1. **Thread là gì?**

"Thread" thường được sử dụng để chỉ một luồng thực thi độc lập trong một tiến trình. Một tiến trình có ít nhất 1 luồng để chính để thực hiện code của chương trình. Một tiến trình có thể có nhiều luồng.

**Multi threading**có thể hiểu là một kỹ thuật để cùng**một lúc xử lý nhiều tác vụ**. VD: Khi duyệt 1 trang web, có rất nhiều hình ảnh, CSS, javascript… được tải đồng thời bởi các luồng khác nhau.

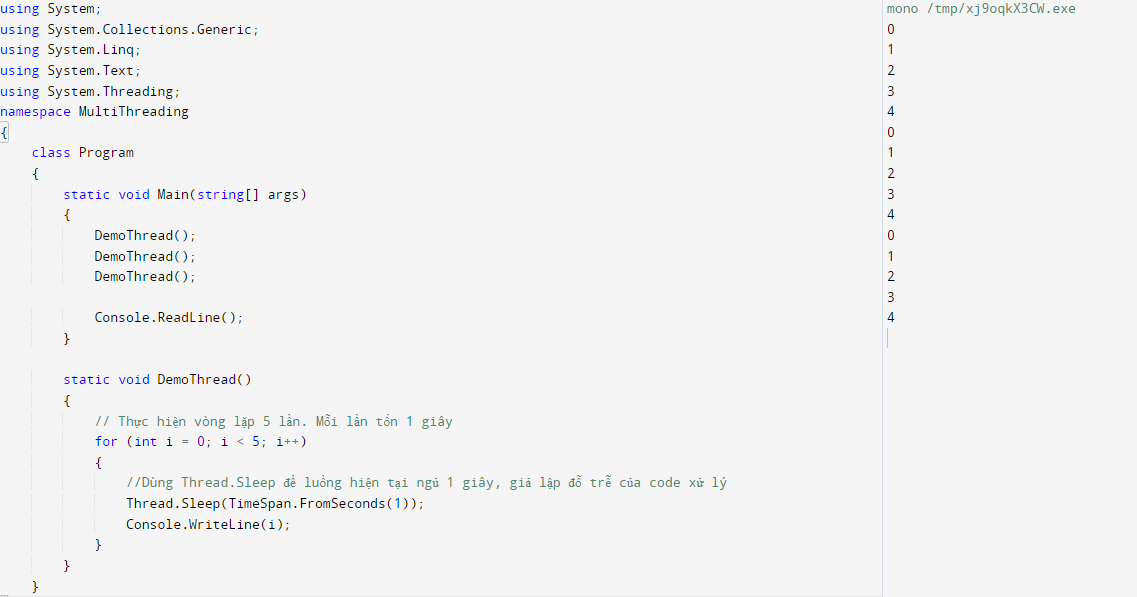
Bản chất chương trình [C#](https://www.howkteam.vn/learn/lap-trinh/lap-trinh-c-net-7-5) được tạo ra sẽ có **hai luồng chạy chính**. Luồng thứ nhất là MainThread (luồng chính của chương trình mặc định là hàm Main) và UIThread (luồng cập nhật giao diện).

Các code xử lý mà chúng ta code chính là nằm trên luồng MainThread. Và chương trình **sẽ thực hiện tuần tự từng dòng code từ trên đi xuống**. Nên nếu chúng ta có 3 hàm xử lý cần chạy cùng lúc là không thể. Vì chương trình **phải đợi hàm trước đó xử lý xong** mới đến hàm kế tiếp.

Vậy để giải quyết việc cùng một lúc, có thể xử lý nhiều tác vụ, multi threading ra đời

