LAB 2.1: QUẢN LÝ TIẾN TRÌNH (PHẦN 1)

(2312678 – Nguyễn Lê Bảo Long – CTK47A)

# Mục tiêu

* Hiểu về tiến trình, trạng thái tiến trình
* Hiểu các thuật toán lập lịch CPU (CPU scheduling)

# Nội dung

## Trạng thái tiến trình

1. Cho đoạn code sau:

/\* helloworld.c \*/ #include <stdio.h> int main() {

printf("Hello, World!"); return 0;

}

Biên dịch và thực thi chương trình (Dùng hệ điều hành Windows hoặc Linux). Trong hệ thống sẽ có một tiến trình được tạo ra, thực thi và kết thúc. Cho biết chuỗi trạng thái của tiến trình trên?

***Ghi chú:*** Biên dịch vào thực thi trên Linux

gcc helloworld.c –o helloworld

./helloworld

Chuỗi trạng thái của tiến trình trong hệ điều hành Linux (hoặc các hệ điều hành tương tự) thường trải qua các trạng thái sau:

1. **Created (New)**: Tiến trình được tạo ra khi bạn chạy lệnh ./helloworld. Lúc này, hệ điều hành sẽ cấp phát các tài nguyên cần thiết (như bộ nhớ, PID, v.v.) cho tiến trình.
2. **Ready**: Tiến trình được đưa vào hàng đợi của hệ điều hành, chờ được lập lịch để thực thi trên CPU.
3. **Running**: Khi CPU bắt đầu thực thi các lệnh trong chương trình helloworld, tiến trình chuyển sang trạng thái **Running**. Trong trạng thái này, lệnh printf("Hello, World!"); được thực thi, và thông điệp "Hello, World!" được in ra màn hình.
4. **Terminated**: Sau khi thực thi xong lệnh return 0;, tiến trình kết thúc và chuyển sang trạng thái **Terminated**. Hệ điều hành sẽ thu hồi các tài nguyên đã cấp phát cho tiến trình.

Do chương trình rất đơn giản (chỉ in một dòng và kết thúc) và không có thao tác chờ đợi (I/O, sleep, v.v.), tiến trình không chuyển sang trạng thái **Waiting**.  
**Chuỗi trạng thái:**  
**Created (New) → Ready → Running → Terminated**

1. Cho biết chuỗi trạng thái của tiến trình khi thực thi đoạn mã sau

main()

{

int i=1; while(i<100)

i = i+1; // increment i

cout << i; // print value of i while(i>0)

i = i -1; // decrement i

cout << i; // print value of i

}

**Chuỗi trạng thái của tiến trình:**

* **Created (New)**: Tiến trình được tạo ra khi bạn chạy chương trình. Hệ điều hành cấp phát tài nguyên (bộ nhớ, PID, v.v.) cho tiến trình.
* **Ready**: Tiến trình được đưa vào hàng đợi của hệ điều hành, chờ được lập lịch để thực thi trên CPU.
* **Running**: Tiến trình bắt đầu thực thi các lệnh trong hàm main():
  + Khởi tạo biến i = 1.
  + Vòng lặp while (i < 100) thực hiện tăng giá trị của i từ 1 lên 100.
  + In giá trị của i (lúc này i = 100).
  + Vòng lặp while (i > 0) thực hiện giảm giá trị của i từ 100 xuống 0.
  + In giá trị của i (lúc này i = 0).
* **Terminated**: Sau khi thực thi xong lệnh return 0;, tiến trình kết thúc và chuyển sang trạng thái **Terminated**. Hệ điều hành thu hồi các tài nguyên đã cấp phát cho tiến trình.
* **Created (New) → Ready → Running → Terminated**

1. Cho đoạn chương trình sau, sau khi kết thúc thì chương trình đã nằm trong hàng

đợi waiting bao nhiêu lần?

int main()

{

int a;

for (int i = 1; i < 5; i++)

{

if ( i % 2 == 0)

printf(“Hello world\n"); else a = 5\*i;

}

return 0;

}

* **Phân tích đoạn mã:**

**Vòng lặp for**:

Vòng lặp chạy từ i = 1 đến i = 4 (vì điều kiện i < 5).

