



ЖИТТЄВИЙ ЦИКЛ ПОТОКУ

Питання 12.2.

Потоки в Python

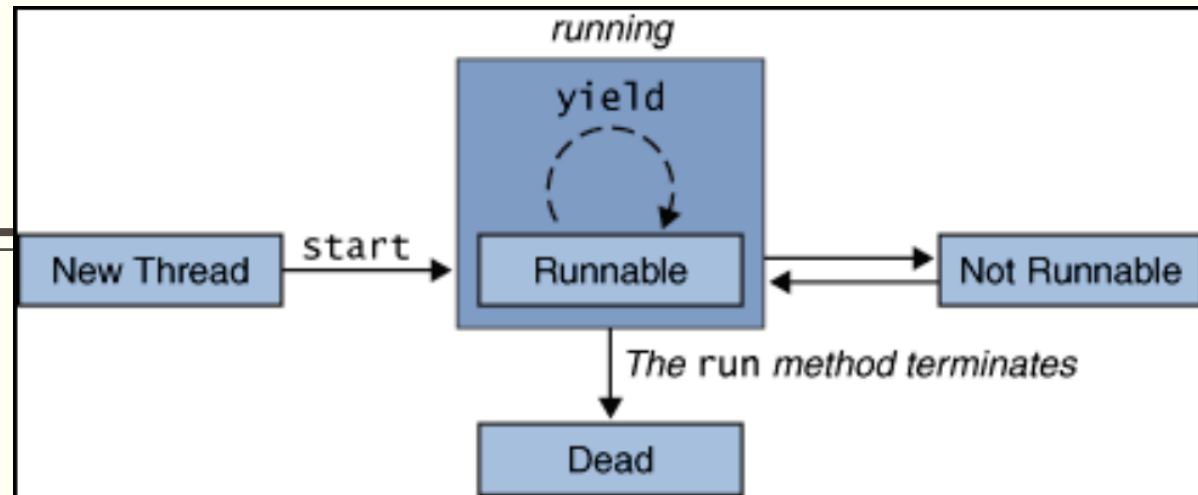
- Клас Thread у Python можна знайти в модулі `threading.py`.

- Ініціалізатор класу виглядає так:

```
# Python Thread class Constructor
def __init__(self, group=None, target=None, name=None,
             args=(), kwargs=None, verbose=None):
```

- *group*: спеціальний параметр, який зарезервовано для подальшого розширення.
 - *target*: callable-об'єкт, який викликається в методі `run()`. Без його передачі значення прирівнюється до `None`, нічого не буде запущено.
 - *name*: назва потоку.
 - *args*: кортеж аргументів для цільового виклику. За замовчуванням - `()`.
 - *kwargs*: словник keyword-аргументів to invoke the base class constructor.

Стани потоку



- **Новий потік (New Thread):**

- Потік ще не запущено, йому ще не виділено ресурсів. Просто об'єкт.

- **Готовий до виконання (Runnable):**

- Потік очікує запуску, має виділені всі необхідні ресурси, тільки планувальник стримує його від виконання.

- **Запущений (Running):**

- Потік виконує роботу. З цього стану він може перейти або в dead state, якщо буде вирішено «вбити» його, або в незапущений стан.

- **Незапущений (Not-running):**

- Потік певним чином поставлений на паузу. Причин цьому може бути багато: очікування завершення тривалої операції вводу-виводу, навмисне блокування з метою виконання роботи іншим потоком тощо.

- **Dead:**

- Досягти цього стану можна двома шляхами: потік «помре» природно або буде вбитим.

