TRƯỜNG ĐẠI HỌC KINH TẾ - KỸ THUẬT CÔNG NGHIỆP
KHOA: KHOA HỌC ỨNG DỤNG.

CHƯƠNG 6

ĐƠN LUỒNG (THREAD) VÀ ĐA LUỒNG (MULTITHREAD)

Giảng viên: TS.Cao Diệp Thắng
Khoa: Khoa học Ứng dụng

NỘI DUNG BÀI HỌC

6.1 LUỒNG (THREAD) VÀ ĐA LUỒNG (MULTITHREAD)

6.2 LUỒNG VÀ ĐA LUỒNG TRONG PYTHON

6.3 TẠO LUỒNG TRONG PYTHON

6.4 ĐỒNG BỘ HÓA CÁC THREAD

MỤC TIÊU BÀI HỌC

Sau khi học xong bài này, người học sẽ nắm được các vấn đề sau:

+ Khái niệm về tiến trình và luồng, sự khác nhau giữa chúng, và ưu điểm của việc sử dụng luồng để tận dụng lợi thế của máy tính đa nhân.
+ Khái niệm đồng bộ hóa và cách tránh race condition khi sử dụng đa luồng.
+ Cách sử dụng thư viện threading trong Python để tạo và quản lý luồng.
+ Cách quản lý tài nguyên chia sẻ giữa các luồng để tránh xung đột và race condition.
+ Hiểu về các khái niệm liên quan đến đa luồng như Lock, Semaphore, Condition,..
+ Quy trình tạo và quản lý đa luồng trong Python, bao gồm tạo luồng, khởi động luồng, dừng luồng, và đợi luồng kết thúc.

Một số thách thức lập trình đa luồng

Việc làm quen và sử dụng luồng không phải đơn giản.

- Các chương trình đa luồng thường khó kiểm thử và gỡ lỗi do tính phức tạp của việc quản lý đồng thời các luồng.
- Xung đột dữ liệu khi các luồng truy cập và thay đổi cùng một biến hoặc dữ liệu.
- Khó khăn trong việc đồng bộ hóa các luồng để đảm bảo rằng chúng không xảy ra xung đột và truy cập dữ liệu an toàn.

Yêu cầu người lập trình phải có kiến thức và kỹ năng quản lý luồng, xử lý xung đột dữ liệu, và đồng bộ hóa một cách an toàn.

Lợi ích

Khi được sử dụng một cách đúng đắn

- Cải thiện hiệu suất của chương trình so với lập trình đơn luồng truyền thống.
- Tận dụng được tài nguyên hệ thống một cách hiệu quả,

Mở đầu

Khi chúng ta sử dụng máy tính hàng ngày, mở hàng chục trang web khác nhau, cùng một cơ số đếm không xuể các ứng dụng nghe nhạc, xem phim, game ở ngoài, một câu hỏi đặt ra là: vì sao máy tính có thể thực thi hết chừng đấy việc một lúc không?

Trong mỗi đơn vị thời gian cực nhỏ, chỉ có một tiến trình **(process**) thực sự được CPU xử lý. Bên trong tiến trình đó, hệ điều hành có thể tạo ra nhiều luồng (**thread**) con, giúp các tác vụ tiến triển xen kẽ nhau với tốc độ rất cao.

Nhờ khả năng xử lí các **task** có thể coi như đồng thời (concurrency), chương trình có thể đáp ứng tốt với người dùng trong khi đang bận làm việc khác.

Đó là chính ý tưởng cơ bản của **multithread**.

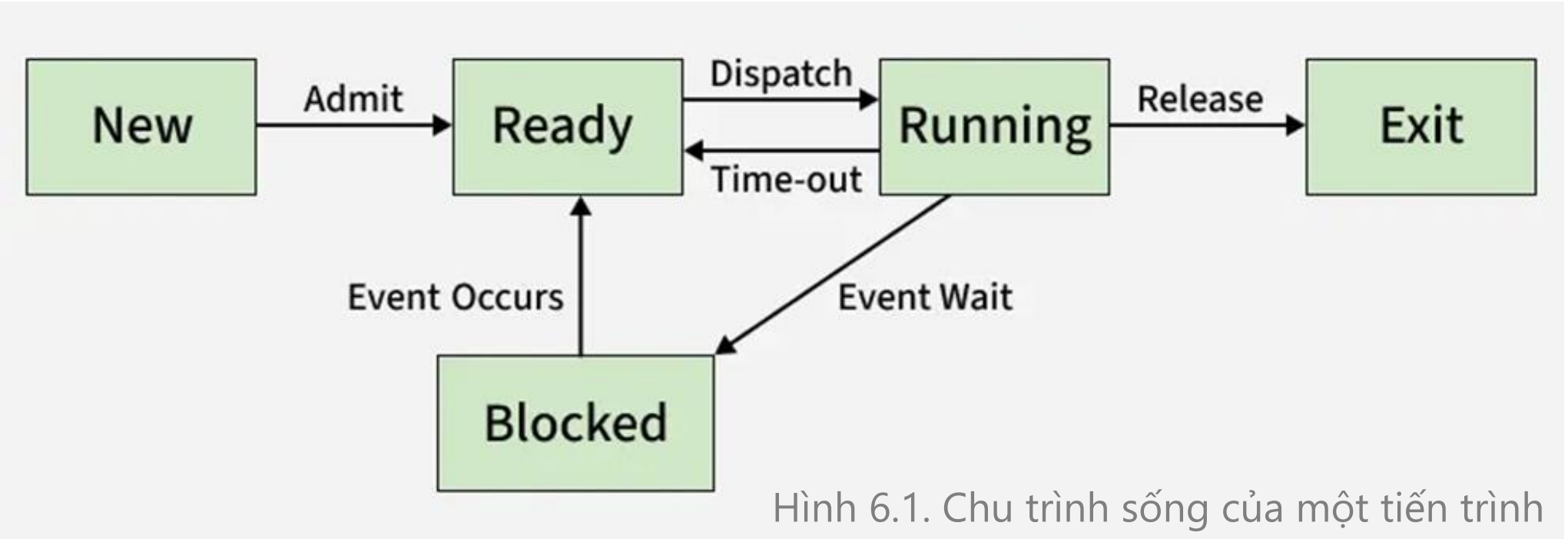
6.1. TIẾN TRÌNH (PROCESS), LUỒNG (THREAD), .

6.1.1. Khái niệm tiến trình – Process

- Một **tiến trình** (process) là một phiên bản của chương trình đang được thực thi. Tiến trình là đơn vị **độc lập** được HĐH (OS) cấp phát tài nguyên hệ thống.

- **Không gian bộ nhớ riêng** và chạy độc lập với các tiến trình khác.
- **Tài nguyên hệ thống**, trạng thái thực hiện của chương trình, bao gồm: định danh tiến trình (**process ID**), User ID, group ID, các thanh ghi (register), stack, heap, các file đang mở, và các tài nguyên I/O khác.
- **Chi phí:** Việc khởi tạo và chuyển đổi ngữ cảnh (Context Switching) của Tiến trình tốn kém hơn so với Luồng do phải quản lý không gian bộ nhớ độc lập.

Mỗi khi một tiến trình được tạo, một khối điều khiển tiến trình **PCB**(Process Control Block) mới được tạo ra để lưu trữ thông tin của tiến trình đó



Hình 6.1. Chu trình sống của một tiến trình

Các trạng thái (States)

- **New (Mới):** Tiến trình vừa được tạo ra nhưng chưa sẵn sàng để thực thi.
- **Ready (Sẵn sàng):** Tiến trình đã sẵn sàng để được thực thi và đang chờ CPU.
- **Running (Đang chạy):** Tiến trình đang được CPU thực thi.
- **Blocked (Bị chặn) / Waiting (Đang chờ):** Tiến trình đang chờ một sự kiện nào đó xảy ra (ví dụ: hoàn thành I/O, nhận tín hiệu).
- **Exit (Thoát) / Terminated (Kết thúc):** Tiến trình đã hoàn thành việc thực thi và sắp bị loại bỏ khỏi hệ thống.

Các chuyển đổi Trạng thái (Transitions)

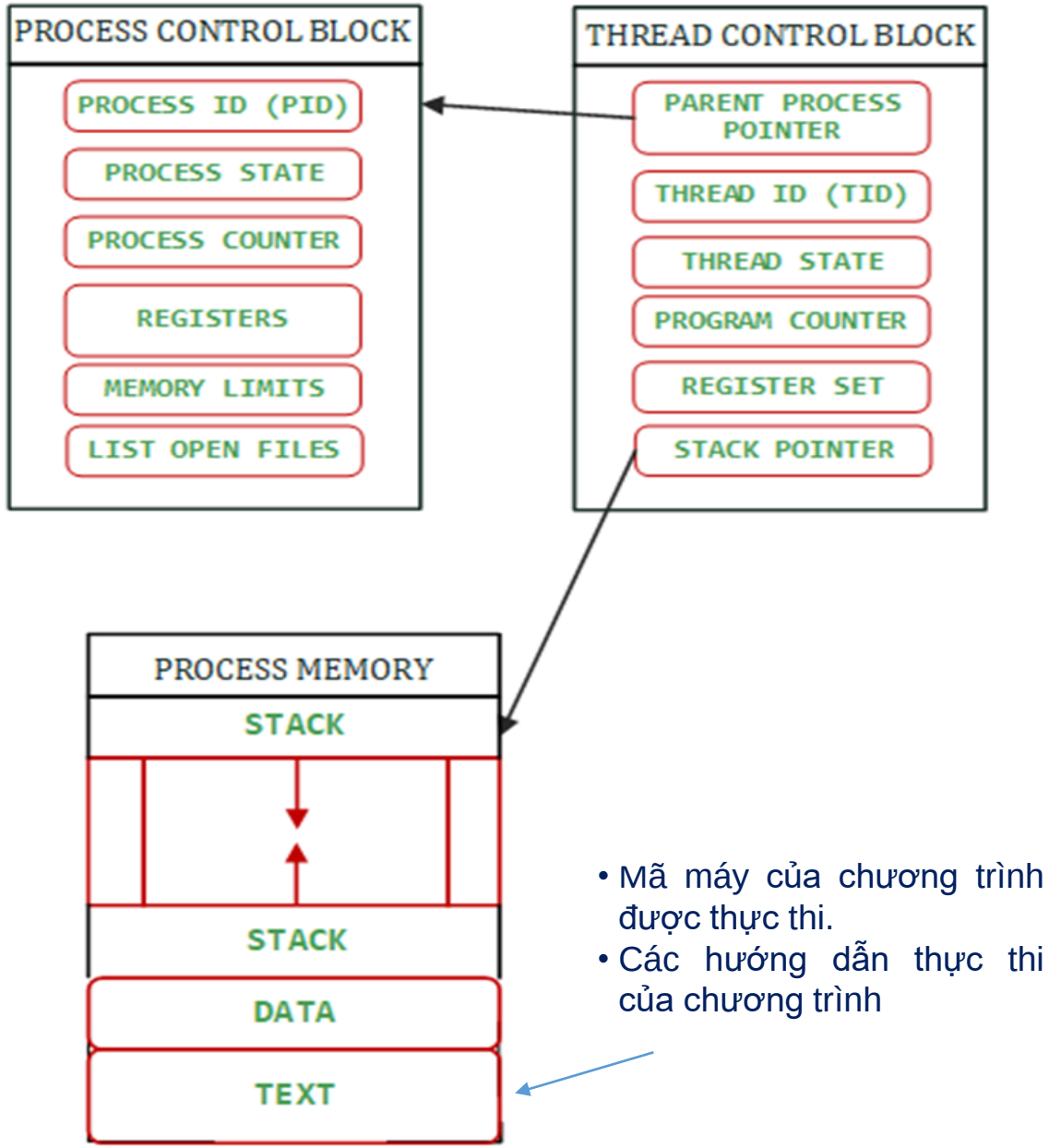
- **New → Ready (Admit/Thừa nhận):** tiến trình được HĐH nạp vào bộ nhớ chính và sẵn sàng để thực thi.
- **Ready →Running (Dispatch/Điều phối):** Bộ lập lịch chọn tiến trình này và cấp CPU cho nó để thực thi.
- **Running → Ready (Time-out/Hết thời gian):** Lát cắt thời gian (time slice) cấp cho tiến trình đã hết, và nó bị CPU ngắt để nhường chỗ cho tiến trình khác.
- **Running → Blocked (Event Wait/Chờ Sự kiện):** tiến trình cần thực hiện một hoạt động I/O hoặc chờ một sự kiện nào đó xảy ra, nên nó tự nguyện rời CPU.
- **Blocked → Ready (Event Occurs/Sự kiện Xảy ra):** Sự kiện mà tiến trình đang chờ đợi (ví dụ: I/O hoàn tất) đã xảy ra, và tiến trình quay lại hàng đợi sẵn sàng.
- **Running →Exit (Release/Giải phóng):** Tiến trình đã hoàn thành công việc của mình (thực thi xong) và kết thúc.