Tổng cộng, vòng lặp thực hiện 4 lần lặp.

**Điều kiện if**:

Nếu i % 2 == 0 (tức i là số chẵn), chương trình in ra "Hello world".

Nếu i % 2 != 0 (tức i là số lẻ), chương trình gán a = 5 \* i.

**Các giá trị của i và hành động tương ứng**:

i = 1 (lẻ): a = 5 \* 1 = 5.

i = 2 (chẵn): In ra "Hello world".

i = 3 (lẻ): a = 5 \* 3 = 15.

i = 4 (chẵn): In ra "Hello world".

Trong hệ điều hành, một tiến trình có thể chuyển sang trạng thái Waiting khi nó phải chờ đợi một sự kiện nào đó xảy ra. Ví dụ:

* Chờ dữ liệu từ bàn phím (như khi dùng scanf).
* Chờ đọc/ghi file hoàn thành.
* Chờ một tài nguyên hệ thống khác được giải phóng.

Tuy nhiên, chương trình này chỉ thực hiện các phép toán và in ra màn hình, không có bất kỳ lệnh nào yêu cầu chờ đợi. Vì vậy, nó chạy từ đầu đến cuối mà không cần vào trạng thái **Waiting**.

* **Kết luận :** Kết quả là số lần nằm trong hàng đợi **Waiting** là **0**.

## Lập lịch CPU

1. Cho bảng mô tả các tiến trình:

|  |  |  |
| --- | --- | --- |
| **Process** | **Arrival Time** | **Burst Time** |
| *P1* | 0 | 10 |
| *P2* | 4 | 29 |
| *P3* | 8 | 3 |
| *P4* | 38 | 7 |
| *P5* | 40 | 12 |

Sử dụng thuật toán FCFS:

* 1. Vẽ lược đồ Gannt
  + P1 đến lúc 0, chạy từ 0 đến 10 (Burst Time = 10).
  + P2 đến lúc 4, chờ đến khi P1 kết thúc (10), chạy từ 10 đến 39 (10 + 29 = 39).
  + P3 đến lúc 8, chờ đến khi P2 kết thúc (39), chạy từ 39 đến 42 (39 + 3 = 42).
  + P4 đến lúc 38, chờ đến khi P3 kết thúc (42), chạy từ 42 đến 49 (42 + 7 = 49).
  + P5 đến lúc 40, chờ đến khi P4 kết thúc (49), chạy từ 49 đến 61 (49 + 12 = 61).

**Lược đồ Gantt:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P1** | **P2** | **P3** | **P4** | **P5** |

0 10 39 42 49 61

* 1. Tính thời gian chờ trung bình (Average Waiting Time)?

**Waiting Time (WT) = Start Time - Arrival Time**

P1: 0

P2: 10 - 4 = 6

P3: 39 - 8 = 31

P4: 42 - 38 = 4

P5: 49 - 40 = 9

* **Average Waiting Time = = 10**
  1. Tính thời gian hoàn thành trung bình (Average Complete Time)?

**Complete Time (CT) = Finish Time**

P1: 10

P2: 39

P3: 42

P4: 49

P5: 61

* **Average Complete Time = = 40.2**
  1. Tính thời gian quay vòng trung bình (Average Turnaround Time)?

