Universidade de São Paulo
Escola de Engenharia de São Carlos
Departamento de Engenharia Elétrica e de Computação

Prática 5: Introdução a programação de periféricos com tratamento de interrupções e uso de threads

Resumo

Introdução ao uso de periféricos embarcados na raspberry com interrupções, timers e threads a partir da programação em Python, com bibliotecas de manipulação de pinos "RPI.GPIO", "*time*", e "*threading*".

Conceitos importantes:

GPIOs, *threading, timer, polling, interrupt, contadores, parallel computing,* CPU *multicore*.

Objetivo

O objetivo desta prática é a familiarização do uso de conexões externas a rasp, a partir de seus pinos de entrada e saída de uso geral, além disso, do uso de temporização, a partir de delays e do uso de *threads*.

Em Python, diferente do que foi abordado para microcontroladores em disciplinas anteriores, não existe um "*timer* interno". Isto se deve ao fato que sistemas operacionais não possuem a responsabilidade com o tempo real. Ademais, aplicações de executando kernel Linux não são adequadas para aplicações críticas de tempo real, já que as funções tempo são organizadas junto às outras entradas da CPU, sua ordem é organizada pelo próprio kernel do sistema e suas bibliotecas, a definir pela prioridade dada a cada atividade.

Porém, pode-se obter sistemas de uso em "tempo quase real", chamados de *soft real time*, a partir do uso de *threads* no sistema operacional. Desta forma, como o processador possui mais de um núcleo (no caso da Rasp modelo 3B+, são 4 núcleos), pode-se dedicar a atividade de um núcleo a executar uma tarefa. Assim, os restantes seguem sendo controlados pela ordem de prioridade do sistema, e um deles, porém, é mantido sob prioridade maior do comando passado. Processos e threads advém do conceito de *parallel computing* e CPU *multicore* (Fig. 1).

- **Processos:** qualquer programa rodando em um sistema operacional possui um mais processos associados!
- ***Thread*:** uma unidade de execução de um processo - logo um processo pode consistir de uma ou várias threads (multithreading!).
- **CPU multicore:** possui 2 ou mais núcleos de processamento.

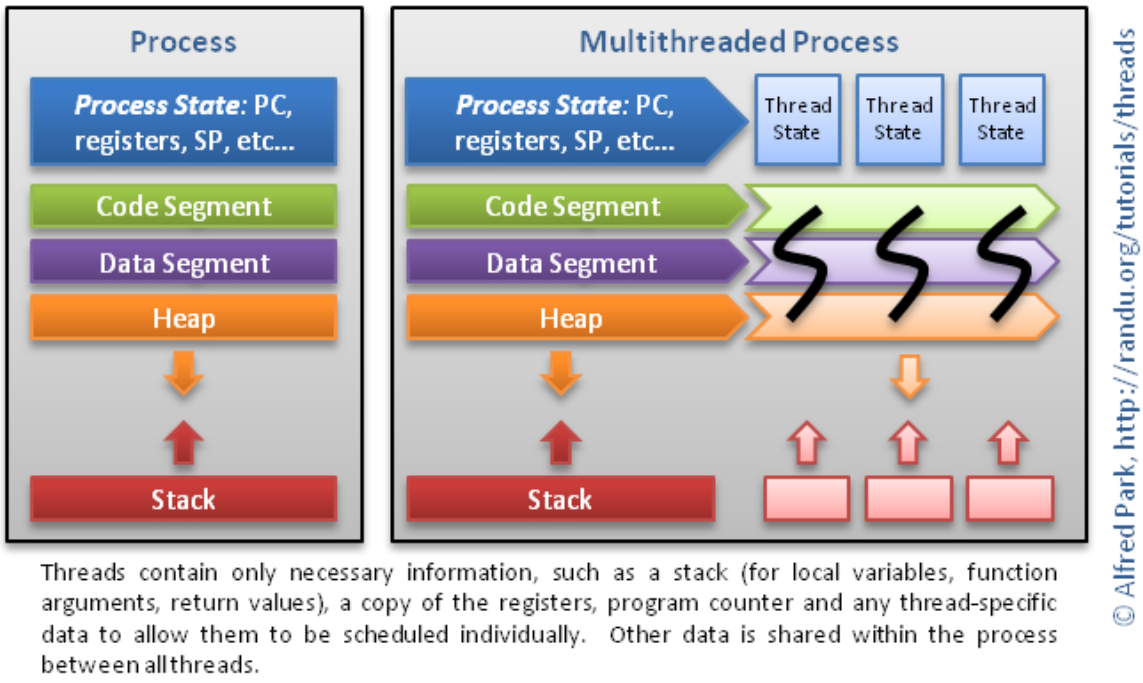

Fig. 1 - Processos, threads e conceito de CPU multicore. Fonte: https://randu.org/tutorials/threads/

Exemplo 1

```
import threading
dir(threading)
import time

def timer(t):
   print('Starting {} timer'.format(threading.current_thread().name)
   time.sleep(t)
   print('Finished {} timer'.format(threading.current_thread().name))
timer(2)

for i in range(5):
   a = threading.Thread(target=timer, args=(i,))
   a.start()

for i in range(5):
  a=threading.Thread(target=timer, args=(i,))
  a.start()
  a.join()
```

Execute as linhas de código acima no terminal Shell da IDE Thonny Python e verifique quantas threads iniciam e terminam após o comando “a.join()” .