6.1.2. Khái niệm luồng – thread:

- Thread là **đơn vị thực thi nhỏ nhất** trong tiến trình., Một tiến trình có thể chứa nhiều luồng.
- Các thread trong cùng tiến trình **chia sẻ bộ nhớ chung** (code, dữ liệu, tài nguyên).
- Mỗi thread có **stack, thanh ghi, program counter** riêng.
- Cho phép chương trình xử lý **nhiều tác vụ gần như đồng thời**, kể cả trên CPU một lõi (nhờ **context switching**).
- Giúp tăng **tính phản hồi** của ứng dụng, đặc biệt khi có nhiều tác vụ độc lập.
- Thread được OS quản lý qua **Thread Control Block (TCB)**:

  → lưu trạng thái, thanh ghi, PC, ID, con trỏ stack…
- Khi chuyển ngữ cảnh, OS **lưu & phục hồi** TCB để thread tiếp tục chính xác.

Tất cả các luồng trong chương trình dường như đang được thực thi đồng thời, mặc dù chúng "**lần lượt**" thực hiện trên một bộ xử lý/mạch lõi (processor/core) duy nhất. Bằng cách sử dụng luồng, chương trình có thể chạy đồng thời nhiều tác vụ và tận dụng tài nguyên hệ thống hiệu quả hơn.[15].



Hình 6.2. Quan hệ giữa PCB, TCB và bố cục bộ nhớ tiến trình

- **Process Control Block (PCB)** lưu trữ trạng thái và tài nguyên của tiến trình, trong khi **Thread Control Block (TCB)** ghi nhận ngữ cảnh thực thi của từng luồng.

- Trong một tiến trình đa luồng, tất cả các luồng **chia sẻ chung không gian địa chỉ** gồm vùng mã lệnh, dữ liệu toàn cục và vùng heap, trong khi mỗi luồng vẫn duy trì các tài nguyên riêng như ngăn xếp và ngữ cảnh thực thi được lưu trong TCB.

- Hệ điều hành có thể quản lý chính xác cả tài nguyên chung của tiến trình lẫn trạng thái riêng của từng luồng, đồng thời đảm bảo khả năng luân phiên thực thi giữa các luồng một cách an toàn và nhất quán.

6.1.3. Context Switching – Chuyển đổi ngữ cảnh

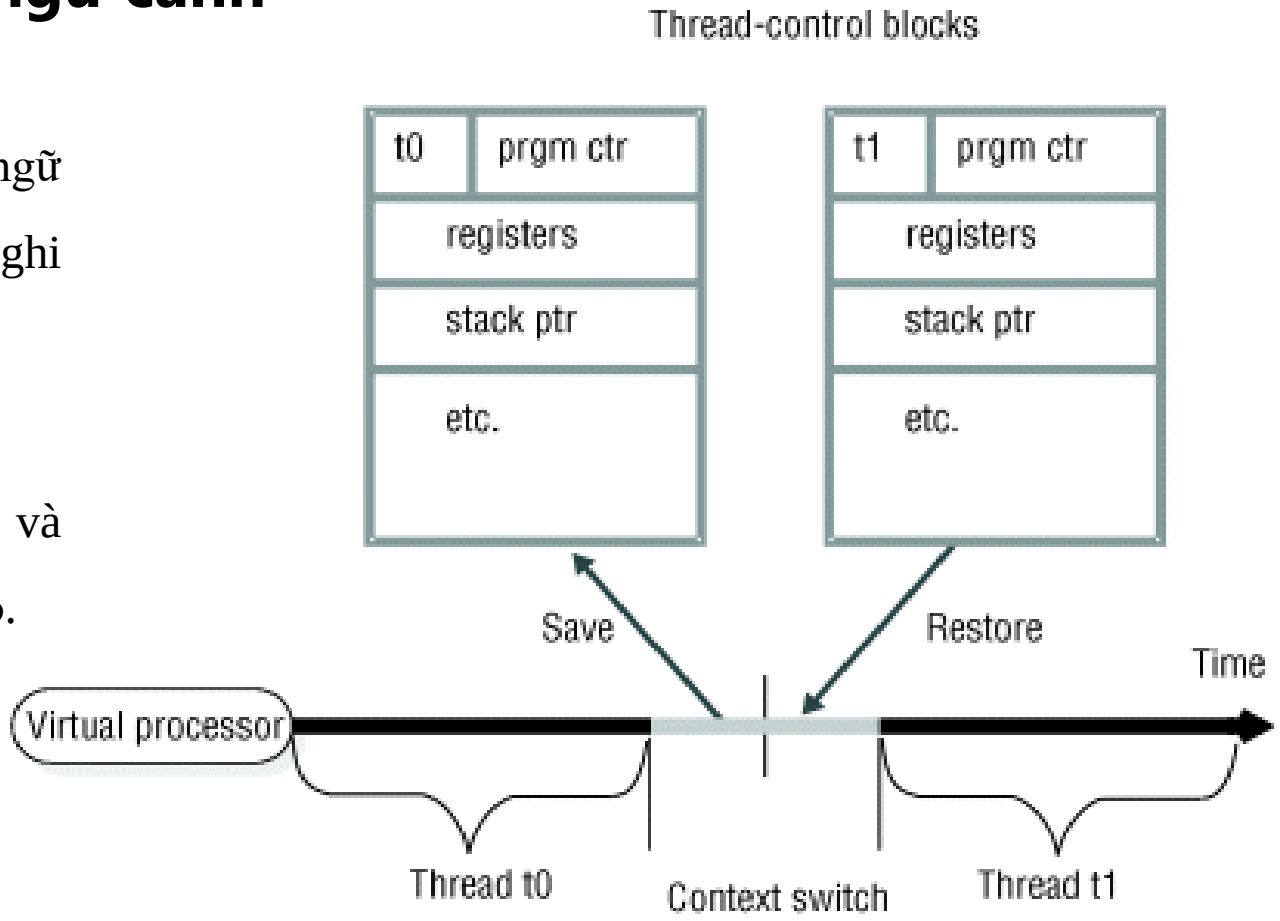
- Mỗi lõi CPU **chỉ chạy 1 luồng tại một thời điểm** → OS tạo "ảo giác đa nhiệm" bằng cách luân phiên cấp CPU cho các luồng.
- Khi chuyển luồng, OS **lưu trạng thái hiện tại** (PC, thanh ghi, con trỏ stack…) vào TCB/PCB và **khôi phục** trạng thái của luồng khác.
- Việc chuyển ngữ cảnh xảy ra khi:
    - hết quantum CPU,
    - chờ I/O,
    - có tác vụ ưu tiên cao hơn. ⚡
- Context switching **có chi phí**: mất thời gian lưu/khôi phục, giảm hiệu quả cache → quá nhiều luồng sẽ làm hệ thống chậm.
- Nhờ cơ chế này, OS **xen kẽ thực thi nhiều tác vụ trên 1 lõi CPU** → nền tảng của lập lịch preemptive trong hệ điều hành hiện đại.

6.1.3. Context Switching – Chuyển đổi ngữ cảnh

Khi luồng t0 đang chạy, hệ điều hành lưu lại toàn bộ ngữ cảnh của nó—bao gồm bộ đếm chương trình, tập thanh ghi và con trỏ stack—vào TCB tương ứng

Hệ điều hành nạp ngữ cảnh của luồng t1 từ TCB của nó và tiếp tục thực thi từ điểm mà luồng t1 đã tạm dừng trước đó.

Trục thời gian trong hình cho thấy CPU chỉ thực thi một luồng tại một thời điểm, còn khả năng đa nhiệm đạt được nhờ việc lặp đi lặp lại thao tác lưu và khôi phục ngữ cảnh với tốc độ rất cao.



Hình 6.3. Sơ đồ chuyển đổi ngữ cảnh

6.1.4. Concurrency (Đa nhiệm) và Parallelism (Song song)

- **Concurrency**: nhiều tác vụ *tiến triển trong cùng khoảng thời gian* nhờ **context switching**, ngay cả trên **một lõi CPU**.
- **Parallelism**: nhiều tác vụ *chạy đồng thời về mặt vật lý* trên **nhiều lõi CPU / nhiều bộ xử lý**.
- Trong Python:
  - Tác vụ **I/O-bound** → concurrency hiệu quả (threading, asyncio).
  - Tác vụ **CPU-bound** → cần multiprocessing để đạt parallelism (vì GIL giới hạn chỉ 1 luồng chạy bytecode).
- Concurrency là vấn đề **tổ chức và điều phối tác vụ**; parallelism là vấn đề **phần cứng và khả năng thực thi thực sự song song**.
- Một chương trình có thể có concurrency mà không parallelism (CPU 1 lõi), hoặc có parallelism mà không cần mô hình concurrency phức tạp (nhiều tiến trình độc lập).