Приклад стану потоку

```
1 import threading
2 import time
3
4 # простий метод для виконання потоку
5 def threadWorker():
6     # тут потік починає виконання (стан 'Runnable' -> 'Running')
7     print("My Thread has entered the 'Running' State")
8
9     # Якщо викликаємо метод time.sleep(), наш потік переходить у стан not-runnable
10    # We can do no further work on this particular thread
11    time.sleep(10)
12    # Потік завершує роботу та переривається
13    print("My Thread is terminating")
14
15 # Тут потік не має стану, тому що для нього не виділено системних ресурсів
16 myThread = threading.Thread(target=threadWorker)
17
18 # Після виклику myThread.start(), Python виділяє необхідні ресурси
19 # для запуску потоку, а потім викликає метод run потоку.
20 # Перехід зі стану 'Starting' -> 'Runnable', протом not running
21 myThread.start()
22
23 # Після виклику методу join потік переходить у стан 'Dead'.
24 # Він завершив передбачену для нього роботу.
25 myThread.join()
26 print("My Thead has entered a 'Dead' state")
```

```
My Thread has entered the 'Running' State
My Thread is terminatingMy Thead has entered a 'Dead' state
```

Види потоків

- Python абстрагується від найбільш низькорівневої частини API, які працюють з потоками.
- Також мова дозволяє писати портативний код, обираючи залежно від ОС вид потоків, які будуть виконувати код:
 - **POSIX-потоки.** Реалізація потоків відповідно до стандарту IEEE POSIX 1003.1c. Він був розроблений для стандартизації реалізації потоків для різноманітного апаратного забезпечення під управлінням UNIX-систем. Часто такі потоки називають POSIX-потоками або PThreads.
 - **Windows-потоки.** Працюють відповідно до стандартів Microsoft та мають багато відмінностей щодо POSIX-потоків. Windows threads API простіший та елегантніший за POSIX threads API.

Способи запуску потоку

```
1 import threading
2 import time
3 import random
4
5 def executeThread(i):
6     print("Thread {} started".format(i))
7     sleepTime = random.randint(1,10)
8     time.sleep(sleepTime)
9     print("Thread {} finished executing".format(i))
10
11 for i in range(10):
12     thread = threading.Thread(target=executeThread, args=(i,))
13     thread.start()
14
15     print("Active Threads:" , threading.enumerate())
```

```
Thread 2 finished executing
Thread 4 finished executing
Thread 5 finished executing
Thread 1 finished executing
Thread 3 finished executing
Thread 8 finished executing
Thread 9 finished executing
Thread 6 finished executing
Thread 0 finished executing
Thread 7 finished executing
```

- Маємо просту функцію, яка «спить» протягом випадкового періоду часу.

Наслідування від класу потоку

- Для того, щоб створити новий потік, потрібно субкласувати вбудований клас з мови Python для роботи з потоками:
 - Передати в описі нашого класу, що описує потік, thread class мови Python
 - Викликати Thread.__init__(self) в конструкторі, щоб ініціалізувати потік.
 - Визначити функцію run(), яка буде викликатись, коли потік буде запущено.

```
1 from threading import Thread
2
3 class myWorkerThread(Thread):
4
5     def __init__(self):
6         print("Hello world")
7         Thread.__init__(self)
8
9     def run(self):
10        print("Thread is now running")
11
12 myThread = myWorkerThread()
13 print("Created my Thread Object")
14 myThread.start()
15 print("Started my thread")
16 myThread.join()
17 print("My Thread finished")
```

- У конструкторі викличемо необхідний Thread.__init__(self)
 - Далі оголошуємо функцію run(), яка буде викликатись після myThread.start().
 - У функції run() просто виведемо поточний стан на консоль.
 - Після цього потік завершить роботу.

```
Hello world
Created my Thread Object
Thread is now running

Started my thread
My Thread finished
```

Форкінг (Forking)

- Для форку (відгалуження) процесу потрібно створити точну копію вже існуючого.
 - Відбувається ефективно клонування та запуск клонованого процесу як дочірнього для батьківського.
 - Новостворений процес отримує власний адресний простір, а також точну копію батьківських даних і коду, який виконується в батьківському процесі.
 - Клон при створенні отримує унікальний **Process Identifier (PID)**, таким чином не залежачи від свого батьківського процесу.
- Коли виникає потреба клонувати існуючий процес?
 - Наприклад, Apache активно використовує форкінг, щоб створити багато серверних процесів.
 - Кожен з цих незалежних процесів здатен обробити *their own requests within their own address space*.
 - Якщо процес падає чи вмирає, вплив на інші процеси, які конкурентно з ним працюють, буде відсутнім.
 - Вони зможуть продовжити *to cater to any new requests*.