Exemplo 2

```
import RPi.GPIO as GPIO
import time
LED_vermelho = 12
LED_azul = 6
GPIO.setmode(GPIO.BCM)
GPIO.setup(Rled,GPIO.OUT)
GPIO.setup(Bled,GPIO.OUT)

def blue():
  while True:
  print "LED azul aceso" GPIO.output(LED_azul,GPIO.LOW)
  time.sleep(1)
  print "LED azul apagado" GPIO.output(LED_azul,GPIO.HIGH)
  time.sleep(1)

def red():
  while True:
  print "LED vermelho aceso" GPIO.output(LED_vermelho,GPIO.LOW)
  time.sleep(1)
  print "LED vermelho apagado" GPIO.output(LED_vermelho,GPIO.HIGH)
  time.sleep(1)

blue()
red()
```

No código acima, devido ao loop infinito para ambos os métodos, o LED azul () não irá apagar e o vermelho () também não irá iniciar.

Exemplo 3

```
import threading

# < código do exemplo 2 aqui >

t1 = threading.Thread(target = blue)
t2 = threading.Thread(target = red)

t1.start()
t2.start()
```

Solução: separando os blocos do código do Exemplo 2 em threads usando biblioteca “theading” em Python.

Assim como *threads* e *timers* permitem que possa-se executar uma tarefa enquanto outras atividades acontecem (neste caso, conta-se o tempo enquanto o restante do programa continua a executar sua tarefa principal), o mesmo pode ser feito com a leitura de pinos de entrada, sendo este processo de identificação do estado de pinos de entrada sem comprometimento do programa principal, denominado *interrup*t, ou interrupção (Fig. 2). O mecanismo de interrupção é a base para viabilização do conceito de CPU multicore. Portanto, para controlar entrada e saída de dados, por exemplo, ao invés da CPU ficar continuamente monitorando o status de um dispositivo visando detectar uma mudança de estado (evento), a interrupção permite chamar a atenção do hardware ,

comunicando a ocorrência de um evento, para suspender suas atividades e atender exclusivamente a rotina do evento que o interrompeu (tratamento da interrupção).

Cumpre frisar que o uso de interrupções não é algo disponível em qualquer sistema, uma vez que pinos de entrada são existentes somente em sistemas embarcados. Sua função não está implementada na biblioteca padrão do Python, mas sim nas bibliotecas específicas, responsáveis pelo controle dos pinos de entrada e saída (GPIO) já vistos em práticas anteriores. Com esta biblioteca pode-se definir quais pinos estarão em interrupção, e qual função eles irão chamar ao mudarem de estado. Também pode ser definida a sua sensibilidade a um tipo específico de borda de ativação. A documentação da biblioteca (RPi.GPIO), possui mais detalhes dos argumentos das funções de interrupção, além da existência de diversos tutoriais na internet a respeito das duas funções explicadas anteriormente.

Fig. 2 - Disparo de interrupção a partir da mudança de estados

Exemplo 4

```
# exemplo de uso da biblioteca "Rpi.GPIO" com funções callback quando a interrupção é
# acionada. Função especial da GPIO: "add_event_detected"

def button_callback(channel):
    print("Botão pressionado!")

    # A função button_callback() verifica o estado do botão
    # (após a interrupção ter sido acionada) e poderia ser usada para a lógica:
    # Se botão pressionado (“GPIO.input(button_GPIO”), então: mudar freq. do LED

# O bloco seguinte é destinado a interpretação da interrupção,

    GPIO.setmode(GPIO.BCM)
    GPIO.setup(BUTTON_GPIO, GPIO.IN, pull_up_down=GPIO.PUD_UP)

    GPIO.add_event_detect(BUTTON_GPIO, GPIO.FALLING, # ou GPIO.RISING ou GPIO.BOTH…
            callback=button_callback, bouncetime=100)

    # A função "add_event_detect" pertence à lib “RPi.GPIO” e irá acionar
    # button_callback()assim que o sinal GPIO do botão estiver em transição
    # "falling" (HIGH para LOW),com um tempo de debounce de 100 milissegundos usado
```

```
#como exemplo.

#-------------------------------------------------------------------------------
# Parâmetros da função:

# Canal: número GPIO (modo BCM)

# Tipo de interrupção: "GPIO.FALLING", "GPIO.RISING", ou "GPIO.BOTH".

# Callback (opcional): chamada quando uma interrupção é acionada
#  (em outras situações o retorno pode ser chamado por outra função
#  "add_event_callback").

# Bouncetime, em milissegundos (opcional): no caso de várias interrupções
#   serem acionadas em um curto período de tempo - devido ao retorno do botão -
#   o retorno a será chamado apenas uma vez, e outras vezes em que, um botão for
#   pressionado, por exemplo, não serão consideradas pelo tempo em ms informado
```