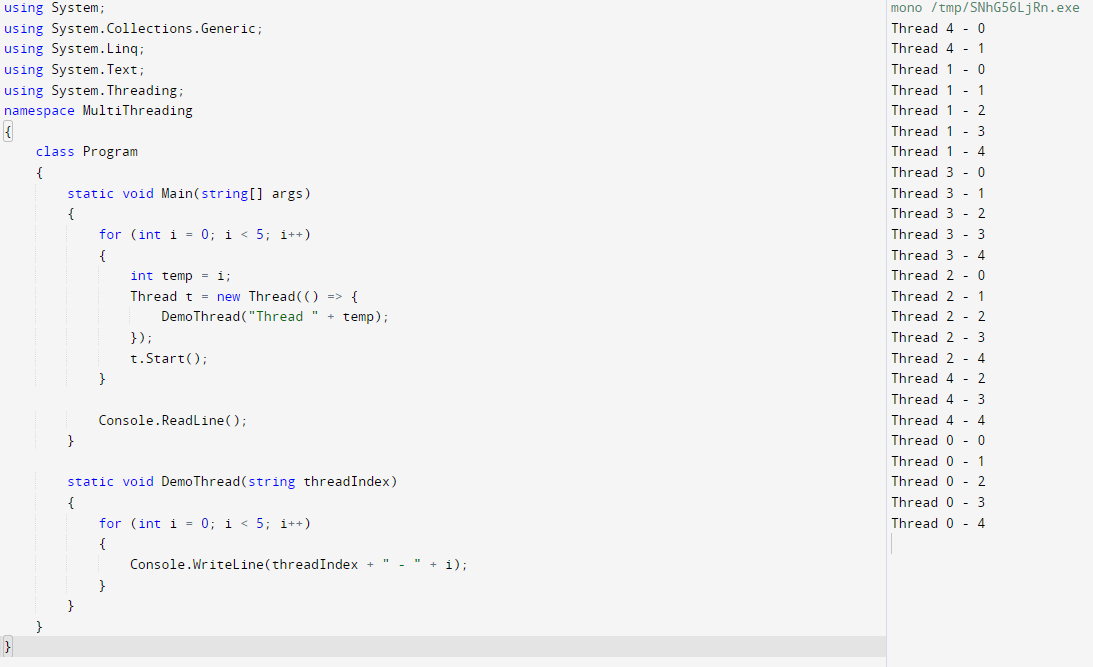
1. **Cách dùng Thread trong C#:**

Class Thread nằm trong thư viện System.Threading của **.Net**. Để có thể sử dụng Thread thì chúng ta sẽ using System.Threading.



Hàm DemoThread với mục đích là giả lập lại một hàm xử lý mất thời gian 5 giây.

Kết quả có thể thấy màn hình Console sẽ in ra giá trị của I sau mỗi 1 giây. Sau 5 giây thì hoàn thành. Vậy nếu chúng ta gọi 3 hàm DemoThread này liên tục sẽ tiêu tốn 15s để hoàn thành chương trình.



Bây giờ chúng ta sẽ cho mỗi hàm Demo vào một luồng riêng biệt để thực hiện. Lúc này tốc độ sẽ tăng lên đáng kể vì 3 luồng chạy song song nhau. Chương trình sẽ chỉ mất 5 giây để hoàn thành.

1. **Lợi ích của multi-threading bao gồm:**

1. Duy trì giao diện người dùng phản hồi: Khi một chương trình thực hiện các tác vụ dài, tốn thời gian trên luồng giao diện người dùng, giao diện có thể bị đóng băng và trở nên không phản hồi. Bằng cách sử dụng multi-threading, các tác vụ tốn thời gian có thể được thực hiện trên các luồng bổ sung, đồng thời cho phép giao diện tiếp tục phản hồi và tương tác với người dùng một cách mượt mà.

2. Tận dụng thời gian xử lý của bộ xử lý: Khi chương trình đang chờ đợi các hoạt động nhập/ xuất (I/O) hoàn thành, như đọc dữ liệu từ đĩa cứng hoặc lấy dữ liệu từ mạng, việc sử dụng multi-threading cho phép các luồng khác tiếp tục thực hiện các tác vụ khác trong khi chờ đợi. Điều này giúp tận dụng tối đa thời gian xử lý của bộ xử lý, giảm thời gian chờ đợi và làm cho chương trình hoạt động hiệu quả hơn.

3. Chia nhỏ các tác vụ lớn: Multi-threading cho phép chia nhỏ các tác vụ lớn, đòi hỏi nhiều xử lý từ CPU, để thực hiện đồng thời trên các máy tính có nhiều bộ xử lý hoặc nhân CPU. Điều này giúp tăng hiệu suất và tốc độ xử lý của chương trình, đặc biệt là trong các ứng dụng có tính toán phức tạp hoặc xử lý dữ liệu lớn.

1. **Nhược điểm của multi-threading:**

Trên máy tính chỉ có một lõi/ bộ xử lý, việc sử dụng multi-threading có thể ảnh hưởng đến hiệu suất một cách tiêu cực do sự tăng overhead liên quan đến việc chuyển đổi ngữ cảnh (context-switching). Khi các luồng phải chia sẻ một lõi/ bộ xử lý đơn lẻ, việc chuyển đổi giữa các luồng yêu cầu thời gian và tài nguyên, gây giảm hiệu suất và tốc độ xử lý của chương trình.