Приклад

```
import os
def child():
    print "We are in the child process with PID= %d"%os.getpid()
def parent():
    print "We are in the parent process with PID= %d"%os.getpid()
    newRef=os.getpid()
    if newRef==0:
        child()
    else:
        print "We are in the parent process and our child process has PID=
%d"%newRef
parent()
```

- У функції parent() виводиться PID поточного процесу перед викликом методу os.fork() для форку поточного запущеного процесу.
 - Це створює геть новий процес, який отримує унікальний PID.
- Далі викликається функція child(), яка виводить поточний PID.
 - Він відрізняється від початкового PID, виведеного на початку виконання скрипту.
 - Різний PID представляє успішний форкінг та створення повністю нового процесу.

Демонування (Daemonizing) потоку

- Потоки-демони (Daemon threads) – це потоки без визначеної точки свого завершення.
 - Вони продовжують роботу, поки програма не завершить свою.
 - Наприклад, маємо балансувальника навантаження, який відправляє service requests до багатьох екземплярів програми.
 - Може існувати деяка форма registry service, яка повідомляє балансувальнику, куди пересилати ці запити.
 - Зазвичай періодично надсилають heartbeat (keep alive packet).
- Цей приклад – основний сценарій використання потоків-демонів у додатку.
 - Можна передати роботу з відправки heartbeat-сигналу до service registry в потік- демон та start this up при запуску додатку.
 - Цей потік-демон у фоні періодично відправлятиме дане оновлення, не втручаючись в роботу основного потоку.
 - Потік-демон «вбивається» без нашого відома при закритті додатку.

Приклад

```
import threading
import time
def standardThread():
    print("Starting my Standard Thread")
    time.sleep(20)
    print("Ending my standard thread")
def daemonThread():
    while True:
        print("Sending Out Heartbeat Signal")
        time.sleep(2)
if __name__ == '__main__':
    standardThread = threading.Thread(target=standardThread)
    daemonThread = threading.Thread(target=daemonThread)
    daemonThread.setDaemon(True)
    daemonThread.start()

    standardThread.start()
```

Завантаження групи потоків

- Можна створити багато потоків, використовуючи GPS loop, а потім запускати їх всередині того ж GPS loop.
 - Визначимо функцію, яка прийматиме ціле число та «спатиме» випадковий період часу і друкуватиме час запуску та завершення.

```
import threading
import time
import random
def executeThread(i):
    print("Thread {} started".format(i))
    sleepTime = random.randint(1,10)
    time.sleep(sleepTime)
    print("Thread {} finished executing".format(i))
for i in range(10):
    thread = threading.Thread(target=executeThread, args=(i,))
    thread.start()
    print("Active Threads:" , threading.enumerate())
```

Вивід програми

```
$ python3.6 00_startingThread.py
Thread 0 started
Active Threads: [<_MainThread(MainThread, started 140735793988544)>,
<Thread(Thread-1, started 123145335930880)>]
Thread 1 started
Active Threads: [<_MainThread(MainThread, started 140735793988544)>,
<Thread(Thread-1, started 123145335930880)>, <Thread(Thread-2, started
123145341186048)>]
Thread 2 started
Active Threads: [<_MainThread(MainThread, started 140735793988544)>,
<Thread(Thread-1, started 123145335930880)>, <Thread(Thread-2, started
123145341186048)>, <Thread(Thread-3, started 123145346441216)>]
```

Сповільнення програм, які використовують потоки

```
import time
import random
import threading
def calculatePrimeFactors(n):
    primfac = []
    d = 2
    while d*d <= n:
        while (n % d) == 0:
            primfac.append(d)
            n //= d
        d += 1
    if n > 1:
        primfac.append(n)
    return primfac
```