Utilizando a biblioteca “Rpi.GPIO” com funções callback quando a interrupção é acionada. Função especial da GPIO: “add_event_detected”. Exemplo com a interrupção “falling”.

Aplicação - tarefa prática

Portanto, para a prática, deve-se executar um script em python, que seja capaz de realizar uma contagem de tempo, a partir de uma *thread* separada (threading.Timer), e após a contagem do tempo, deve-se chamar uma função de callback que imprima na tela o resultado do fim da contagem de tempo. Além disso, enquanto a contagem é feita, o código não pode ser mantido em *polling* (quando a CPU ficaria verificando se algum periférico pretende reportar algum evento), pois o código durante a contagem de tempo deve realizar o *blink* de um LED conectado a protoboard , com duas frequências diferentes, e a alteração destas frequências deve ser feita a partir de um botão conectado a rasp (vide Fig. 2).

A leitura do botão também deve ser feita de forma a não interromper o código principal de *blink* do LED e, para isto, deve-se utilizar interrupções para leitura do botão, que também deve chamar uma função de “*callback”* para alterar uma variável que define a frequência.

Ao final da contagem de tempo do *timer*, deve-se atentar a desconectar todas as GPIOs do sistema, a partir do comando em python definido pela biblioteca, caso contrário as GPIOs vão se manter em estado ativo mesmo após o fechamento do script, o que pode causar danos a placa por curto circuitos.

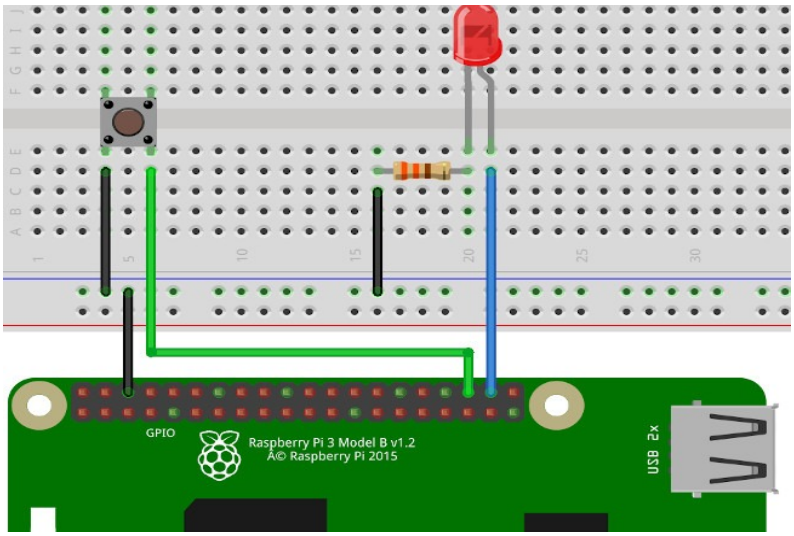

Fig. 3 - O pinout com descrição e numeração de cada um dos pinos pode ser consultado a partir da internet a partir do site de pinout da rasp.

Motivação

O uso de *Threadings* em Python é um conceito extremamente importante, não somente para o uso de *timers*, mas para o uso de múltiplas tarefas concorrentes. Dessa forma, não é necessário que se faça múltiplos scripts para funções distintas, além de permitir que os dados não tenham que trafegar entre scripts, o que aumenta a robustez e segurança de sistemas embarcados. Além disso, o conhecimento de ferramentas que permitem que o código possa ser executado sem a parada do código principal é de extrema importância para atividades que exigem responsividade ou que dependam do tempo para serem executadas. É de importância enfatizar que no âmbito de sistemas embarcados, a maioria das aplicações exige que se mantenha alguma atividade sendo executada enquanto se verifica o restante, desde sistemas mais simples, como teclados matriciais em eletrodomésticos (os quais devem sempre estar prontos para leitura de uma entrada do usuário, e não esperar que o usuário clique no botão no momento em que a leitura está sendo feita), até sistemas mais complexos, como máquinas industriais (que não podem interromper seu processo para verificar o acionamento ou entradas de sensores, assim como não podem ficar parada em um delay esperando o fim de um processo, sem que mais nada funcione). Estes são somente alguns exemplos da importância do uso destas ferramentas para o desenvolvimento de sistemas robustos e seguros.