So sánh Concurrency (Đa nhiệm) và Parallelism (Song song)

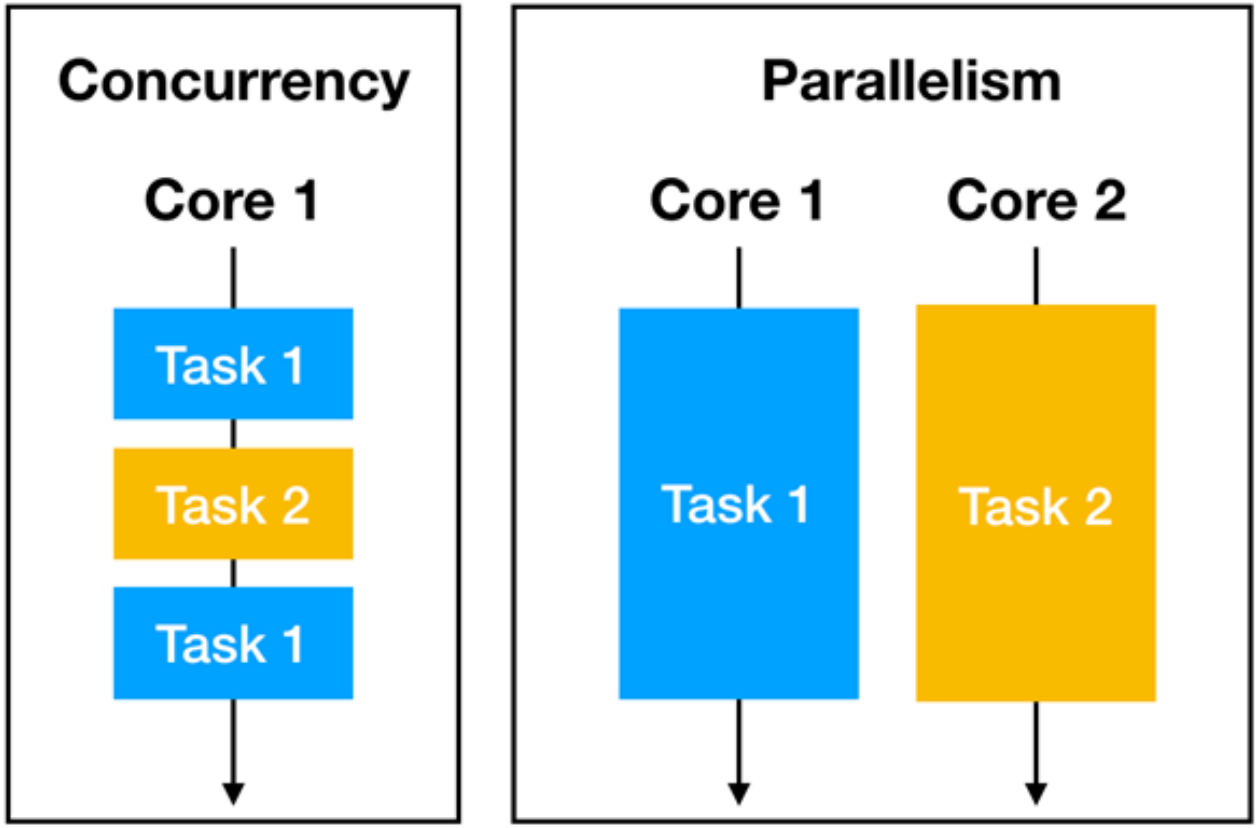
Đặc điểm	Concurrency (Đa nhiệm)	Parallelism (Song song)
Mục tiêu	Quản lý nhiều việc cùng một lúc.	Làm nhiều việc cùng một thời điểm.
Thực thi	Các tác vụ **xen kẽ** nhau theo thời gian trên một tài nguyên (ví dụ: một lõi CPU duy nhất).	Các tác vụ được thực thi **đồng thời** trên nhiều tài nguyên (ví dụ: nhiều lõi CPU).
Hiệu suất	Tăng khả năng **đáp ứng** (Responsiveness) và sử dụng hiệu quả thời gian chờ I/O.	Tăng **tốc độ** xử lý tổng thể (Throughput) cho các tác vụ tính toán.
Ứng dụng	Đa luồng (**threading**) trong Python (do GIL).	Đa tiến trình (**multiprocessing**) trong Python.

So sánh Concurrency (Đa nhiệm) và Parallelism (Song song)

Concurency, hai tác vụ được thực thi xen kẽ trên cùng một lõi CPU, thể hiện mô hình concurrency dựa trên cơ chế chuyển đổi ngữ cảnh.

Parallesm: hai tác vụ chạy đồng thời trên hai lõi CPU độc lập, biểu diễn parallelism thực sự

concurrency liên quan tới cách tổ chức tác vụ, còn parallelism phụ thuộc vào khả năng phần cứng đa lõi.



Hình 6.4. Concurrency và Parallelism

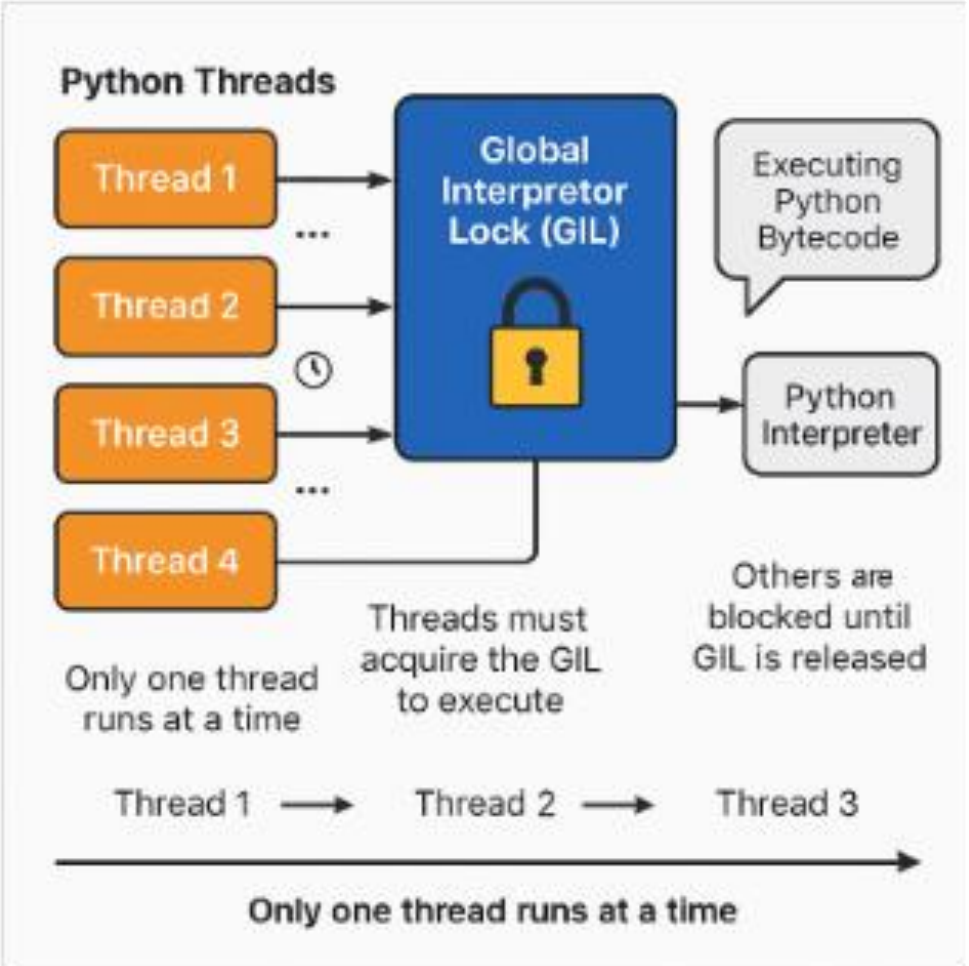
6.2. LUỒNG & ĐA LUỒNG TRONG PYTHON

6.2.1. Mô hình đa luồng trong CPython và vai trò của GIL

- **GIL (Global Interpreter Lock)** đảm bảo tại một thời điểm **chỉ một luồng được thực thi bytecode Python**, nhằm bảo vệ các cấu trúc dữ liệu nội bộ vốn **không thread-safe**.
- GIL giúp đơn giản hóa bộ thông dịch CPython nhưng **hạn chế khả năng song song thực sự** của đa luồng trên tác vụ CPU-bound.
- Cơ chế hoạt động:
  - Một luồng muốn chạy bytecode phải **giữ GIL**.
  - GIL được cấp theo **lượt rất ngắn**, rồi được thu hồi và chuyển cho luồng khác.
  - Nếu luồng bị chặn bởi I/O → **GIL được nhả ra** → luồng khác chạy tiếp.
- Với tác vụ **I/O-bound**: đa luồng hoạt động hiệu quả nhờ thời gian chờ I/O → luồng khác có cơ hội nắm GIL.
- Với tác vụ **CPU-bound**: một luồng tính toán liên tục sẽ **giữ GIL gần như toàn bộ thời gian**, khiến các luồng khác bị chặn → đa luồng **không đạt song song**.

6.2.1. Luồng trong Python, tiếp …:

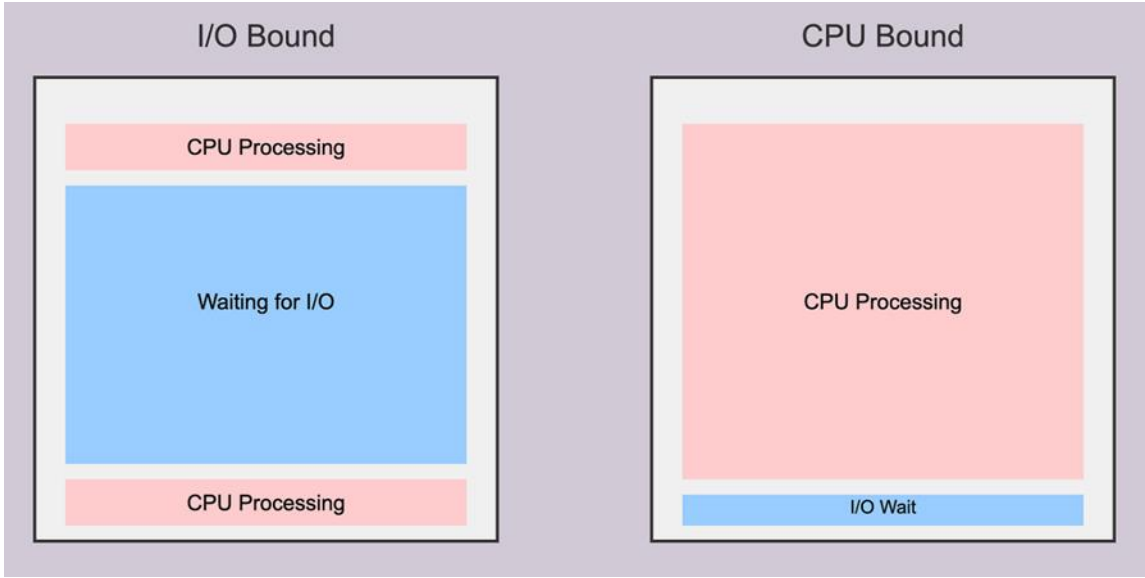
Trên hình 6.5 cho thấy rõ rằng dù chương trình có thể tạo ra nhiều luồng (Thread 1, Thread 2, Thread 3…), tất cả đều phải đi qua "cổng kiểm soát" GIL. Luồng nào giành được GIL thì được chạy; các luồng còn lại sẽ tạm dừng, đợi GIL được giải phóng. Vì vậy, trong các bài toán đòi hỏi tính toán nặng về CPU (CPU-bound), đa luồng Python không cải thiện tốc độ—do bản thân GIL giới hạn mức độ song song.