- Запуск сотень потоків та їх спрямування на вирішення конкретної проблеми, скоріше за все, не покращить продуктивність додатку.

```
def executeProc():
    for i in range(1000):
        rand = random.randint(20000, 1000000000)
        print(calculatePrimeFactors(rand))
def main():
    print("Starting number crunching")
    t0 = time.time()
    threads = []
    for i in range(10):
        thread = threading.Thread(target=executeProc)
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()
    t1 = time.time()
    totalTime = t1 - t0
    print("Execution Time: {}".format(totalTime))
if __name__ == '__main__':
    main()
```

Порівняння результатів роботи

Single-threaded sample:	3.69 seconds
Multi-processing sample:	1.98 seconds
Multi-threaded sample	3.95 seconds

- Запуск багатьох потоків та їх спрямування на одну проблему тут спричиняє 7%-ве сповільнення в порівнянні з однопоточним режимом і майже 100%-ве прискорення в багатопроцесному режимі.

Отримування загальної кількості активних потоків

- Модуль `threading` включає спеціальну функцію

```
import threading
import time
import random
def myThread(i):
    print("Thread {}: started".format(i))
    time.sleep(random.randint(1,5))
    print("Thread {}: finished".format(i))
def main():
    for i in range(random.randint(2,50)):
        thread = threading.Thread(target=myThread, args=(i,))
        thread.start()
    time.sleep(4)
    print("Total Number of Active Threads:
    {}".format(threading.active_count()))
if __name__ == '__main__':
    main()
```


Отримання поточного потоку

```
import threading
import time
def threadTarget():
    print("Current Thread: {}".format(threading.current_thread()))
threads = []
for i in range(10):
    thread = threading.Thread(target=threadTarget)
    thread.start()
    threads.append(thread)
for thread in threads:
    thread.join()
```

```
Current Thread: <Thread(Thread-1, started 123145429614592)>
Current Thread: <Thread(Thread-2, started 123145429614592)>
Current Thread: <Thread(Thread-3, started 123145434869760)>
Current Thread: <Thread(Thread-4, started 123145429614592)>
Current Thread: <Thread(Thread-5, started 123145434869760)>
Current Thread: <Thread(Thread-6, started 123145429614592)>
Current Thread: <Thread(Thread-7, started 123145434869760)>
Current Thread: <Thread(Thread-8, started 123145429614592)>
Current Thread: <Thread(Thread-9, started 123145434869760)>
Current Thread: <Thread(Thread-10, started 123145429614592)>
```

Головний потік (Main thread)

```
import threading
import time
def myChildThread():
    print("Child Thread Starting")
    time.sleep(5)
    print("Current Thread -----")
    print(threading.current_thread())
    print("-----")
    print("Main Thread -----")
    print(threading.main_thread())
    print("-----")
    print("Child Thread Ending")
child = threading.Thread(target=myChildThread)
child.start()
child.join()
```

- Всі Python-програми запускають принаймні один (головний) потік.
 - У Python можна викликати функцію `main_thread()`, щоб отримати об'єкт головного потоку.

```
Child Thread Starting
Current Thread -----
<Thread(Thread-1, started 123145387503616)>
-----
Main Thread -----
<_MainThread(MainThread, started 140735793988544)>
-----
Child Thread Ending
```

- Всередині функції просто виводимо поточний потік і головний потік.
 - Потім створюємо thread-об'єкт, запускаємо його виконання та приєднуємо (`join`) його.

```
import threading
import time
import random
def myThread(i):
    print("Thread {}: started".format(i))
    time.sleep(random.randint(1,5))
    print("Thread {}: finished".format(i))
def main():
    for i in range(4):
        thread = threading.Thread(target=myThread, args=(i,))
        thread.start()
    print("Enumerating: {}".format(threading.enumerate()))
if __name__ == '__main__':
    main()
```