Roteiro

- Atente-se a conexão da placa (vide Fig. 3), utilizando os pinos corretos, caso algum dos pinos já esteja em uso na placa, como um ground, utilize outro seguindo a pinagem disponibilizada. Dê preferência a utilizar o ground para realizar a interrupção no botão.
- Verifique se os estados iniciais do botão e do LED estão corretos e não ocasionarão nenhum curto circuito na placa.
- Importe as bibliotecas *threading* e RPi.GPIO para uso no projeto.
- As funções de *timer* e interrupção por botão já estão desenvolvidas nestas bibliotecas, veja a documentação (vide material de aula: Cap.3 -item 3.3) ou exemplos na internet.

- Utilize as funções de *callback* para manipular flags, que serão utilizadas pelo código principal, evitando, desta forma, que o programa fique muito tempo dentro de uma função de interrupção.
- Atente-se de aplicar um "*bounce time*" no botão, uma vez que não está sendo utilizado um *debounce* em hardware no projeto.
- **Entregas:** enviar o script em python ".py" que implementa o programa solicitado e o relatório documentando a prática (relatório breve e objetivo de ~ 5 páginas com introdução, desenvolvimento e conclusão, contendo explanações sobre os blocos do código, funções e lógica usada).
- Bom projeto! e em caso de dúvidas contate o professor ou monitor da disciplina.

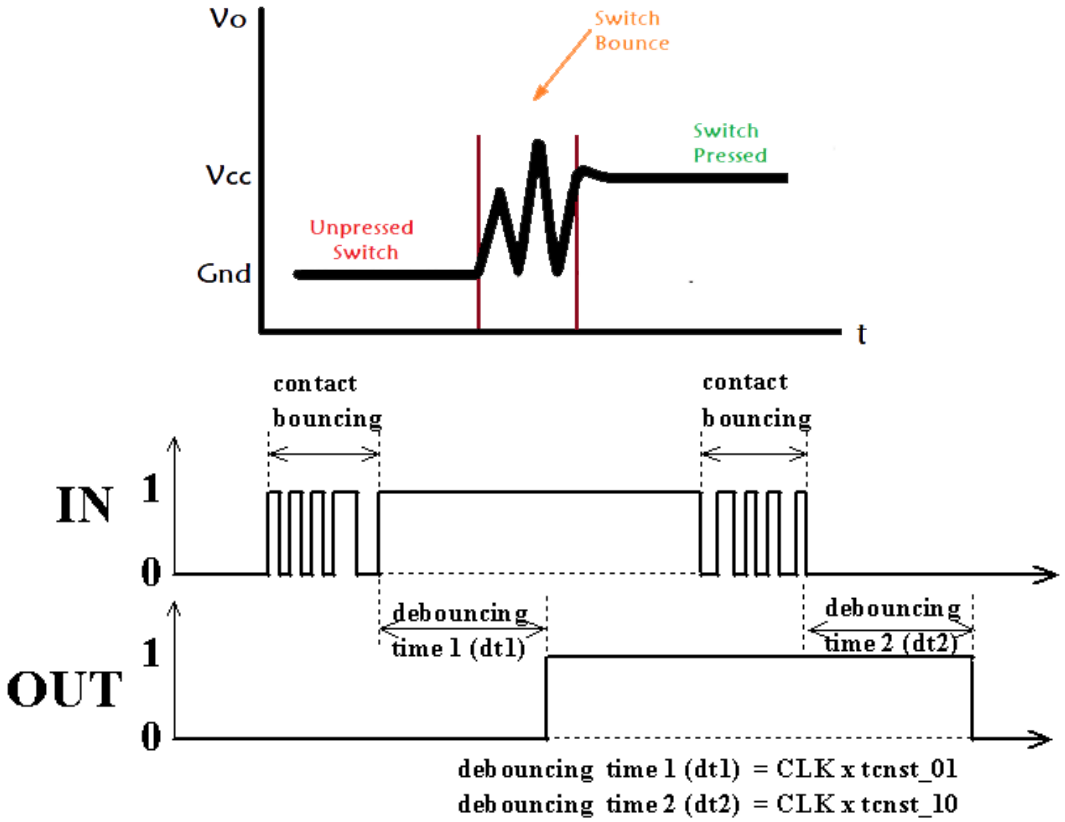

Fig. 4 - Ilustração do conceito de bounce e debounce time