**Turnaround Time (TAT) = Complete Time - Arrival Time**

P1: 10 - 0 = 10

P2: 39 - 4 = 35

P3: 42 - 8 = 34

P4: 49 - 38 = 11

P5: 61 - 40 = 21

* **Average TAT = = 22.2**

1. Thuật toán FCFS có thể dẫn tới tình trạng trì hoãn vô hạn định (starvation) không? Cho ví dụ?

**Không**, thuật toán FCFS không dẫn đến tình trạng trì hoãn vô hạn định (starvation) vì nó xử lý tiến trình theo thứ tự đến của chúng mà không có sự phân biệt ưu tiên. Nếu tiến trình không có thời gian chờ, nó sẽ được xử lý ngay khi đến lượt. Tuy nhiên, một điểm yếu của FCFS là nó có thể gây ra tình trạng **convoy effect**—nghĩa là nếu một tiến trình có thời gian burst dài, các tiến trình khác sẽ phải chờ đợi rất lâu.

Ví dụ:

* Nếu một tiến trình P1 có thời gian burst rất dài (ví dụ, 100 units), thì các tiến trình đến sau P1 sẽ phải chờ rất lâu, dẫn đến thời gian chờ của chúng sẽ rất cao. Tuy nhiên, P1 vẫn sẽ được xử lý mà không có vấn đề nào về việc bị bỏ qua.

1. Cho bảng mô tả các tiến trình, xét các giải thuật **FCFS**, **SFJ**, và **RR** với quantum time = 10

|  |  |
| --- | --- |
| **Process** | **Burst Time** |
| *P1* | 10 |
| *P2* | 28 |
| *P3* | 3 |
| *P4* | 5 |
| *P5* | 12 |

* 1. Giải thuật nào có thời gian chờ trung bình nhỏ nhất?
  2. Giải thuật nào có thời gian hoàn thành trung bình nhỏ nhất?
  3. Giải thuật nào có thời gian đáp ứng trung bình tốt nhất?
* **FCFS:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P1** | **P2** | **P3** | **P4** | **P5** |

0 10 38 41 46 58

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Thời gian chờ** | **Thời gian hoàn thành** | **Thời gian đáp ứng** |
| **P1** | 0 | 10 | 0 |
| **P2** | 10 | 38 | 10 |
| **P3** | 38 | 41 | 38 |
| **P4** | 41 | 46 | 41 |
| **P5** | 46 | 58 | 46 |
| **Trung bình** | **27** | **38.6** | **27** |

* **SFJ:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P3** | **P4** | **P1** | **P5** | **P2** |

0 3 8 18 30 58

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Thời gian chờ** | **Thời gian hoàn thành** | **Thời gian đáp ứng** |
| **P1** | 8 | 18 | 8 |
| **P2** | 30 | 58 | 30 |
| **P3** | 0 | 3 | 0 |
| **P4** | 3 | 8 | 3 |
| **P5** | 5 | 30 | 18 |
| **Trung bình** | **11.8** | **23.4** | **11.8** |

* **RR (q=10):**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **P1** | **P2** | **P3** | **P4** | **P5** | **P2** | **P5** | **P2** |

0 10 20 23 28 38 48 50 58

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Thời gian chờ** | **Thời gian hoàn thành** | **Thời gian đáp ứng** |
| **P1** | 0 | 10 | 0 |
| **P2** | 30 | 58 | 10 |
| **P3** | 20 | 23 | 20 |
| **P4** | 23 | 28 | 23 |
| **P5** | 38 | 50 | 28 |
| **Trung bình** | **22.2** | **33.8** | **16.2** |

**Kết quả:**

* a) **SJF** có thời gian chờ trung bình nhỏ nhất: **11.8**
* b) **SJF** có thời gian hoàn thành trung bình nhỏ nhất: **23.4**
* c) **SJF** có thời gian đáp ứng trung bình tốt nhất: **11.8**

1. Cho bảng mô tả các tiến trình, sử dụng thuật toán **Priority Scheduling**:

|  |  |  |
| --- | --- | --- |
| **Process** | **Burst Time** | **Priority** |
| P1 | 18 | 3 |
| P2 | 5 | 1 |
| P3 | 6 | 5 |
| P4 | 3 | 2 |
| P5 | 25 | 4 |

Vẽ lược đồ Gantt và tính thời gian đợi trung bình?