Hình 6.5. Cơ chế Global Interpreter Lock (GIL) trong CPython.

Ảnh hưởng của GIL lên I/O-bound và CPU-bound

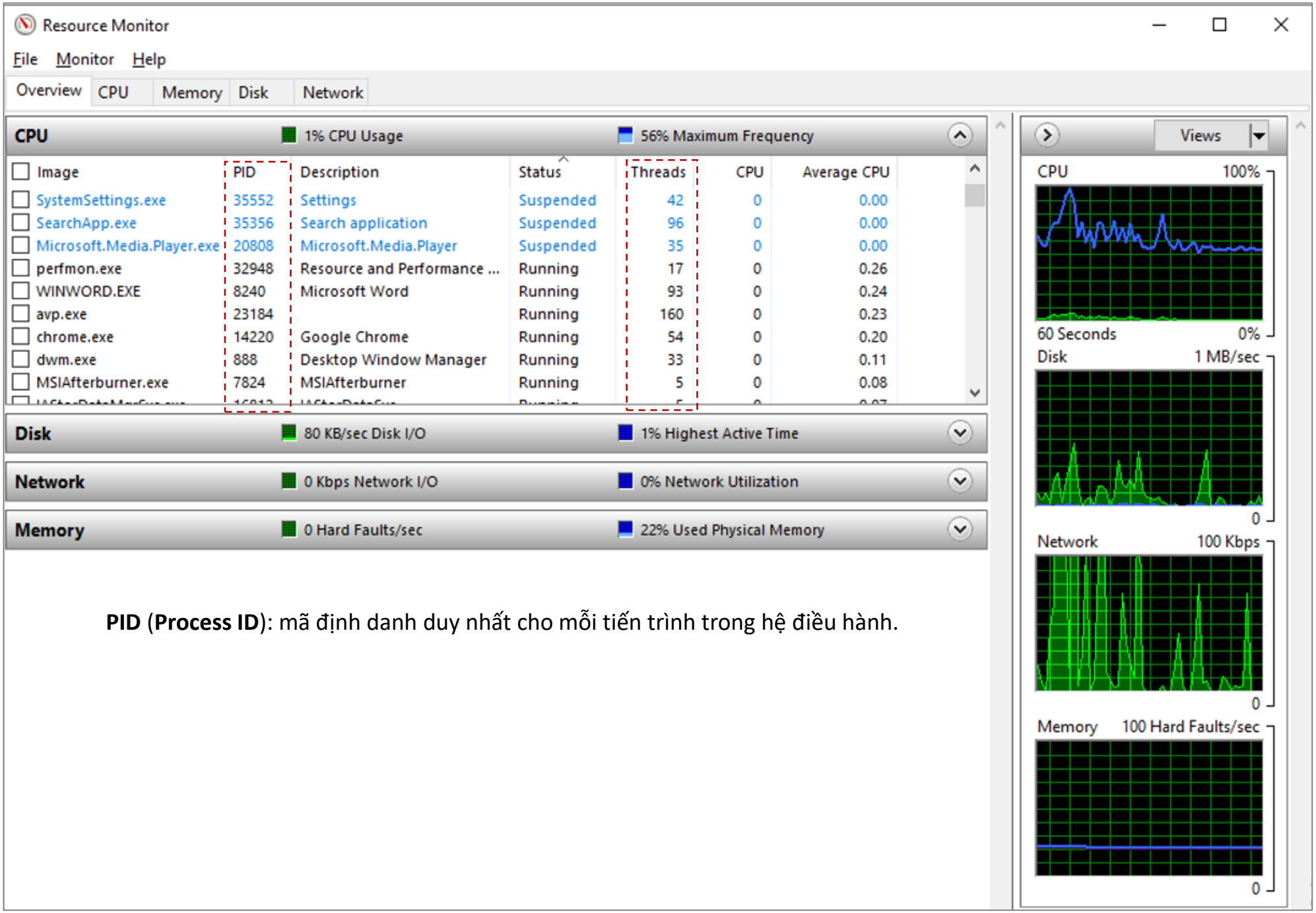
- **I/O-bound**: phần lớn thời gian *chờ* (file, mạng, DB…), nên khi chờ I/O → **GIL được nhả**.
  → Các luồng khác có thể chạy tiếp → Python đạt **đồng thời cao**, dù chỉ có 1 GIL.
- **CPU-bound**: luồng phải liên tục thực hiện phép toán, *không có thời gian chờ* → **không nhả GIL**.
  → Một luồng giữ GIL gần như toàn bộ thời gian → các luồng khác bị chặn → **không đạt song song** trên nhiều lõi CPU.
- **Kết luận**:
  - Đa luồng Python hiệu quả cho **I/O-bound**.
  - Không hiệu quả cho **CPU-bound** do GIL giới hạn.



Hình 6.6. Minh họa sự khác biệt giữa I/O-bound và CPU-bound

Hình 6.6 cho thấy:

- I/O-bound → thời gian CPU ít, thời gian chờ nhiều
- CPU-bound → thời gian CPU chiếm ưu thế → GIL bị giữ lâu



PID (**Process ID**): mã định danh duy nhất cho mỗi tiến trình trong hệ điều hành.

6.2.2. Mô-đun threading trong Python

Mặc dù CPython chịu sự điều phối của Global Interpreter Lock (GIL), mô-đun threading vẫn là công cụ tiêu chuẩn để triển khai đa nhiệm (concurrency) đặc biệt trong các tác vụ I/O-bound. Mô-đun này cho phép chương trình tạo và quản lý nhiều luồng (thread) hoạt động đồng thời trong cùng một tiến trình (process), tận dụng hiệu quả các giai đoạn chờ I/O của từng luồng.

Để sử dụng hiệu quả threading, lập trình viên cần nắm ba lớp kiến thức:

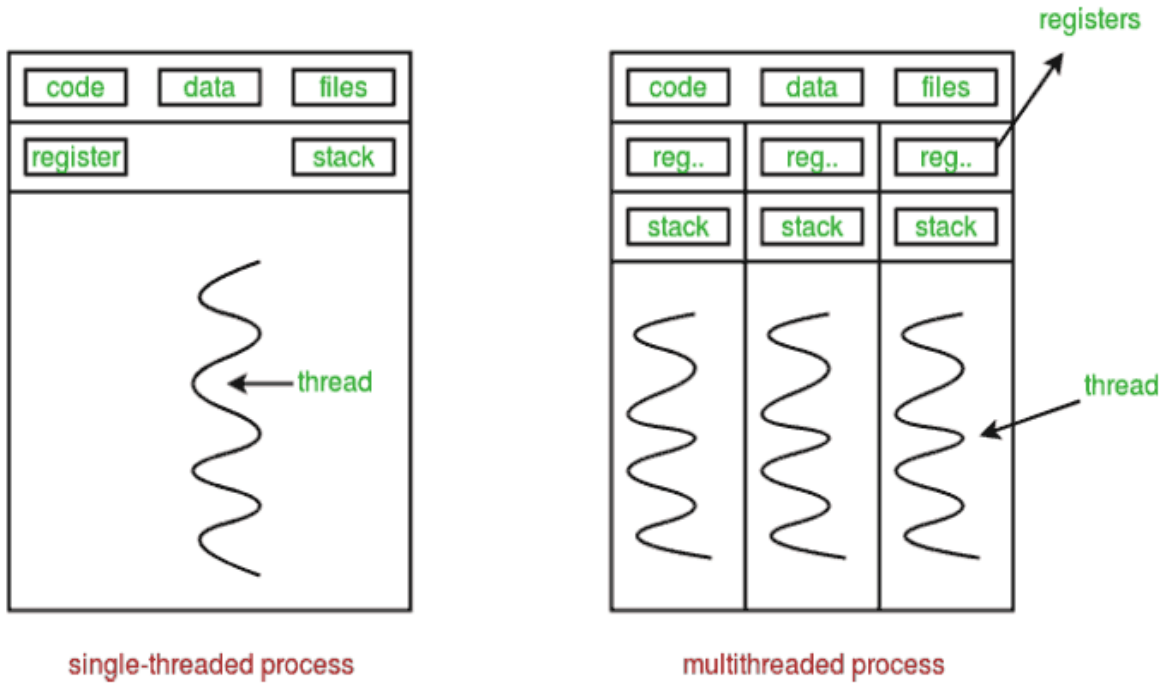
(1) cấu trúc tiến trình đa luồng ở mức hệ điều hành,

(2) cách tạo và quản lý luồng trong Python,

(3) cách quan sát luồng thực tế khi chương trình chạy.

6.2.2.1. Thread trong tiến trình Python:

- Trong một tiến trình, **tất cả các luồng chia sẻ cùng một không gian bộ nhớ** (code, data, files),
- Mỗi luồng có **ngăn xếp (stack) và thanh ghi (registers) riêng**, phục vụ cho quá trình thực thi độc lập.
- Nhờ mô hình chia sẻ bộ nhớ nhưng tách biệt stack/registers, **một chương trình Python có thể duy trì nhiều luồng hoạt động xen kẽ nhau mà không cần sinh thêm tiến trình mới.**



Hình 6.7. So sánh tiến trình đơn luồng và đa luồng.

- **single-threaded process,** tiến trình chỉ có một luồng duy nhất; toàn bộ tài nguyên (code, stack, register) thuộc về một luồng.
- **multi-threaded process**, các luồng dùng chung code/data/files, nhưng mỗi luồng có stack riêng để lưu biến cục bộ và trạng thái thực thi, và register riêng để CPU khôi phục bối cảnh (context) khi chuyển đổi luồng.
- Đây chính là mô hình nền tảng của OS cho phép Python tạo ra hàng chục luồng bên trong một tiến trình duy nhất.

6.2.2.2. Tạo và quản lý luồng bằng lớp Thread

- Lớp **Thread** là trung tâm của mô-đun threading, đại diện cho **một luồng thực thi độc lập** trong tiến trình Python.