Перелічення всіх потоків

Python дозволяє опитати всі активні потоки, а потім перелічити їх для того, щоб отримати про них інформацію, правильно зупинити тощо.

```
Thread 0: started
Thread 1: started
Thread 2: started
Thread 3: started
Enumerating: [<_MainThread(MainThread, started 140735793988544)>,
<Thread(Thread-1, started 123145554595840)>, <Thread(Thread-2,
started 123145559851008)>, <Thread(Thread-3, started
123145565106176)>, <Thread(Thread-4, started 123145570361344)>]
Thread 2: finished
Thread 3: finished
Thread 0: finished
Thread 1: finished
```

Ідентифікація потоків

- Може спростити налагодження багатопоточних додатків.
- Для великої кількості потоків зручно об'єднувати їх у групи, якщо вони виконують різні задачі.
 - Наприклад, маємо додаток, який прослуховує зміни біржової ціни та намагається передбачити нову ціну.
 - Можемо мати дві окремі групи потоків: одна прослуховує, а інша – виконує обчислення.

```
import threading
import time
def myThread():
    print("Thread {} starting".format(threading.currentThread().getName()))
    time.sleep(10)
    print("Thread {} ending".format(threading.currentThread().getName()))
for i in range(4):
    threadName = "Thread-" + str(i)
    thread = threading.Thread(name=threadName, target=myThread)
    thread.start()
print("{}".format(threading.enumerate()))
```

Вивід програми

```
Thread Thread-0 starting
Thread Thread-1 starting
Thread Thread-2 starting
Thread Thread-3 starting
[<_MainThread(MainThread, started 140735793988544)>,
<Thread(Thread-0, started 123145368256512)>, <Thread(Thread-1,
started 123145373511680)>, <Thread(Thread-2, started
123145378766848)>, <Thread(Thread-3, started 123145384022016)>]
Thread Thread-0 ending
Thread Thread-2 ending
Thread Thread-3 ending
Thread Thread-1 ending
```

Примусове завершення потоку

- Вважається дуже поганою практикою програмування.
 - Python не постачає нативних функцій, що «вбиватимуть» інші потоки.
 - Такі потоки можуть містити критичні ресурси, що повинні бути правильно відкриті чи закриті.
 - Також вони можуть бути батьківськими потоками для багатьох дочірніх потоків (так утворюються процеси-сироти, які не мають живого батьківського процесу).
 - Процеси-сироти займають місце в пам'яті, а єдиний спосіб їх «убити» - перелічити всі живі потоки, а потім вбити решту.
- За потреби певного механізму завершення роботи потоку, потрібно м'яко реалізувати його в коді.
 - Проте існує обхідний шлях: процеси, фактично, мають такий механізм на відміну від потоків.

Приклад

```
from multiprocessing import Process
import time
def myWorker():
    t1 = time.time()
    print("Process started at: {}".format(t1))
    time.sleep(20)
myProcess = Process(target=myWorker)
print("Process {}".format(myProcess))
myProcess.start()
print("Terminating Process...")
myProcess.terminate()
myProcess.join()
print("Process Terminated: {}".format(myProcess))
```

Process <Process(Process-1, initial)>

Terminating Process...

Process Terminated: <Process(Process-1, stopped[SIGTERM])>


```
import threading
from multiprocessing import Process
import time
import os
def MyTask():
    print("Starting")
    time.sleep(2)
t0 = time.time()
threads = []
for i in range(10):
    thread = threading.Thread(target=MyTask)
    thread.start()
    threads.append(thread)
t1 = time.time()
print("Total Time for Creating 10 Threads: {} seconds".format(t1-t0))
for thread in threads:
    thread.join()
t2 = time.time()
procs = []
for i in range(10):
    process = Process(target=MyTask)
    process.start()
    procs.append(process)
t3 = time.time()
print("Total Time for Creating 10 Processes: {} seconds".format(t3-t2))
for proc in procs:
    proc.join()
```

Створення процесів та потоків

- Процеси у деякій мірі є важковаговою версією потоків.
 - Вони можуть виконувати більше задач, пов'язаних з CPU, ніж стандартний потік, оскільки кожен з них володіє власним екземпляром GIL.
- У той же час робота процесу більш ресурсозатратна.
 - Перемикання між процесами на льоту та «вбивство» процесу відбувається повільніше.