Để hoàn thành cùng một nhiệm vụ, cần phải viết nhiều dòng mã hơn. Multi-threading đòi hỏi phải viết mã để quản lý và đồng bộ hóa các luồng, điều này có thể làm tăng độ phức tạp và độ dài của mã. Việc viết và duy trì mã multi-threaded có thể phức tạp hơn so với viết mã tuần tự đơn giản.

Ứng dụng multi-threaded khó khăn trong việc viết, hiểu, gỡ lỗi và bảo trì. Với multi-threading, các vấn đề liên quan đến đồng bộ hóa, race condition, deadlock và livelock có thể xảy ra. Điều này đòi hỏi kỹ năng và kiến thức cao để viết mã an toàn và đảm bảo tính nhất quán của các luồng. Sửa lỗi và bảo trì các ứng dụng multi-threaded cũng có thể phức tạp hơn do tính tương tác và phụ thuộc giữa các luồng.

Tóm lại, nhược điểm của multi-threading bao gồm việc ảnh hưởng tiêu cực đến hiệu suất trên máy tính chỉ có một lõi/ bộ xử lý, việc phải viết nhiều dòng mã hơn để hoàn thành cùng một nhiệm vụ và sự khó khăn trong việc viết, hiểu, gỡ lỗi và bảo trì các ứng dụng multi-threaded.

1. **Thread life cycle:**

1. Trạng thái chưa khởi động: Khi một thể hiện của lớp Thread được tạo ra, nó mặc định ở trạng thái chưa khởi động (unstarted state).

2. Trạng thái sẵn sàng chạy:

Khi phương thức start() được gọi trên luồng, nó ở trạng thái sẵn sàng chạy (runnable state) hoặc sẵn sàng để chạy.

3. Trạng thái đang chạy: Chỉ một luồng trong một quá trình có thể được thực thi tại một thời điểm. Trong quá trình thực thi, luồng ở trạng thái đang chạy (running state).

4. Trạng thái không sẵn sàng chạy: Luồng ở trạng thái không sẵn sàng chạy (not runnable state) nếu phương thức sleep() hoặc wait() được gọi trên luồng, hoặc hoạt động nhập/ xuất bị chặn.

5. Trạng thái đã kết thúc: Sau khi hoàn thành công việc, luồng chuyển vào trạng thái đã kết thúc (dead or terminated state).

-------

Starting a thread :

Thread thread= new Thread(new ThreadStart(ThreadFunc));

//Creates a thread object// ThreadStartidentifies the method that the thread executes when it starts

thread.Start();

//starts the thread running

1. **Thread Priorities :**

Controls the amount of CPU time that can be allotted to a thread.