Các bước sử dụng Thread:

1. Khởi tạo: truyền hàm nhiệm vụ qua target=..., hoặc kế thừa lớp Thread và ghi đè run().
2. Bắt đầu luồng: gọi start() để tạo native thread ở cấp hệ điều hành; luồng mới chạy song song với luồng chính.
3. Đồng bộ hóa: gọi join() khi cần đợi luồng con hoàn tất trước khi chương trình tiếp tục.

Mặc dù GIL giới hạn song song CPU, đa luồng vẫn hiệu quả với tác vụ I/O-bound: khi một luồng chờ I/O → GIL được nhả → luồng khác được thực thi.

Một số thuộc tính hữu ích: name (đặt tên luồng),daemon (luồng nền tự kết thúc khi luồng chính kết thúc),is_alive() (kiểm tra trạng thái).

threading.Thread tạo native thread do kernel quản lý; có thể quan sát số luồng tăng trong Task Manager.

*Dù có nhiều luồng, bytecode Python vẫn chịu GIL → chỉ đạt **concurrency với I/O-bound**, không đạt **song song thực sự** cho tác vụ CPU-bound*

6.2.3. Đa luồng trong Python

Lớp thread cung cấp các phương thức để làm việc với luồng, cụ thể là:

- Phương thức **starst**(): phương thức dùng để kích hoạt (chạy) một thread object, mặc định sẽ gọi phương thức run().
- Phương thức **run**(): Đây là phương thức chính chúng ta cần cài đặt, mô tả các công việc mà luồng thực hiện. Mặc định phương thức này sẽ gọi hàm liên kết với đối số target lúc khởi tạo luồng.
- Phương thức **join**(): Khi được gọi, method này sẽ block thread gọi nó (calling thread) cho đến khi thread được gọi (called thread - tức là thread có method join() vừa gọi) kết thúc. Method này thường được dùng trong luồng chính để đợi các thread khác kết thúc công việc của mình và xử lý tiếp kết qủa.
- Một số phương thức khác của lớp thread như name(), getName(), setName(),…có thể được tham khảo thêm tại

  https://docs.python.org/3/library/threading.html

6.3.TẠO LUỒNG (THREAD) TRONG PYTHON:

Có hai cách chính để tạo luồng mới bằng lớp Thread trong Python

- Khởi tạo từ hàm, chúng ta có thể truyền một hàm làm tham số cho lớp Thread. Luồng mới sẽ thực hiện công việc được định nghĩa trong hàm đó.
- Kế thừa từ lớp Thread và ghi đè phương thức run() để thực hiện công việc của luồng.

6.3.1. Khởi tạo luồng từ hàm

Tạo luồng mới bằng cách truyền một hàm làm tham số*:*

Cú pháp

hàm mục tiêu (target function) mà luồng sẽ thực thi.

```
threading.Thread(target=function, args=args, kwargs=kwargs)
```

(tuỳ chọn) là một tuple chứa các đối số (arguments) của hàm mục tiêu.

(tuỳ chọn) là một dictionary chứa các đối số (keyword arguments) của hàm mục tiêu.

Ví dụ 6.3.1.1. Tạo một luồng mới và thực thi hàm my_function

```
1  import threading
2
3  def my_function(arg):
4      # Công việc của luồng
5      print("Đối số là: ",arg)
6      print("Đây là luồng đang chạy!!")
7  my_thread = threading.Thread(target=my_function, args=(10,))
8  my_thread.start()
```

```
Đối số là:  10
Đây là luồng đang chạy!!
```

Ví dụ 6.3.1.2. Sử dụng mô đun threading.

In[]:

```
1  import threading
2  def print_cube(num):
3      # function to print cube of given num
4      print("Cube: {}".format(num * num * num))
5  def print_square(num):
6      # function to print square of given num
7      print("Square: {}".format(num * num))
8  if __name__ == "__main__":
9      # creating thread
10     t1 = threading.Thread(target=print_square, args=(10,))
11     t2 = threading.Thread(target=print_cube, args=(10,))
12     # start the threads
13     t1.start()
14     t2.start()
15     # wait until both threads finish
16     t1.join()
17     t2.join()
18
19     # continue with the main loop
20     for i in range(5):
21         print("Main loop: {}".format(i))
22     print("Done!")
```

Tạo luồng t1, t2.

Khởi động luồng (Start a Thread) t1, t2.

Dừng thực thi chương trình hiện tại cho đến khi hoàn thành luồng t1,t2

```
Square: 100
Cube: 1000
Main loop: 0
Main loop: 1
Main loop: 2
Main loop: 3
Main loop: 4
Done!
```

Multithreading: Khi sử dụng threading.Thread và gọi start(), các công việc được thực thi (song song) trên các luồng riêng biệt, không theo thứ tự tuần tự như khi gọi các hàm thông thường.

https://docs.python.org/3/library/threading.html#threading.Thread

6.3.2. Khởi tạo luồng mới bằng phương pháp kế thừa từ lớp thread.

Một cách khác để khởi tạo một thread đó là kế thừa lại Threading module. Chúng ta cần ghi đè override các phương thức cần điều chỉnh từ lớp cha.

Bước 1: import module threading:

Bước 2: Định nghĩa lớp con kế thừa từ lớp Thread

```
class MyThread(threading.Thread):
    def run(self):
        # Công việc của luồng, chẳng hạn in ra thông báo luồng đang thực hiện
        print("Thread is running")
```

Bước 3: Tạo đối tượng luồng từ lớp con

```
my_thread = MyThread()
```

Bước 4: Khởi động luồng (Start a Thread)

```
my_thread.start()
```

Bước 5: Kết thúc thực thi luồng

```
my_thread.join()
```

Ví dụ 6.3.2.1.

```python
import threading

class PrintThread(threading.Thread):
    def __init__(self, num):
        super().__init__()
        self.num = num

    def run(self):
        self.print_square()
        self.print_cube()

    def print_cube(self):
        # function to print cube of given num
        print("Cube: {}".format(self.num * self.num * self.num))

    def print_square(self):
        # function to print square of given num
        print("Square: {}".format(self.num * self.num))

if __name__ == "__main__":
    # creating thread
    t = PrintThread(10)

    # start the thread
    t.start()

    # wait until the thread finishes
    t.join()

    # continue with the main loop
    for i in range(5):
        print("Main loop: {}".format(i))
    print("Done!")
```

```
Square: 100
Cube: 1000
Main loop: 0
Main loop: 1
Main loop: 2
Main loop: 3
Main loop: 4
Done!
```

Trong hệ điều hành, mỗi chương trình khi chạy sẽ được gán một ID duy nhất để phân biệt với các chương trình khác. ID này được gọi là Process ID (PID).

- sử dụng hàm **os.getpid()** để lấy ID của chương trình hiện tại.
- Sử dụng hàm **threading.main_thread()** để lấy đối tượng luồng chính. Trong điều kiện bình thường, luồng chính là luồng mà trình thông dịch Python được bắt đầu. thuộc tính tên của đối tượng luồng được sử dụng để lấy tên của luồng.
- sử dụng hàm **threading.current_thread()** để lấy đối tượng luồng hiện tại.

Ví dụ 6.3.2.2: minh họa cách sử dụng module threading để tạo và quản lý nhiều luồng.

Ví dụ 6.3.2.2. Minh họa đa luồng (MultiThreading).

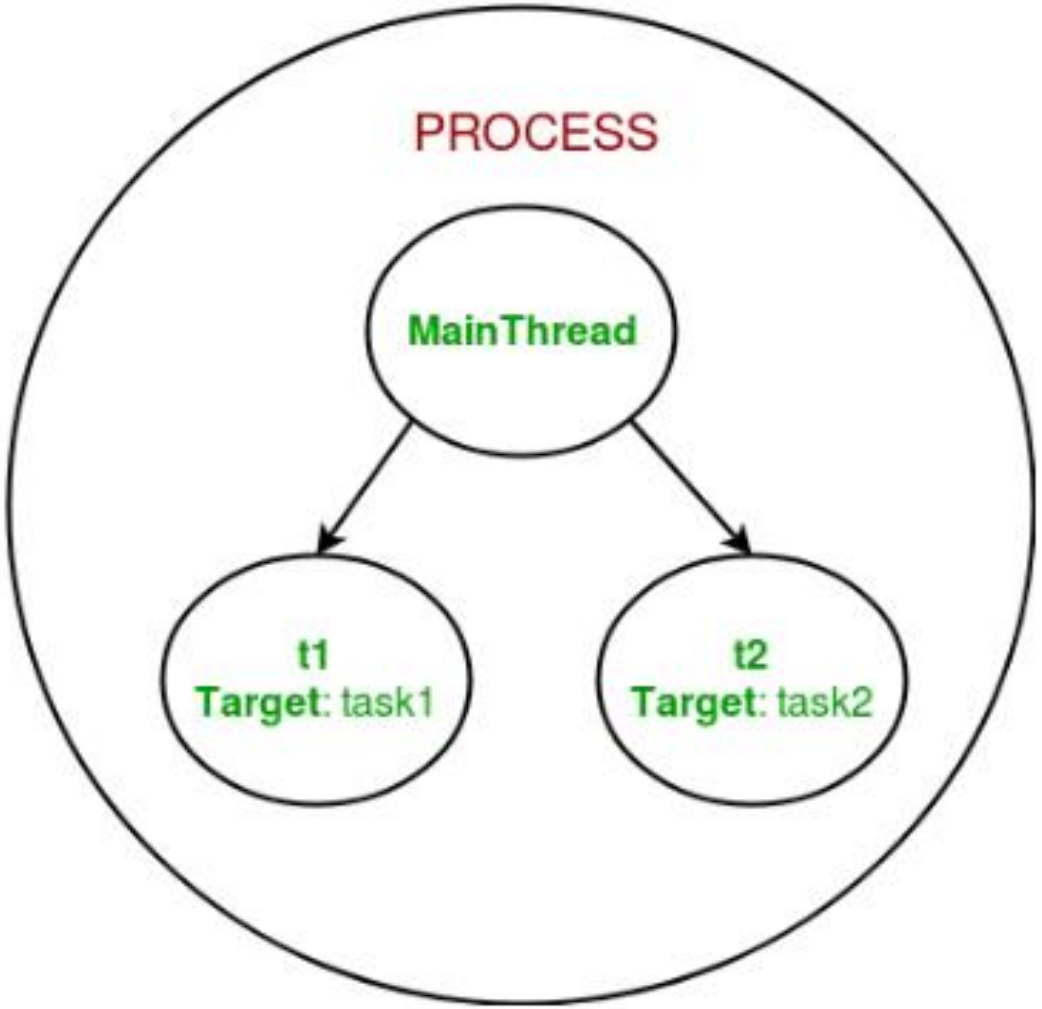
```
1   # Chương trình minh họa khái niệm luồng 'Threading'
2   import threading
3   import os
4
5   def task1():
6       print("Gắn tác vụ 1 đến luồng: {}".
7                               format(threading.current_thread().name))
8       print("ID của tiến trình chạy task 1: {}".format(os.getpid()))
9
10  def task2():
11      print("Gắn tác vụ 1 đến luồng: {}".
12                              format(threading.current_thread().name))
13      print("ID của tiến trình chạy task 2:  {}".format(os.getpid()))
14
15  if __name__ == "__main__":
16        # In ra ID của luồng hiện thời
17      print("ID của luồng đang thực thi trong chương trình chính: {}".
18                              format(os.getpid()))
19      # in ra tên luồng chính
20      print("Tên của luồng chính: {}".
21                             format(threading.current_thread().name))
22      # Tạo các luồng
23      t1 = threading.Thread(target=task1, name='t1')
24      t2 = threading.Thread(target=task2, name='t2')
25
26      # Bắt đầu các luồng:
27      t1.start()
28      t2.start()
29
30      # Chờ cho đến khi các luồng kết thúc:
31      t1.join()
32      t2.join()
```

Kết quả chạy chương trình:

ID của luồng đang thực thi trong chương trình chính: **4192**
Tên của luồng chính: MainThread
Gắn tác vụ 1 đến luồng: t1
ID của tiến trình chạy task 1: **4192**
Gắn tác vụ 1 đến luồng: t2
ID của tiến trình chạy task 2: **4192**

Nhận xét: Cả hai luồng có cùng ID tiến trình (4192), vì Python thực thi các luồng trong cùng một tiến trình do GIL (Global Interpreter Lock).