Вивід програми

Total Time for Creating 10 Threads: 0.0017189979553222656 seconds

Total Time for Creating 10 Processes: 0.02233409881591797 seconds

- Визначаємо функцію MyTask, яка буде ціллю створених у майбутньому потоків та процесів.
 - Спочатку зберігаємо час відліку, а далі створюємо порожній масив (threads), який у зручній формі зберігатиме посилання на всі створені об'єкти-потоки.
 - Потім створюємо та запускаємо ці потоки перед новим записом значення часу, щоб обчислити загальний час створення та запуску.
 - Аналогічні дії виконуємо з процесами.
- Тут час створення та запуску процесів на порядок вищий за відповідний час для потоків.

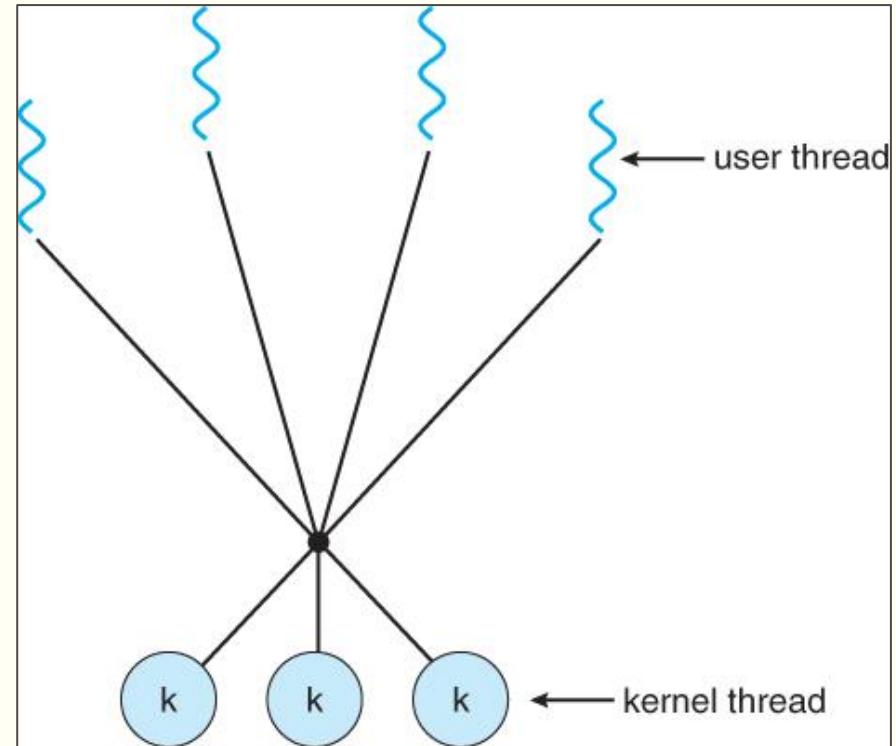
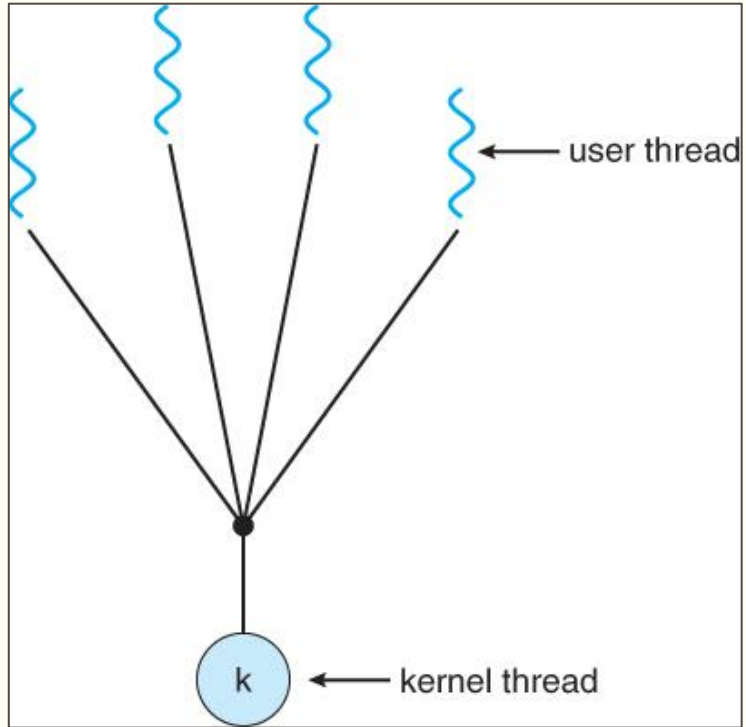
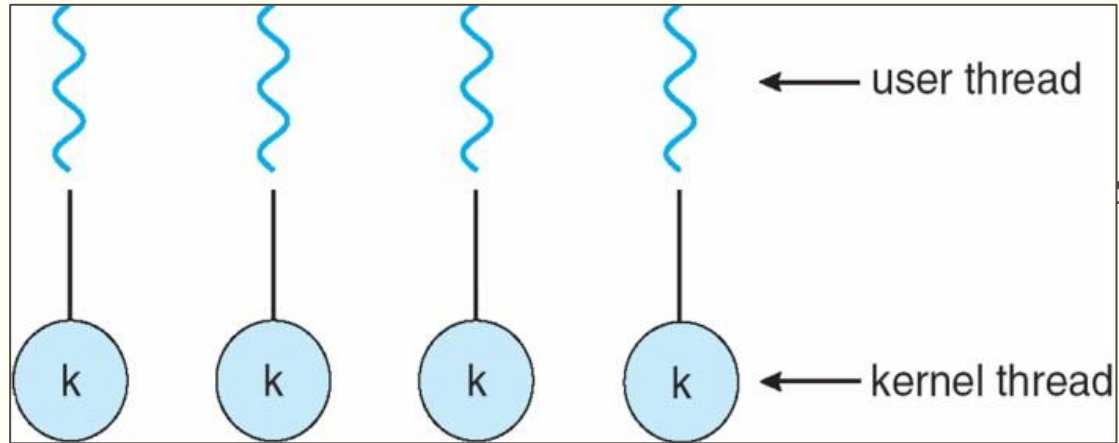
Моделі мультипроцесності

