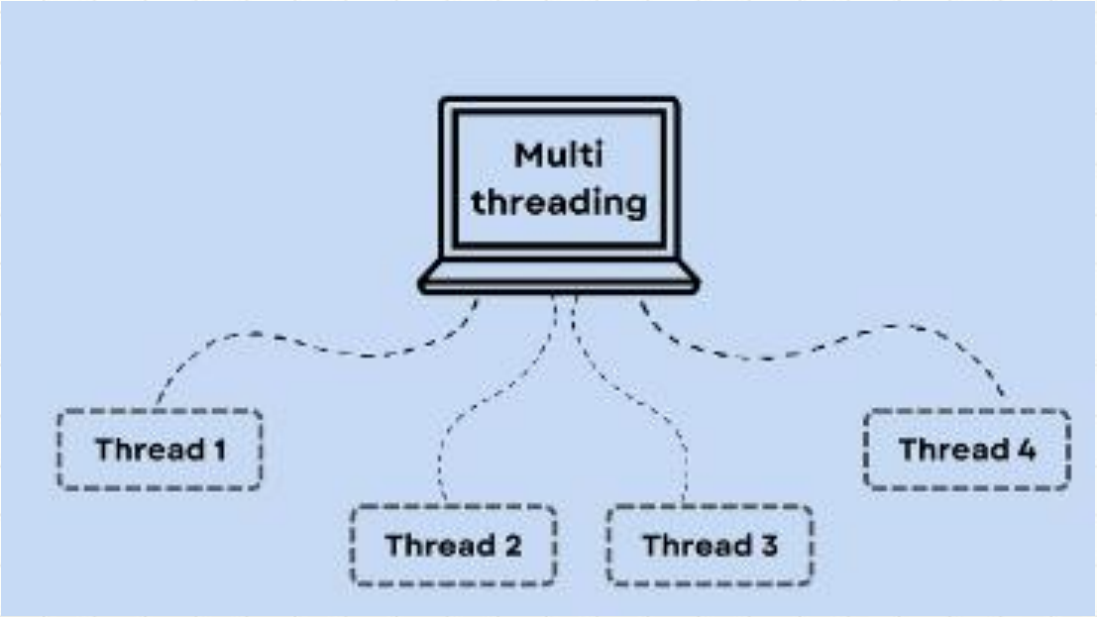
LẬP TRÌNH ĐA LUỒNG
(MULTITHREADING)



Nội dung

1	Tổng quan về tiến trình, luồng và đa luồng
2	Tạo và quản lý luồng trong C#
3	Các giải pháp tránh xung đột
4	Sử dụng đa luồng trong ứng dụng WinForms

Tổng quan

1. PROCESS là gì?

Process (tiến trình) là **một chương trình đang chạy** trong hệ điều hành.

Mỗi process có **không gian nhớ độc lập**, nghĩa là:

- Có vùng nhớ riêng (RAM)
- Có các biến, dữ liệu, stack, heap riêng
- Không chia sẻ dữ liệu trực tiếp với process khác
- Hệ điều hành quản lý process (khởi tạo, tạm dừng, kết thúc)

Một chương trình chạy 2 lần ⇒ 2 process hoàn toàn độc lập.

Ví dụ: Mở 2 cửa sổ Word sẽ tạo ra 2 process Word.

Tổng quan

2. THREAD là gì?

Thread (luồng) là **đơn vị thực thi nhỏ nhất** bên trong một process, còn được gọi là tiến trình con (tiểu trình).

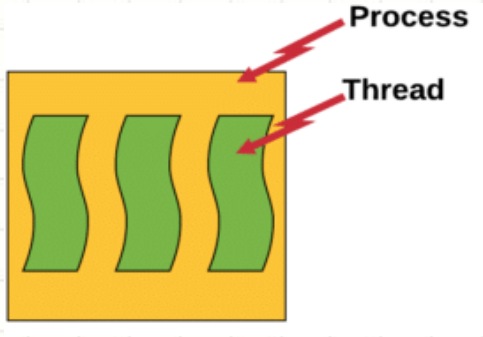
- Một process **có ít nhất 1 thread** (gọi là Main thread)
- Các thread trong cùng process **chia sẻ chung bộ nhớ**
- Mỗi thread có **stack riêng**, nhưng **dùng chung heap**
- Tạo thêm thread ⇒ chương trình có thể chạy song song nhiều công việc

Vì chia sẻ chung bộ nhớ nên thread giao tiếp dễ, nhưng dễ gây lỗi race condition (tranh chấp tài nguyên), deadlock… nếu không đồng bộ.