- Như vậy, mặc dù Python hỗ trợ đa luồng, các luồng đều chạy trong cùng một tiến trình do GIL. (global Interpreter Lock



Hình 6.4. Minh họa hoạt động đa luồng (Multithreading).

Để kiểm tra thời gian thực thi và thấy rõ lợi ích của luồng, có thể sử dụng module **time** để đo thời gian chương trình chạy với và không sử dụng luồng.

Ví dụ 6.3.2.2.a

```
import threading
import os
import time


def task1():
    print("Gắn tác vụ 1 đến luồng:{}".
                                        format(threading.current_thread().name))
    print("ID của tiến trình chạy task 1: {}".format(os.getpid()))
    time.sleep(2)  # Mô phỏng công việc mất 2 giây
    print("Task 1 hoàn thành")


def task2():
    print("Gắn tác vụ 2 đến luồng:
{}".format(threading.current_thread().name))
    print("ID của tiến trình chạy task 2: {}".format(os.getpid()))
    time.sleep(3)  # Mô phỏng công việc mất 3 giây
    print("Task 2 hoàn thành")
```

```python
if __name__ == "__main__":
    print("ID của tiến trình chính: {}".format(os.getpid()))
    print("Tên của luồng chính: {}".format(threading.current_thread().name))

    # Kiểm tra thời gian thực thi không dùng luồng
    start_time = time.time()
    print("\n--- Thực thi tuần tự ---")
    task1()
    task2()
    sequential_time = time.time() - start_time
    print("Thời gian thực thi tuần tự: {:.2f} giây".format(sequential_time))
# Kiểm tra thời gian thực thi dùng luồng
    start_time = time.time()
    print("\n--- Thực thi với luồng ---")
    t1 = threading.Thread(target=task1, name='t1')
    t2 = threading.Thread(target=task2, name='t2')

    t1.start()
    t2.start()

    t1.join()
    t2.join()
    threaded_time = time.time() - start_time
    print("Thời gian thực thi với luồng: {:.2f} giây".format(threaded_time))
```

Giải thích code:

1. Thời gian thực thi tuần tự:

- Chạy task1() trước, sau đó chạy task2().
- Tổng thời gian là **2 giây (task1) + 3 giây (task2) = 5 giây**.

2. Thời gian thực thi dùng luồng:

- Chạy task1() và task2() song song bằng luồng.
- Thời gian thực thi sẽ là **thời gian dài nhất trong các luồng** (3 giây của task2).

3. Mô phỏng công việc:

- Dùng time.sleep() để giả lập các công việc mất thời gian.

4. Đo thời gian thực:

- Sử dụng time.time() trước và sau khi thực thi để đo thời gian.

Kết quả thực hiện chương trình:

ID của tiến trình chính: 20736
Tên của luồng chính: MainThread
--- Thực thi tuần tự ---
Gắn tác vụ 1 đến luồng: MainThread
ID của tiến trình chạy task 1: 20736
Task 1 hoàn thành
Gắn tác vụ 2 đến luồng: MainThread
ID của tiến trình chạy task 2: 20736
Task 2 hoàn thành
Thời gian thực thi tuần tự: 5.01 giây

--- Thực thi với luồng
---Gắn tác vụ 1 đến luồng: t1
ID của tiến trình chạy task 1: 20736
Gắn tác vụ 2 đến luồng: t2
ID của tiến trình chạy task 2: 20736
Task 1 hoàn thành
Task 2 hoàn thành
Thời gian thực thi với luồng: 3.02 giây

So sánh thời gian:

Thực thi tuần tự:

- Các công việc được thực hiện lần lượt.
- Tổng thời gian bằng tổng thời gian của tất cả các công việc (5 giây).

Thực thi song song bằng luồng:

- Các công việc được thực hiện đồng thời.
- Thời gian thực thi bằng thời gian dài nhất của một trong các công việc (3 giây).

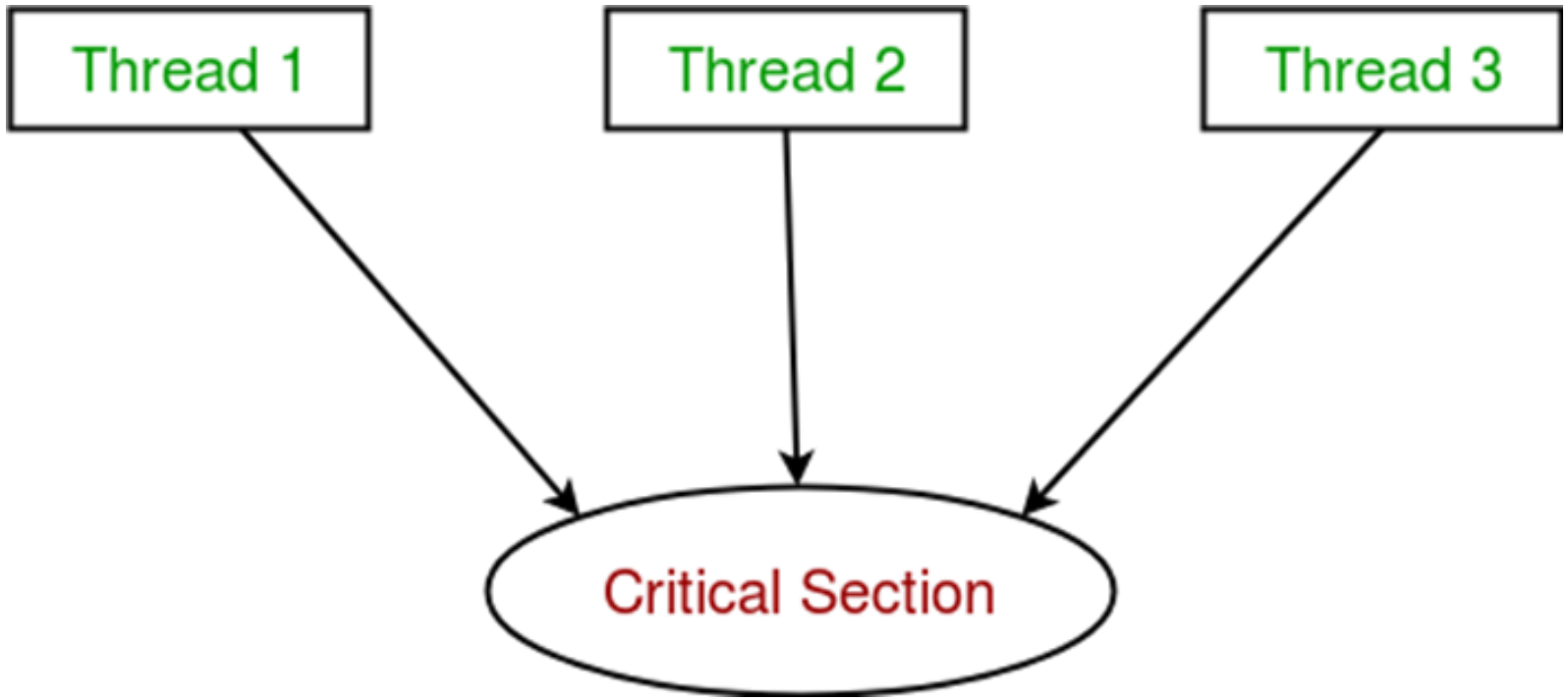
6.4. ĐỒNG BỘ HÓA CÁC THREAD

6.4.1. Khái niệm CS – Critical Sesssion.

- **Critical section** viết tắt (CS) là một phần của mã "**khối mã quan trọng**" hay ***"Đoạn mã găng"***, được đánh dấu là cần được thực thi tuần tự bởi một luồng tại một thời điểm.
- Trong critical section, các tài nguyên có thể chia sẻ như biến, cấu trúc dữ liệu, hoặc tệp tin chỉ được truy cập bởi một luồng duy nhất, trong khi các luồng khác phải đợi cho đến khi critical section được giải phóng
- **Critical section** là đảm bảo tính nhất quán và an toàn trong quá trình truy cập và sửa đổi các tài nguyên chia sẻ. Bằng cách chỉ cho một luồng truy cập critical section tại một thời điểm, ta tránh được các xung đột dữ liệu và đảm bảo rằng các thay đổi được thực hiện một cách đồng nhất.

Thực hiện critical section

Sử dụng các phương pháp và cơ chế đồng bộ hóa như locks, mutexes, hay semaphores để đảm bảo rằng chỉ có một luồng được phép thực thi critical section tại một thời điểm.



Hình 6.5. Minh họa 3 luồng cùng truy cập Critical session tại một thời điểm [17]

Ghi nhớ: *Phần mã gang, quan trọng độc quyền (critical section) đề cập đến các phần của chương trình mà tài nguyên chia sẻ được truy cập.*

6.4.2. Đồng bộ hóa đa luồng

- Đồng bộ hóa luồng (thread synchronization) được định nghĩa là một cơ chế đảm bảo cho hai hoặc nhiều luồng không thực thi đồng thời một số đoạn chương trình cụ thể **'CS-Critical Session'**.[].

- Đồng bộ hóa luồng (thread synchronization) là quá trình điều chỉnh và đồng bộ hóa các hoạt động của các luồng (threads) trong một chương trình đồng thời. Khi có nhiều luồng đang thực thi cùng một tài nguyên chung, việc đồng bộ hóa là cần thiết để đảm bảo tính nhất quán và tránh các vấn đề xung đột dữ liệu.