- Відношення між потоками рівня користувача та потоками рівня ядра описується однією з трьох моделей відображення:
 - Один потік рівня користувача до одного потоку рівня ядра
 - Багато потоків рівня користувача до одного потоку рівня ядра
 - Багато потоків рівня користувача до багатьох потоків рівня ядра.
- У Python зазвичай використовується модель відображення «1:1», тому кожен потік використовуватиме більше машинних ресурсів.
 - Проте в екосистемі Python представлені модулі, які дають можливість реалізувати багатопоточний функціонал, залишаючись в одному потоці.
 - Поширений приклад – модуль `asyncio`.

Відображення потоків

- Відображення «1:1» може бути ресурсозатратним (властиво потокам рівня ядра), проте потоки рівня користувача не належать до одного рівня блокування (як для відображення «1:M»).
- Перевагою відображення «1:M» (один потік рівня ядра) є ефективне управління потоками рівня користувача.
- Проте якщо потік рівня користувача блокується, решта відображених потоків теж будуть заблоковані.
- Моделі «M:M» представляє вирішення недоліків попередніх моделей.
- Окремі потоки рівня користувача можна відображати на комбінацію одного або кількох потоків рівня ядра.
- Це дає програмісту змогу обирати, які потоки рівня користувача слід відобразити на потоки рівня ядра та в цілому спробувати забезпечити якомога вищу продуктивність у багатопоточному середовищі.

Відображення потоків





ДЯКУЮ ЗА УВАГУ!

Наступне питання: Синхронізація потоків