Tổng quan

3. SO SÁNH PROCESS VÀ THREAD

Tiêu chí	Process	Thread
Bộ nhớ	Riêng biệt	Chia sẻ trong cùng 1 process
Tốc độ tạo	Chậm	Nhanh
Khả năng giao tiếp	Khó – IPC (pipe, socket…)	Dễ – chia sẻ chung heap
Ảnh hưởng lỗi	1 process crash không ảnh hưởng process khác	1 thread crash có thể làm sập process
Số lượng	Thường ít	Nhiều



Tổng quan

4. MULTITHREADING là gì?

Multithreading (đa luồng) là kỹ thuật lập trình cho phép ứng dụng tạo ra và chạy nhiều luồng cùng lúc.

Với ứng dụng WinForms, có 2 luồng mặc định:

- MainThread (luồng chính của ứng dụng, chạy trong hàm Main)
- UIThread (luồng cập nhật giao diện)

Khi nào cần đa luồng?

- Chạy tác vụ nền (background)
- Nhiều công việc song song: tải file, xử lý dữ liệu
- Giao diện (UI) không bị treo khi đang xử lý nặng (WinForms/WPF)

Tổng quan

5. CÁC RỦI RO KHI DÙNG MULTITHREADING:

- Race condition (tranh chấp tài nguyên)
- Deadlock
- Dữ liệu bị ghi đè
- Debug khó hơn

Tạo và quản lý luồng

Sử dụng namespace System.Threading để làm việc trên đa luồng

Các bước tạo và chạy thread:

1. Tạo phương thức (gọi là phương thức callback) mà sẽ thực thi khi thread chạy. Phương thức này không có giá trị trả về (void) và phải không có tham số hoặc chỉ có một tham số là kiểu object. Nếu dùng lambda expression (hàm ẩn danh) thì có thể bỏ qua bước này

```
void Callback_Method(){…}
```

2. Tạo đối tượng Thread và truyền một delegate ThreadStart chứa phương thức callback sẽ thực thi ở trên vào constructor của Thread

```
Thread t = new Thread(new ThreadStart(Callback_Method));
```

Hoặc ngắn gọn hơn:

```
Thread t = new Thread(Callback_Method);
```

3. Chạy thread bằng cách gọi phương thức Start() của đối tượng Thread vừa tạo:

```
t.Start();
```

Tạo và quản lý luồng

Truyền tham số vào thread:

Giả sử ta có hàm callback như sau:

```
static void DoWork(string workName)
{
    Console.WriteLine(workName);
}
```

Cách 1: dùng ParameterizedThreadStart. Khi đó hàm callback cần viết lại để chuyển tham số thành object và ép kiểu khi dùng

```
static void DoWork(object workName)
{
    Console.WriteLine((string)workName);
}
```

Tạo và chạy thread với tham số:

```
static void Main(string[] args)
{
    Thread t = new Thread(DoWork);
    t.Start("work A");
}
```

Tạo và quản lý luồng

Cách 2: dùng lambda expression. Khi đó hàm callback không cần thay đổi

Tạo và chạy thread với tham số:

```
static void Main(string[] args)
{
    Thread t = new Thread(()=>{ DoWork("work A");});
    t.Start();
}
```

Hoặc:

```
static void Main(string[] args)
{
  Thread t = new Thread((obj)=>{DoWork((string)obj);});
  t.Start("work A");
}
```

Tạo và quản lý luồng

Các loại thread:

- Foreground thread (luồng tiền cảnh)
- Background thread (luồng hậu cảnh)