Một số vấn đề thường gặp khi làm việc với nhiều luồng

- **Race condition (tình huống tương tranh):** Đây là tình huống xảy ra khi hai hay nhiều luồng cùng truy cập và cập nhật cùng một tài nguyên chia sẻ, dẫn đến việc cạnh tranh và thay đổi dữ liệu một cách không đoán trước được. Kết quả cuối cùng có thể không đúng theo mong đợi và thay đổi ngẫu nhiên.
- **Thứ tự thực thi không xác định**: Khi có nhiều luồng, thứ tự thực thi của chúng có thể không được kiểm soát, dẫn đến kết quả không đồng nhất.
- **Deadlock:** Đây là tình huống xảy ra khi các luồng đợi lẫn nhau để nhận tài nguyên mà không bao giờ được giải phóng. Điều này dẫn đến tình trạng bị treo (hang) và không thể tiếp tục thực thi.

Các cơ chế đồng bộ hóa luồng được sử dụng, bao gồm:

- **Locks**: Sử dụng khóa (lock) để chỉ cho một luồng được phép truy cập vào tài nguyên chia sẻ một cách độc quyền. Các luồng khác phải chờ đợi khóa được giải phóng trước khi có thể truy cập.
- **Semaphores (đèn tín hiệu)**: Sử dụng semaphores để quản lý việc truy cập đồng thời vào một tài nguyên có giới hạn. Semaphores cho phép định rõ số lượng luồng được phép truy cập cùng một lúc.
- **Condition variables**: Sử dụng biến điều kiện (condition variable) để cho phép luồng chờ đợi thông báo từ một luồng khác trước khi tiếp tục thực thi.
- **Event objects**: Sử dụng đối tượng sự kiện (event object) để cho phép luồng chờ đợi xảy ra của một sự kiện trước khi tiếp tục thực thi.

Đồng bộ hóa luồng đảm bảo tính và an toàn khi làm việc với các tài nguyên chia sẻ trong môi trường đa luồng. Nó đảm bảo rằng các luồng thực thi theo đúng thứ tự và không xảy ra xung đột dữ liệu

Ví dụ 6.4.1.1. Trường hợp Race condition

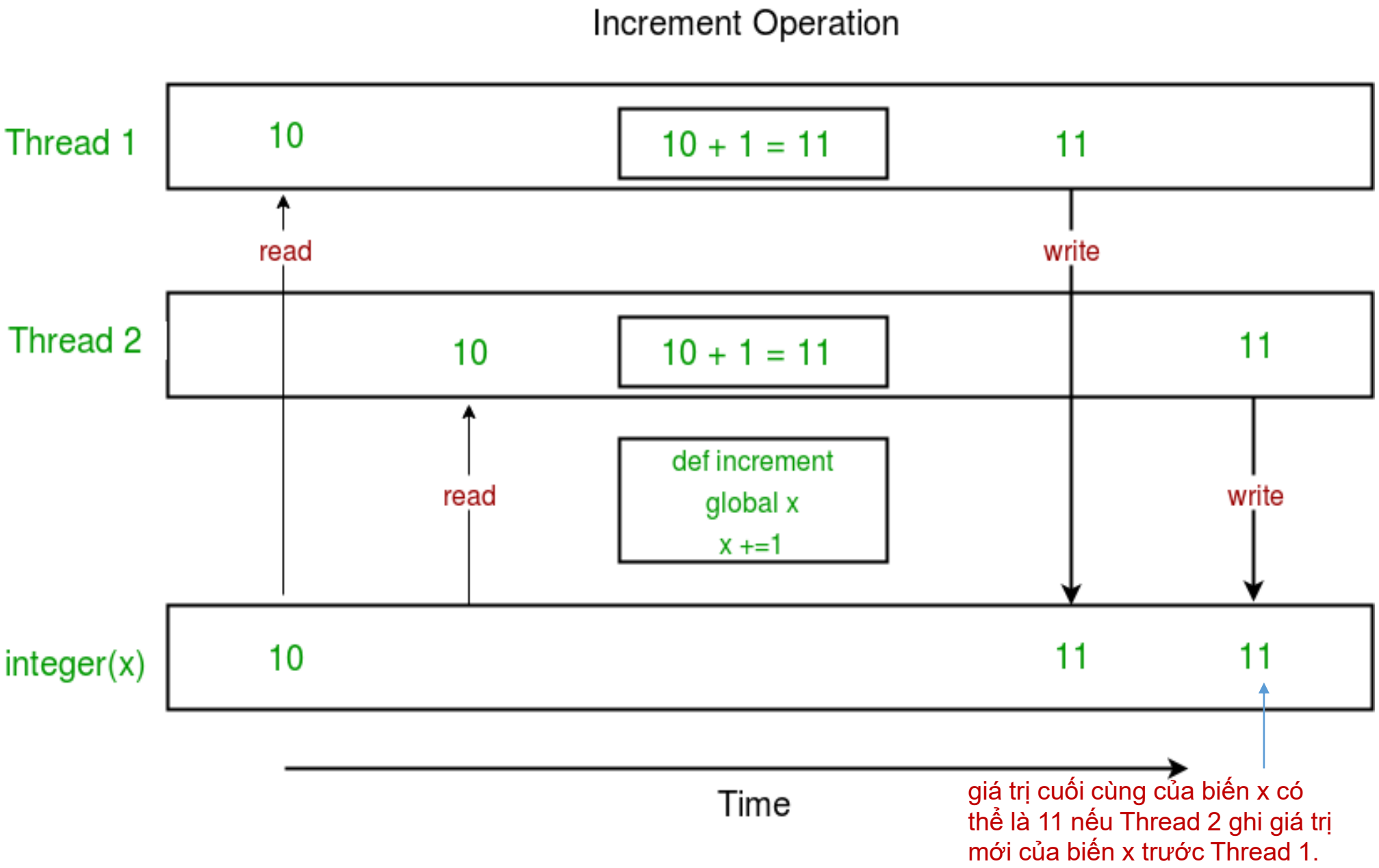
```
import threading

# global variable x
x = 0

#Viết hàm increment(), tăng giá trị biến toàn cục x
def increment():
    global x
    x += 1
#Hàm theread_task() thực hiện tác vụ luồng,  thực hiện gọi hàm
# increment() 100000 lần.'''
def thread_task():
    for _ in range(100000):
        increment()

#Hàm main_task(): thực hiện nhiệm vụ chính của chương trình.
#Nó đặt giá trị của biến toàn cục x=0, tạo hai luồng (t1 và t2)
# để thực thi công việc trong thread_task().
def main_task():
    global x
    x = 0

    # creating threads
    t1 = threading.Thread(target=thread_task)
    t2 = threading.Thread(target=thread_task)

    # start threads
    t1.start()
    t2.start()

    # wait until threads finish their job
    t1.join()
    t2.join()
#Chương trình chạy vòng lặp 10 lần,
#trong mỗi lần lặp gọi main_task() để thực hiện công việc.
#Sau đó in giá trị x ở mỗi lần lặp ra màn hình.
for i in range(10):
    main_task()
    print("Lần lặp {0}: x = {1}".format(i, x))
```

Kết quả chạy chương trình:

```
Lần lặp 0: x = 200000
Lần lặp 1: x = 196410
Lần lặp 2: x = 200000
Lần lặp 3: x = 200000
Lần lặp 4: x = 200000
Lần lặp 5: x = 200000
Lần lặp 6: x = 126749
Lần lặp 7: x = 133861
Lần lặp 8: x = 200000
Lần lặp 9: x = 200000
```

Ở ví dụ 6.4.1.1, lần lặp 1, 6, 7 trả về một số giá trị khác nhau. Hiện tượng này xảy ra do các luồng tương tác đồng thời với biến x, **race condition** có thể xảy ra và dẫn đến kết quả không đồng nhất, khiến giá trị x thay đổi không đoán trước được.

Minh họa race condition



Để khắc phục vấn đề race condition và đảm bảo tính nhất quán của kết quả, chúng ta có thể sử dụng các cơ chế đồng bộ hóa như Lock, Semaphore, hoặc Condition để đảm bảo rằng chỉ một luồng được phép truy cập và cập nhật x tại một thời điểm.

Class lock: cơ chế đồng bộ hóa được sử dụng để đảm bảo chỉ một luồng có thể truy cập và thay đổi dữ liệu trong một khoảng thời gian nhất định.

Lock sử dụng Semaphore (một cơ chế đồng bộ hóa được HĐH cung cấp) để đảm bảo rằng chỉ có một luồng được phép giữ lock và truy cập tài nguyên chia sẻ tại một thời điểm. Khi một luồng giữ lock, Semaphore sẽ đếm giảm và không cho phép các luồng khác giữ lock cho đến khi lock được giải phóng bởi luồng hiện tại.

Lớp lock cung cấp hai phương thức sau:

- **acquire**(): phương thức acquire() để khóa lock trước khi truy cập và thay đổi dữ liệu chia sẻ.
- **release()**: phương thức release() để mở khóa lock sau khi hoàn thành công việc trên dữ liệu chia sẻ.

Ghi nhớ: Điểm quan trọng là lập trình viên nên đảm bảo rằng mọi phương thức acquire() được gọi bởi một luồng được phù hợp với các phương thức release() tương ứng. Mỗi lần gọi acquire() nên có một lần gọi release() tương ứng để đảm bảo tính nhất quán và tránh sự chặn (deadlock) trong các luồng khác.

Ví dụ 6.4.1.2.