* Giả sử tất cả tiến trình đến cùng lúc (Arrival Time = 0).
* Thứ tự thực thi: P2 (1) → P4 (2) → P1 (3) → P5 (4) → P3 (5).

**Lược đồ Gantt:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P2** | **P4** | **P1** | **P5** | **P3** |

0 5 8 26 51 57

**Thời gian đợi trung bình (Average Waiting Time)**

* WT = Start Time - Arrival Time
  + P2: 0
  + P4: 5 - 0 = 5
  + P1: 8 - 0 = 8
  + P5: 26 - 0 = 26
  + P3: 51 - 0 = 51
* **Average WT = = 18**

1. Cho bảng mô tả các tiến trình

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Arrival Time** | **Burst Time** | **Priority** |
| P1 | 0 | 10 | 3 |
| P2 | 1 | 3 | 2 |
| P3 | 2 | 4 | 1 |
| P4 | 3 | 1 | 2 |
| P5 | 4 | 5 | 4 |

Vẽ giản đồ Gantt và tính thời gian chờ trung bình và thời gian quay vòng trung bình cho các giải thuật?

* 1. **SFJ** Preemptive (SRTF)
  2. **RR** với quantum time = 2
* **SRTF (không độc quyền):**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **P1** | **P2** | **P3** | **P4** | **P2** | **P1** | **P5** |

0 1 2 6 7 9 18 23

|  |  |  |
| --- | --- | --- |
|  | **Thời gian chờ** | **Thời gian xoay vòng** |
| **P1** | 8 | 18 |
| **P2** | 5 | 8 |
| **P3** | 0 | 6 |
| **P4** | 3 | 4 |
| **P5** | 14 | 19 |

* **Thời gian đợi trung bình**:
* **Thời gian xoay vòng trung bình**:
* **RR(q=2):**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **P1** | **P2** | **P3** | **P1** | **P4** | **P5** | **P2** | **P3** | **P1** | **P5** | **P1** | **P5** | **P1** |

0 2 4 6 8 9 11 12 14 16 18 20 21 23

|  |  |  |
| --- | --- | --- |
|  | **Thời gian chờ** | **Thời gian xoay vòng** |
| **P1** | 13 | 23 |
| **P2** | 8 | 11 |
| **P3** | 8 | 12 |
| **P4** | 5 | 6 |
| **P5** | 12 | 17 |

* **Thời gian đợi trung bình**:
* **Thời gian xoay vòng trung bình**:

1. Cho bảng mô tả các tiến trình:

|  |  |
| --- | --- |
| **Process** | **Burst Time** |
| *P1* | 9 |
| *P2* | 25 |
| *P3* | 4 |
| *P4* | 7 |
| *P5* | 12 |

Sử dụng thuật toán **Round Robin** tính thời gian đợi trung bình và thời gian hoàn thành trung bình:

* 1. Quantum time = 10

##### Lần 1:

* **P1** (9) : Chạy hết 9 đơn vị => Hoàn thành => **Thời gian hoàn thành = 9**
* **P2** (25) : Chạy 10 đơn vị => Còn lại 15 => Đưa vào hàng đợi
* **P3** (4) : Chạy hết 4 đơn vị => Hoàn thành => **Thời gian hoàn thành = 13**
* **P4** (7) : Chạy hết 7 đơn vị => Hoàn thành => **Thời gian hoàn thành = 20**
* **P5** (12) : Chạy 10 đơn vị => Còn lại 2 => Đưa vào hàng đợi

##### **Lần 2:**

* **P2** (còn 15) : Chạy 10 đơn vị => Còn lại 5 => Đưa vào hàng đợi
* **P5** (còn 2) : Chạy hết 2 đơn vị => Hoàn thành => **Thời gian hoàn thành = 32**

##### **Lần 3:**

* **P2** (còn 5) : Chạy hết 5 đơn vị => Hoàn thành => **Thời gian hoàn thành = 37**

#### **=> Thời gian hoàn thành :**

P1 = 9;

P2 = 37;

P3 = 13;

P4 = 20;

P5=32.