Tiêu chí	Foreground Thread	Background Thread
Khi chương trình kết thúc	Giữ chương trình tiếp tục chạy cho đến khi thread hoàn tất	CLR sẽ tự động kết thúc thread ngay lập tức
Ưu tiên của CLR	Quan trọng – được thực thi đến cùng	Không quan trọng – có thể bị dừng giữa chừng
Ứng dụng	Tác vụ quan trọng (ghi file, xử lý giao dịch…)	Tác vụ phụ, không quan trọng (log, cleanup, …)

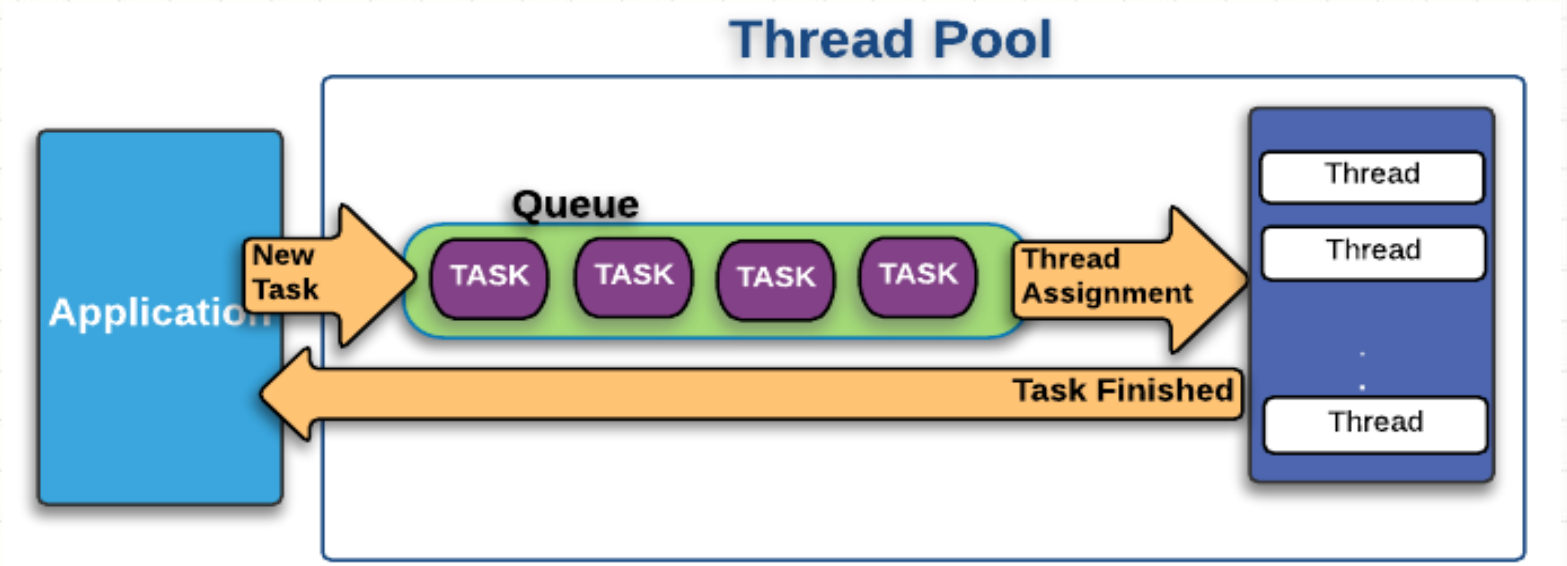
Một thread khi được tạo ra, mặc định là foreground. Khi cần chuyển sang background, ta gán thuộc tính IsBackground là True

Tạo và quản lý luồng

ThreadPool: (bể chứa luồng)

Là một tập hợp các luồng được hệ thống quản lý sẵn để thực thi các tác vụ (task, work item) một cách hiệu quả

- Thay vì tạo ra các luồng theo yêu cầu, ta sẽ tạo ra hẳn một số lượng luồng cố định
- Tác vụ sẽ được chuyển vào hàng đợi
- Các luồng sẽ liên tục kiểm tra hàng đợi, nếu một luồng tìm thấy task, nó sẽ lấy ra, xử lý, rồi trả về kết quả



Tạo và quản lý luồng

Những ưu điểm của ThreadPool:

- **Không phải tự tạo hàng trăm luồng thủ công**

  Tạo luồng rất tốn chi phí CPU và bộ nhớ.
- **Tối ưu hiệu năng**

  ThreadPool tự điều chỉnh số luồng dựa trên tải (workload).
- **Dùng cho các tác vụ ngắn, thực thi nhanh**

  - IO async
  - Xử lý sự kiện
  - Queue công việc nhỏ
  - Xử lý request song song
- **Tránh tình trạng "tạo quá nhiều luồng"**

  Ứng dụng chạy mượt hơn, ổn định hơn.