ThreadPriority.Highest

ThreadPriority.AboveNormal

ThreadPriority.NormalThreadPriority.BelowNormalThreadPriority.Lowest

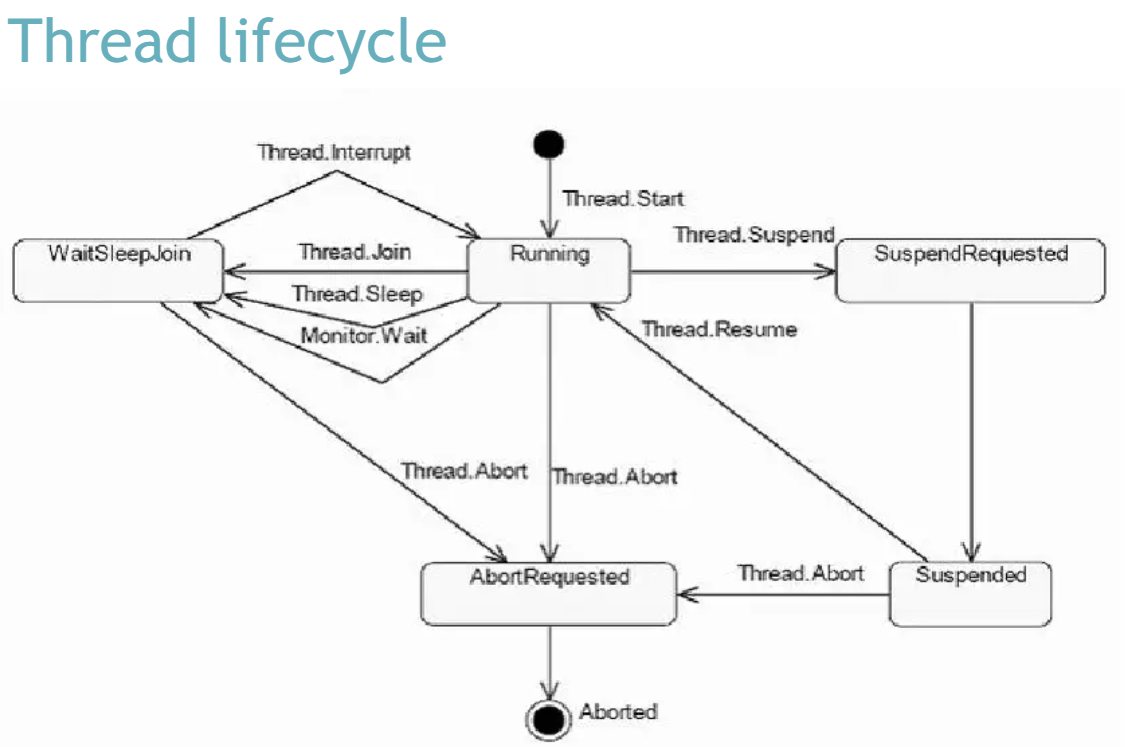
Các mức độ ưu tiên luồng (thread priorities) là các giá trị được sử dụng để kiểm soát lượng thời gian xử lý CPU có thể được cấp cho một luồng. Dưới đây là một số mức độ ưu tiên thông thường được sử dụng trong các hệ thống đa luồng:

1. ThreadPriority.Highest: Mức độ ưu tiên cao nhất. Luồng với mức độ ưu tiên này được ưu tiên xử lý hơn các luồng khác và có cơ hội lớn nhất để được giao thời gian xử lý CPU.

2. ThreadPriority.AboveNormal: Mức độ ưu tiên trên trung bình. Luồng với mức độ ưu tiên này có mức độ ưu tiên cao hơn so với các luồng có mức độ ưu tiên bình thường và thường được ưu tiên trong việc xử lý.

3. ThreadPriority.Normal: Mức độ ưu tiên bình thường. Đây là mức độ ưu tiên mặc định cho luồng. Luồng với mức độ ưu tiên này được xử lý theo cách thông thường và không có ưu tiên đặc biệt.

4. ThreadPriority.BelowNormal: Mức độ ưu tiên dưới trung bình. Luồng với mức độ ưu tiên này có mức độ ưu tiên thấp hơn so với các luồng có mức độ ưu tiên bình thường và thường được ưu tiên ít hơn.

5. ThreadPriority.Lowest: Mức độ ưu tiên thấp nhất. Luồng với mức độ ưu tiên này có mức độ ưu tiên thấp nhất và thường được xử lý sau tất cả các luồng khác. Các mức độ ưu tiên này cho phép bạn điều chỉnh cách mà hệ điều hành quản lý thời gian xử lý CPU giữa các luồng khác nhau trong ứng dụng đa luồng.

+Thread method

* Start(): phương thức này dùng để bắt đầu một thread, khi mà thread hoàn thành, nó không thể restart
* Suspend(): phương thức này sẽ tạm dừng việc thực thi của Thread vô thời hạn cho đến khi nó được yêu cầu chạy tiếp tục với phương thức Resume().
* Tuy nhiên hai phương thức này được gắn attribute Obsolete để khuyến cáo rằng nên sử dụng những phương pháp khác để thay thế.
* Abort(): khi phương thức này được gọi, hệ thống sẽ ném ra một ngoại lệ ThreadAbortException để kết thúc thread. Sau khi gọi phương thức này, thuộc tính ThreadState sẽ chuyển sang giá trị Stopped.
* Resume(): phương thức này giúp một luồng đang tạm dừng có thể được tiếp tục chạy. Cũng giống như Suspend(), phương thức này cũng bị khuyến cáo.
* phương thức Interrupt() được sử dụng để gửi một tín hiệu gián đoạn (interrupt signal) đến một luồng (thread) đang chạy, để yêu cầu nó dừng lại hoặc thoát khỏi trạng thái chờ đợi (waiting state) hoặc trạng thái ngủ (sleeping state) và tiếp tục thực hiện các công việc khác.

