using lists ROWS and COLS respectively. Columns are then configured as outputs and are set to 1. Similarly, the rows are configured as inputs. Function Get_Key reads the pressed key and returns it to the calling program. Two for loops are used in the function: the first loop selects the columns and sets them to 0 one after the other one. The second loop scans the rows and checks if a row is at 0. The main program calls the function and displays the pressed key on the screen.

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

# Keypad keys
KEYPAD = [
    [1,2,3,"A"],
    [4,5,6,"B"],
    [7,8,9,"C"],
    ["*",0,"#","D"]]

# Column pins
COLS = [24,25,8,7]
# Conf columns
for i in range(4):
   GPIO.setup(COLS[i], GPIO.OUT)
   GPIO.output(COLS[i], 1)

# Row pins
ROWS = [14,15,12,23]
# Conf rows
for j in range(4):
   GPIO.setup(ROWS[j], GPIO.IN)

# This function reads a key from the keypad
def Get_Key():
  while True:
    for j in range(4):
      GPIO.output(COLS[j], 0)          # Set col j to 0
      for i in range(4):               # For all rows
        if GPIO.input(ROWS[i]) == 0:   # Row is 0?
          return (KEYPAD[i][j])        # Return key
        while GPIO.input(ROWS[i]) == 0:
           pass
      GPIO.output(COLS[j], 1)          # Col back to 1
    time.sleep(0.05)                   # Wait 0.05s

try:
  while True:
    key = Get_Key() # Get a key
    print(key)      # Display the key
    time.sleep(0.5)

except KeyboardInterrupt:
  GPIO.cleanup()
```

In this program, key D is assumed to be the ENTER key where all inputs to the keypad must be terminated by pressing the ENTER key. There is one process in the program called `FLASH`. This process flashes the LED at a rate set by variable dly (the default value of dly is set to one second). The flashing rate is extracted from the queue and is loaded into variable dly.
The main program controls the LCD and keypad to receive the required flashing rate. The top row of the LCD shows the text `Flash rate (ms):` The program runs in an endless loop where the keys entered by the user are read and the required total delay is calculated and stored in variable Total until the ENTER key is pressed. The LCD shows each number entered by the user in the second row and when the ENTER key is pressed, the required flashing rate is calculated in seconds by dividing the number read by 1000. This number (in variable `tim`) is then put into the queue so that it can be extracted by process `FLASH` to change the flashing rate. The LED SET text is then displayed for 2 seconds to confirm the entered number has been accepted and the flashing rate has been changed. After 2 seconds, the second row of the LCD is cleared, ready for the next entry.

```
import RPi.GPIO as GPIO
import time
import multiprocessing
import RPi_I2C_driver

LCD = RPi_I2C_driver.lcd()
q = multiprocessing.Queue() # Create queue
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

# Keypad keys
KEYPAD = [
    [1,2,3,"A"],
    [4,5,6,"B"],
    [7,8,9,"C"],
    ["*",0,"#","D"]]

# Column pins
COLS = [24,25,8,7]
# Conf columns
for i in range(4):
  GPIO.setup(COLS[i], GPIO.OUT)
  GPIO.output(COLS[i], 1)
# Row pins
ROWS = [14,15,12,23]
# Conf rows
for j in range(4):
  GPIO.setup(ROWS[j], GPIO.IN)

# This function reads a key from the keypad
def Get_Key():
  while True:
    for j in range(4):
      GPIO.output(COLS[j], 0)          # Set col j to 0
      for i in range(4):               # For all rows
        if GPIO.input(ROWS[i]) == 0:   # Row is 0?
          return (KEYPAD[i][j])        # Return key
        while GPIO.input(ROWS[i]) == 0:
           pass
      GPIO.output(COLS[j], 1)          # Col back to 1
    time.sleep(0.05)                 # Wait 0.05s

# This process flashes the LED at the rate dly which is
# entered from the keyboard
def FLASH():
  LED = 21
  GPIO.setup(LED, GPIO.OUT)
  dly = 1 # 1 second (default)

  while True:             # Do forever
    if not q.empty():
      dly = q.get()       # Get flashing rate
    GPIO.output(LED, 1)   # LED ON
    time.sleep(dly)       # Wait dly sec
    GPIO.output(LED, 0)   # LED OFF
    time.sleep(dly)       # Wait dly sec


p = multiprocessing.Process(target = FLASH, args = ())
p.start()
```

```
# Main program. Here the flashing rate is read from the keypad
# and displayed on the LCD and is then sent to process FLASH
LCD.lcd_clear()
LCD.lcd_display_string("Flash rate (ms):", 1)
Total = 0
while True:
  key = Get_Key()                          # Get a key
  if key != "D":                           # If not ENTER
    LCD.lcd_display_string(str(key), 2)    # Display
    N = int(key)                           # In integer
    Total = 10*Total + N                   # Total number
    LCD.lcd_display_string(str(Total), 2) # Display
 else:                                     # If ENTER
   tim = Total / 1000                      # In ms
   q.put(tim)                              # In queue
   LCD.lcd_display_string("LED SET", 2)    # Message
   time.sleep(2)                           # Wait 2 sec
   LCD.lcd_display_string(" ", 2)          # Clear
   Total = 0                               # Total to 0
 time.sleep(0.5)                           # Wait 0.5s
```