WT = TAT− Burst Time (BT)

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Burst Time** | **Turnaround Time** | **Waiting Time** |
| P1 | 9 | 9 | 0 |
| P2 | 25 | 37 | 12 |
| P3 | 4 | 13 | 9 |
| P4 | 7 | 20 | 13 |
| P5 | 12 | 32 | 20 |

* **Thời gian đợi trung bình**:
* **Thời gian hoàn thành trung bình**:
  1. Quantum time = 5

##### Lần 1:

* **P1 (9)** : Chạy **5 đơn vị** => Còn lại **4** => Đưa vào hàng đợi.
* **P2 (25)** : Chạy **5 đơn vị** => Còn lại **20** => Đưa vào hàng đợi.
* **P3 (4)** : Chạy **4 đơn vị** => **Hoàn thành** => **Thời gian hoàn thành = 9**.
* **P4 (7)** : Chạy **5 đơn vị** => Còn lại **2** => Đưa vào hàng đợi.
* **P5 (12)** : Chạy **5 đơn vị** => Còn lại **7** => Đưa vào hàng đợi.

##### Lần 2:

* **P1 (còn 4)** : Chạy **4 đơn vị** => **Hoàn thành** => **Thời gian hoàn thành = 14**.
* **P2 (còn 20)** : Chạy **5 đơn vị** => Còn lại **15** => Đưa vào hàng đợi.
* **P4 (còn 2)** : Chạy **2 đơn vị** => **Hoàn thành** => **Thời gian hoàn thành = 16**.
* **P5 (còn 7)** : Chạy **5 đơn vị** => Còn lại **2** => Đưa vào hàng đợi.

##### Lần 3:

* **P2 (còn 15)** : Chạy **5 đơn vị** => Còn lại **10** => Đưa vào hàng đợi.
* **P5 (còn 2)** : Chạy **2 đơn vị** => **Hoàn thành** => **Thời gian hoàn thành = 27**.

##### Lần 4:

* **P2 (còn 10)** : Chạy **5 đơn vị** => Còn lại **5** => Đưa vào hàng đợi.

##### Lần 5:

* **P2 (còn 5)** : Chạy **5 đơn vị** => **Hoàn thành** => **Thời gian hoàn thành = 40**.

#### **=> Thời gian hoàn thành :**

P1 = 14; P2 = 40; P3 = 9; P4 = 16; P5=27.

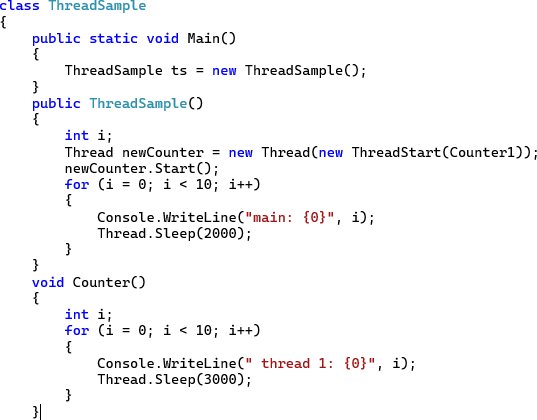
WT = TAT− Burst Time (BT)

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Burst Time** | **Turnaround Time** | **Waiting Time** |
| P1 | 9 | 14 | 5 |
| P2 | 25 | 40 | 15 |
| P3 | 4 | 9 | 5 |
| P4 | 7 | 16 | 9 |
| P5 | 12 | 27 | 15 |

* **Thời gian đợi trung bình**:
* **Thời gian hoàn thành trung bình**:

## Làm việc với Thread bằng C#

* + 1. Thực thi và giải thích ý nghĩa chương trình sau:



Chương trình trên sử dụng tính năng đa luồng (multithreading) trong C# để chạy đồng thời hai luồng (threads) riêng biệt.