Tạo và quản lý luồng

Những đặc trưng của ThreadPool:

Thuộc tính	Mô tả
Tự quản lý số thread	Hệ thống tự tăng/giảm thread
Thread	Chỉ chứa background thread nên app có thể thoát dù chưa xong
Không nên dùng cho việc chạy lâu	Làm nghẽn pool
Không hỗ trợ đặt tên thread	Vì thread được dùng lại
Không nên dùng khi cần thao tác UI	Chạy trong thread phụ

Tạo và quản lý luồng

So sánh ThreadPool với Thread tự tạo:

Tiêu chí	ThreadPool	Thread tự tạo
Tạo thread	Rất nhanh (dùng lại)	Tốn thời gian
Số lượng	Bị giới hạn theo CPU & pool	Không giới hạn (dễ gây crash)
Chạy dài hạn	□ Không nên	✔ phù hợp
Dừng thread	Không cho dừng thủ công	Cho dừng
Ưu tiên	Không chỉnh được	Có
Tên thread	Không đặt được	Đặt được
Khi app dừng	Mất ngay (do là background thread)	Foreground thread có thể giữ app lại

Tạo và quản lý luồng

Nên dùng ThreadPool khi nào:

Trường hợp	Có nên dùng ThreadPool?
Tác vụ ngắn (dưới 1 giây)	✔ Rất nên
Xử lý nhiều task nhỏ	✔
Xử lý IO (đọc file, query nhẹ)	✔
Server xử lý hàng trăm request	✔
Task chạy dài 10–20 giây	□ Không nên
Dùng trong WinForms/WPF để cập nhật UI	□ Không trực tiếp (gọi qua Invoke)

Tạo và quản lý luồng

Cú pháp chạy task dùng ThreadPool:

```
ThreadPool.QueueUserWorkItem(<Task>, [<ParamObject>]);
```

Hoặc dùng lambda expression:

```
ThreadPool.QueueUserWorkItem((state)=>{.....});
```

Tạo và quản lý luồng

```
static void Main(string[] args) {
    // Queue two work items to the ThreadPool
    ThreadPool.QueueUserWorkItem(WorkItem1, "Hello");
    ThreadPool.QueueUserWorkItem(WorkItem2, 42);

    // Wait for a moment
    Thread.Sleep(1000);

    Console.WriteLine("Main thread exits");
}

// A work item that prints a message
static void WorkItem1(object state) {
    Console.WriteLine($"WorkItem1: {(string)state}");
}

// A work item that calculates and prints the square of a number, 100 times
static void WorkItem2(object state) {
    int num = (int)state;
    int square = num * num;
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine($"WorkItem2: {i}: {num} * {num} = {square}");
        Thread.Sleep(20);
    }
}
```

Tạo và quản lý luồng

Minh họa cách chờ các tác vụ chạy xong

```
static void Main(string[] args)
{
    int jobCount = 10;
    CountdownEvent countdown = new CountdownEvent(jobCount);
    for (int i = 1; i <= jobCount; i++)
    {
        int taskNumber = i;

        ThreadPool.QueueUserWorkItem((state) =>
        {
            Console.WriteLine($"Task {taskNumber} starts");
            Thread.Sleep(500);
            Console.WriteLine($"Task {taskNumber} done");
            countdown.Signal();
        });
    }
    countdown.Wait();
    Console.WriteLine("Main thread done");
}
```

Tạo và quản lý luồng

Hiện tại đã thay thế cách chạy task qua lời gọi trực tiếp đến ThreadPool.QueueUserWorkItem bằng Task Parallel Library (TPL):

- Task.Run()
- Parallel.For()