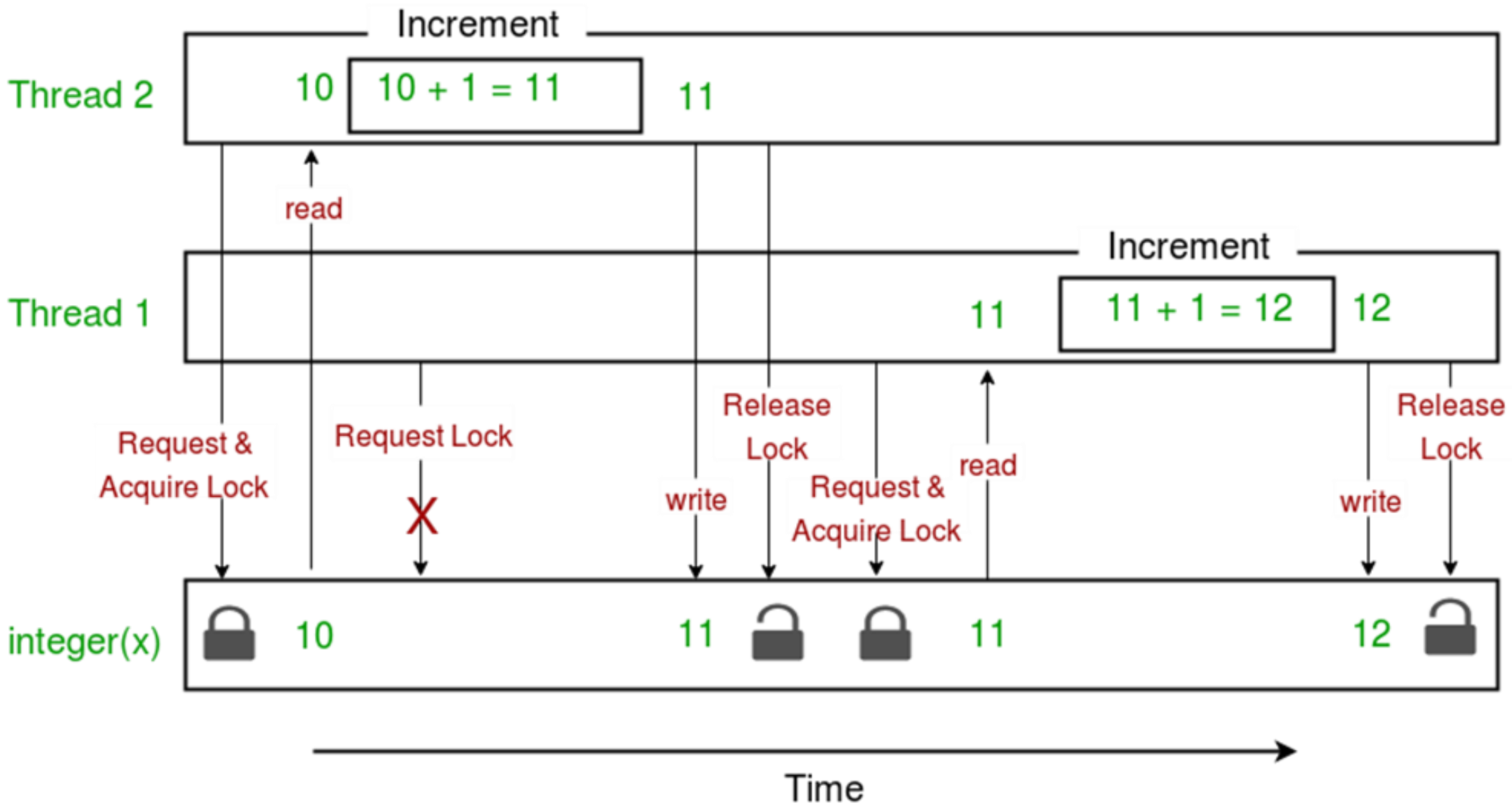
```
1  import threading
2
3  # Khai báo biến toàn cục x và gán giá trị ban đầu là 0.
4  x = 0
5  #Viết hàm increment(), tăng giá trị biến toàn cục x
6  def increment():
7      global x
8      x += 1
9
10 #Hàm theread_task() thực hiện tác vụ luồng, trong task vụ này,
11 # luồng sẽ lặp lại 100000 lần.
12 def thread_task(lock):
13     for _ in range(100000):
14         lock.acquire()   #Yêu cầu lock bằng cách gọi phương thức lock.acquire
15         increment()      #Tăng giá trị của x bằng cách gọi phương thức increment()
16         lock.release()   #Giải phóng lock bằng cách gọi phương thức lock.release()
17
```

```
18  #Định nghĩa hàm main_task() là tác vụ chính của chương trình.
19  def main_task():
20      global x
21      x = 0                       #Đặt biến toàn cục x=0.
22
23      # creating a lock
24      lock = threading.Lock() # tạo ra một lock  bằng cách gọi threading.Lock().
25
26      #  Tạo mới 2 luồng t1, t2 và chạy cùng một lúc, mỗi luồng gọi hàm thread_task() với
27      #   đối số lock.
28      t1 = threading.Thread(target=thread_task, args=(lock,))
29      t2 = threading.Thread(target=thread_task, args=(lock,))
30
31      # start threads
32      t1.start()
33      t2.start()
34
```

```
35      # Chờ cho đến khi các luồng kết thúc công việc
36      t1.join()
37      t2.join()
38
39  if __name__ == "__main__":#Kiểm tra  module đang được chạy là module chính
40                            #hay không.
41      for i in range(10):       # Vòng lặp for chạy 10 lần
42          main_task()           # Mỗi lần lặp main_task() được gọi
43          print("Lần lặp {0}: x = {1}".format(i,x))
```

```
Lần lặp 0: x = 200000
Lần lặp 1: x = 200000
Lần lặp 2: x = 200000
Lần lặp 3: x = 200000
Lần lặp 4: x = 200000
Lần lặp 5: x = 200000
Lần lặp 6: x = 200000
Lần lặp 7: x = 200000
Lần lặp 8: x = 200000
Lần lặp 9: x = 200000
```

Đồng bộ hóa luồng (thread synchronization) thông qua cơ chế Lock để bảo vệ Critical Section (vùng găng)



b. Lớp Semaphonre

- Semaphore duy trì một biến đếm (không âm) được truyền vào khi khởi tạo một Semaphore object, có giá trị giảm sau mỗi lần gọi acquire() và tăng sau mõi lần gọi method release(). Khi gọi method acquire() trên một Semaphore object với giá trị biến đếm bằng 0, nó sẽ block thread gọi và đợi cho đến khi thread khác gọi release() (làm tăng giá trị biến đếm lên 1).

  Semaphore cũng cung cấp 2 method là acquire() và release() tương tự như cơ chế Lock hay RLock, chỉ khác ở chỗ method acquire() sẽ trả về ngay tức thì khi biến đếm có giá trị lớn hơn không.

  Semaphore thường được dùng để giới hạn số lượng thread đồng thời truy cập vào tài nguyên chúng, ví dụ, giới hạn số lượng kết nối tới một database server. Dễ nhận thấy rằng Lock là một trường hợp riêng của Semaphore trong đó biến đếm được khởi tạo bằng 1.

Ví dụ: Giả sử chúng ta có một tài nguyên được chia sẻ là một cây rút tiền ATM. Yêu cầu đặt ra là đảm bảo rằng chỉ có một người dùng có thể sử dụng cây ATM tại một thời điểm.

Chúng ta có thể sử dụng lớp Semaphore để kiểm soát số lượng luồng có thể truy cập vào cây rút tiền ATM.

```
1   import threading
2   class ATM:
3
4       def __init__(self):
5           self.semaphore = threading.Semaphore(1)
6
7       def withdraw(self, amount):
8           self.semaphore.acquire()
9           # Thực hiện giao dịch rút tiền
10          print(f"Đã rút {amount} ngàn từ máy ATM")
11          self.semaphore.release()
```

Giá trị 1 của Semaphore đảm bảo rằng chỉ một luồng có thể thực hiện phương thức withdraw tại một thời điểm.

yêu cầu một permit (đặc quyền) từ Semaphore.

Giải phóng Semaphore, cho phép luồng khác có thể thực hiện giao dịch.

```
12  def main():
13      atm = ATM()
14
15      # Tạo 10 luồng
16      for i in range(10):
17          threading.Thread(target=atm.withdraw, args=(100,)).start()
18  if __name__ == "__main__":
19      main()
```

tạo 10 luồng, mỗi luồng sẽ gọi phương thức Withdraw của đối tượng ATM với đối số là 100.

```
Đã rút 100 ngàn từ máy ATM
Đã rút 100 ngàn từ máy ATM
Đã rút 100 ngàn từ máy ATM
Đã rút 100 ngàn từ máy ATM
Đã rút 100 ngàn từ máy ATM
Đã rút 100 ngàn từ máy ATM
Đã rút 100 ngàn từ máy ATM
Đã rút 100 ngàn từ máy ATM
Đã rút 100 ngàn từ máy ATM
Đã rút 100 ngàn từ máy ATM
```

6.4.3. Queue Module: quyền ưu tiên đa luồng trong Python

Mô-đun Hàng đợi chủ yếu được sử dụng để quản lý xử lý lượng lớn dữ liệu trên nhiều luồng. Nó hỗ trợ việc tạo một đối tượng hàng đợi mới có thể lấy một số mục riêng biệt.

Danh sách các hoạt động được sử dụng để quản lý Hàng đợi:

- get(): Xóa và trả về một item từ queue.
- put(): Thêm một item tới một queue.
- qsize() : Trả về số item mà hiện tại đang trong queue.
- empty(): Trả về true nếu queue là trống, nếu không thì trả về false.
- full(): Trả về true nếu queue là đầy, nếu không thì trả về false.

Hàng đợi ưu tiên là phần mở rộng của hàng đợi với các thuộc tính sau:

- Phần tử có mức ưu tiên cao được xếp trước phần tử có mức ưu tiên thấp.
- Nếu hai phần tử có cùng mức độ ưu tiên, chúng sẽ được phục vụ theo thứ tự của chúng trong hàng đợi.

Ví dụ 6.4.3.1

```
1  import queue
2  import threading
3  import time
4
5  thread_exit_Flag = 0
6
7  class sample_Thread (threading.Thread):
8     def __init__(self, threadID, name, q):
9        threading.Thread.__init__(self)
10       self.threadID = threadID
11       self.name = name
12       self.q = q
13    def run(self):
14       print ("initializing " + self.name)
15       process_data(self.name, self.q)
16       print ("Exiting " + self.name)
17
```

```
18  # helper function to process data
19  def process_data(threadName, q):
20      while not thread_exit_Flag:
21          queueLock.acquire()
22          if not workQueue.empty():
23              data = q.get()
24              queueLock.release()
25              print ("% s processing % s" % (threadName, data))
26          else:
27              queueLock.release()
28              time.sleep(1)
29
30  thread_list = ["Thread-1", "Thread-2", "Thread-3"]
31  name_list = ["A", "B", "C", "D", "E"]
32  queueLock = threading.Lock()
33  workQueue = queue.Queue(10)
34  threads = []
```

	initializing Thread-1 initializing Thread-2 initializing Thread-3 Thread-3 processing A Thread-3 processing B Thread-3 processing C Thread-3 processing D Thread-3 processing E Exiting Thread-3 Exiting Thread-1 Exiting Thread-2 Exit Main Thread

CÂU HỎI THẢO LUẬN:

1. Multithreading trong Python là gì?
2. Giải thích khái niệm Thread trong Python?
3. Có bao nhiêu Thread chạy trong cùng một thời điểm trong Python?
4. Nêu một số lợi ích khi sử dụng đa luồng trong Python?
5. Sự khác nhau giữa tiến trình và luồng là gì?
6. Tạo sao lập trình viên cần các locks trong khi viết code đa luồng?
7. Anh/Chị hiểu gì về đồng bộ hóa khi nói đến đa luồng trong Python?
8. Cho biết số cách phổ biến nhằm triển khai khóa trong Python để truy cập đồng thời an toàn vào tài nguyên được chia sẻ theo nhiều luồng?
9. Cho biết điều gì xảy ra nếu hai luồng cố gắng lấy cùng một khóa đồng thời?
10. Anh/Chị cho biết có bất kỳ giới hạn hoặc hạn chế nào đối với việc sử dụng mô-đun Threading trong Python không?

BÀI TẬP VẬN DỤNG

Bài tập 6.1. 6.10 trang 300,301 TLHT