### Cấu trúc chương trình:

1. **Khởi tạo đối tượng ThreadSample:**
   * Trong phương thức Main, một đối tượng của lớp ThreadSample được khởi tạo. Khi đối tượng này được tạo, hàm khởi tạo (constructor) ThreadSample() được gọi.
2. **Phương thức ThreadSample():**
   * Đây là hàm khởi tạo của lớp ThreadSample. Khi nó được gọi, một luồng mới (thread) có tên newCounter được tạo ra và bắt đầu thực thi phương thức Counter.
   * Sau khi tạo luồng mới, hàm khởi tạo tiếp tục thực thi một vòng lặp từ 0 đến 9, mỗi lần lặp lại in ra thông báo "main: {i}", sau đó chờ 2 giây .
3. **Phương thức Counter():**
   * Đây là phương thức sẽ được luồng newCounter thực thi. Trong phương thức này, một vòng lặp từ 0 đến 9 sẽ in ra thông báo "thread 1: {i}", sau đó chờ 3 giây

### Ý nghĩa và hoạt động:

* Chương trình sử dụng hai luồng:
  + **Luồng chính (main thread):** Chạy phương thức Main, sau đó gọi hàm khởi tạo của ThreadSample, thực thi vòng lặp for in "main: {i}" và ngủ 2 giây mỗi lần.
  + **Luồng phụ (newCounter thread):** Chạy phương thức Counter, thực thi vòng lặp for in "thread 1: {i}" và ngủ 3 giây mỗi lần.

### Quá trình thực thi:

1. Khi đối tượng ThreadSample được tạo trong phương thức Main, hàm khởi tạo (ThreadSample()) sẽ khởi chạy luồng phụ (newCounter) và tiếp tục thực thi vòng lặp in "main: {i}".
2. Luồng chính và luồng phụ sẽ chạy đồng thời:
   * Luồng chính (main thread) sẽ in ra "main: {0}", "main: {1}", ..., "main: {9}" mỗi 2 giây.
   * Luồng phụ (newCounter) sẽ in ra "thread 1: {0}", "thread 1: {1}", ..., "thread 1: {9}" mỗi 3 giây.

### Lưu ý về đồng thời (Concurrency):

* Cả hai luồng (main thread và newCounter thread) sẽ chạy độc lập với nhau, nên kết quả in ra có thể không theo thứ tự cụ thể mà bạn dự đoán (do sự đồng thời).
* Do luồng chính ngủ 2 giây và luồng phụ ngủ 3 giây, bạn sẽ thấy thông báo của hai luồng xen kẽ nhau.

Ví dụ kết quả có thể là:

main: 0

thread 1: 0

main: 1

main: 2

thread 1: 1

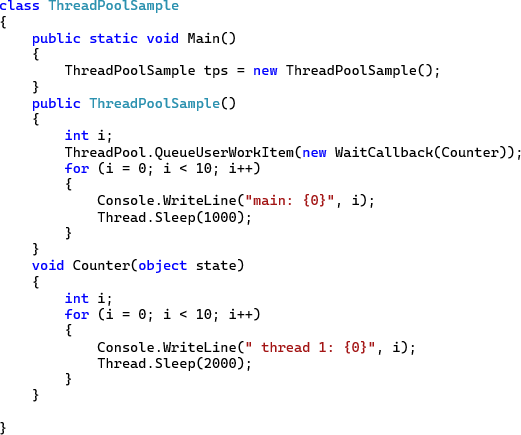
main: 3

thread 1: 2

...

