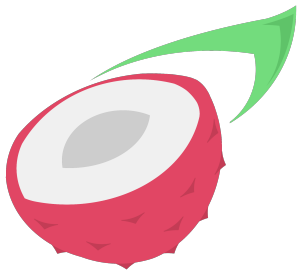

Litchi Pi

Python

Multithreading

6 Novembre 2025

Introduction

Comment faire fonctionner 120 programmes sur 8 coeurs ?

Kernel Linux a un **scheduler**, démarré au lancement de l'ordi, qui va gérer les **processus**

Un **processus** est un ensemble de **threads** qui partagent un même *contexte* d'éxecution

Pour gérer 120 threads, le kernel Linux va switcher entre eux, leur **allouant du temps** de CPU

Plus le thread est *prioritaire*, plus il aura de temps par rapport aux autres

Pour cela, va créer un *hardware timer*, qui va générer une **interruption** après `N` cycle d'horloge

L'interruption stoppe le programme en cours, *sauvegarde le contexte*, et appelle le scheduler

Le scheduler va regarder quelle tâche va prendre la suite, *setup le contexte*, et l'éxecute

Some more ressources

- *StackOverflow - How Linux handles threads and process scheduling ?*
- *Dev.to - A brief history of the linux kernel's process scheduler*

Créer un thread

```python
from threading import Thread
import time

def my_function(x, option=3):
  while True:
    print("toto", x, option)
    time.sleep(1)

thread = threading.Thread(target=my_function, args=(3,), kwargs={"option": 5})
thread.start()
```

```python
def my_function(x, option=3):
  pass

thread = threading.Thread(target=my_function, args=(3,), kwargs={"option": 5})
```

Wrapper un thread avec une classe

```
from threading import Thread

class MyTask(Thread):
  def __init__(self):
    Thread.__init__(self)

  def run(self):
    while True:
      print("BOUCLE INFINIE")

task = MyTask()
task.start()
task.join()
```

L'objet `Thread` permet de contrôller le statut du thread créé

- `Thread.is_alive()`, permet de savoir si le thread est terminé, ou en cours d'éxecution
- `Thread.join()`, permet d'attendre la fin de l'exécution du thread

Et c'est tout.

On ne peut pas *communiquer des données*

On ne peut pas *terminer* le thread manuellement

On ne peut pas *inspecter* la mémoire du thread, savoir ce qui se passe

Comment t'est-ce qu'on fait donc pour le contrôller ?

Communiquer avec un thread

Même processus = mémoire *partagée*

Une même variable partagée entre 120 threads = **Problèmes**

Problèmes très durs à débugger, et tout arrive *en même temps*

Bienvenue dans le monde de la programmation parallèle.

Demande un soin particulier à:

- l'**architecture** du code (qui fait quoi, où, comment)
- la **communication** des informations & du débug
- l'identification des *ressources critiques*
- la *sécurisation* des ressources critiques

Partager un booléen: `Event`

Booléen partagé par tous les threads, peut `set` (=true), `clear` (=false) et `wait` (attendre `true`)

```
from threading import Event, Thread
import time

def thread(ev, ev2):
  while ev2.is_set():
     ev.wait()
     ev.clear()
     print("TOTO")
  print("Stopped")

ev = Event()
ev2 = Event()
ev2.set()

thread = Thread(target=thread, args=(ev, ev2))
thread.start()

for _ in range(5):
  time.sleep(1)
  ev.set()
```

S'envoyer des messages: `Queue`

Canal d'où on peut envoyer avec `put`, et recevoir avec `get`, n'importe quel objet Python

```
from threading import Thread
from queue import Queue
import time

def thread(q):
  while True:
    data = q.get()
    if data == False:
      break
    else:
      print("Got", data)
  print("Finished")

q = Queue()
thread = Thread(target=thread, args=(q,))
thread.start()

q.put("toto")
q.put([3, 2, 1])
q.put(False)
```

Protéger une mémoire partagée: `Lock`

Permet de s'assurer qu'une ressource n'est accessible que par **un thread à la fois**
On se place dans la file d'attente avec `acquire`, on laisse notre place avec `release`

```python
from threading import Thread, Lock
MY_GLOBAL_VARIABLE = 3
VARIABLE_LOCK = Lock()

def set_variable(new_value):
  VARIABLE_LOCK.acquire(True) # True = Bloquer ici si pas dispo de suite
  MY_GLOBAL_VARIABLE = new_value
  VARIABLE_LOCK.release()

def thread_set_new_value():
  for n in range(1000):
    set_variable(n)

for _ in range(15):
  thread = Thread(target=thread_set_new_value)
  thread.start()
```

Dans le même genre, le `Semaphore`, permet `N` threads à la fois

Attendre que tous les threads soient au même point: `Barrier`

Appeler `wait` quand on a atteint le checkpoint

Une fois que tous les threads concernés ont appelé `wait`, tout le monde est libéré

```
from threading import Thread, Barrier
import time, random

def my_thread(barrier, tid):
  time.sleep(random.randrange(1, 10))
  print(f"Thread {tid} waiting")
  barrier.wait()

barrier = Barrier(26)

for tid in range(25):
  thread = Thread(target=my_thread, args=(barrier, tid))
  thread.start()

barrier.wait()
print("Main thread finished")
```

TP

Paralleliser algorithme génétique