### Kết luận: Chương trình minh họa cách sử dụng đa luồng trong C# để chạy các tác vụ đồng thời.

* + 1. Tìm hiểu về TheardPool. Thực thi và giải thích ý nghĩa chương trình sau:



Chương trình này cũng sử dụng tính năng đa luồng trong C#, nhưng thay vì tạo một luồng mới trực tiếp, nó sử dụng **ThreadPool** để quản lý và thực thi các tác vụ trong một nhóm các luồng (threads) đã được tạo sẵn. Chúng ta cùng phân tích chi tiết ý nghĩa của từng phần trong chương trình:

### Cấu trúc và hoạt động:

1. **Khởi tạo đối tượng** ThreadPoolSample**:**
   * Trong phương thức Main, một đối tượng của lớp ThreadPoolSample được tạo. Khi đối tượng này được tạo, hàm khởi tạo (constructor) ThreadPoolSample() được gọi.
2. **Phương thức** ThreadPoolSample()**:**
   * Trong hàm khởi tạo, một tác vụ (task) được đưa vào **ThreadPool** bằng cách sử dụng phương thức ThreadPool.QueueUserWorkItem.
   * QueueUserWorkItem nhận vào một WaitCallback (đại diện cho phương thức sẽ được thực thi bởi một luồng trong ThreadPool). Ở đây, phương thức Counter sẽ được gọi từ ThreadPool.
   * Sau đó, hàm khởi tạo tiếp tục thực thi một vòng lặp từ 0 đến 9, mỗi lần in ra thông báo "main: {i}", và sau đó ngủ 1 giây (Thread.Sleep(1000)).
3. **Phương thức** Counter(object state)**:**
   * Đây là phương thức sẽ được thực thi bởi một luồng trong **ThreadPool**. Trong phương thức này, một vòng lặp từ 0 đến 9 sẽ in ra thông báo "thread 1: {i}", và mỗi lần nó sẽ ngủ 2 giây (Thread.Sleep(2000)).

### Ý nghĩa và hoạt động:

* **ThreadPool** là một nhóm các luồng sẵn có, được sử dụng để xử lý các tác vụ mà không cần phải tạo một luồng mới mỗi khi cần chạy một tác vụ. Điều này giúp tiết kiệm tài nguyên hệ thống và tối ưu hóa hiệu suất khi có nhiều tác vụ cần thực thi đồng thời.
* Khi gọi ThreadPool.QueueUserWorkItem, bạn chỉ cần cung cấp một phương thức (ở đây là Counter), và ThreadPool sẽ lấy một luồng từ nhóm của nó để thực thi phương thức đó. Điều này làm cho việc quản lý luồng trở nên dễ dàng hơn so với việc tự tạo từng luồng như trong chương trình sử dụng Thread trước đó.

### Quá trình thực thi:

1. Khi đối tượng ThreadPoolSample được tạo, phương thức ThreadPoolSample() gọi ThreadPool.QueueUserWorkItem để đưa tác vụ Counter vào ThreadPool.
2. **ThreadPool** sẽ lấy một luồng từ nhóm của nó và thực thi phương thức Counter. Sau đó, luồng này sẽ chạy vòng lặp in "thread 1: {i}" và ngủ 2 giây sau mỗi lần lặp.
3. Đồng thời, vòng lặp trong phương thức khởi tạo (trong luồng chính) cũng sẽ tiếp tục chạy, in ra "main: {i}" và ngủ 1 giây sau mỗi lần lặp.

### Lưu ý về đồng thời (Concurrency):

* Cả luồng chính và luồng trong ThreadPool sẽ chạy đồng thời. Tuy nhiên, vì luồng trong ThreadPool ngủ 2 giây trong khi luồng chính ngủ 1 giây, bạn sẽ thấy thông báo của hai luồng này xen kẽ nhau.
* Các luồng trong ThreadPool không được tạo thủ công, và các luồng trong ThreadPool có thể được tái sử dụng cho các tác vụ khác sau khi hoàn thành công việc hiện tại.

### Ví dụ kết quả có thể là:

main: 0

thread 1: 0

main: 1

main: 2

thread 1: 1

main: 3

main: 4

...

### Kết luận:

Chương trình này minh họa cách sử dụng **ThreadPool** trong C# để thực thi các tác vụ đồng thời mà không cần phải tạo mới từng luồng. Điều này giúp tiết kiệm tài nguyên hệ thống và tối ưu hóa hiệu suất khi có nhiều tác vụ cần thực thi đồng thời.