Module 4: Thread Safety and Concurrent File Processing

(Tutorial / Lab)

Subjects: Systems Programming

Minor: For all bachelor programs

Question 1 – Race Conditions and Thread Safety

Write a C# program that increments a shared counter from multiple threads.

Tasks:

1. Create a shared integer variable.
2. Increment the variable from at least **5 parallel tasks**.
3. Observe and record the incorrect result.
4. Fix the problem using:
    - lock
    - Interlocked.Increment

Discussion:

- Why does the race condition occur?
- Compare lock and Interlocked in terms of performance and use cases.

Trả lời: link Question_01

Kết quả sau 1 lần chạy:

Method	**Counter result**	**Time(ms)**
No sync	288496	134
Lock	500000	33
Interlocked	500000	7

Lý do race condition xảy ra:

- *counter*++ không phải atomic gồm đọc, cộng, ghi
- Nhiều thread đọc cùng giá trị cũ nên ghi đè kết quả của nhau
- Kết quả cuối thấp hơn giá trị đúng, không ổn định giữa các lần chạy

So sánh lock và Interlocked:

- lock: an toàn cho logic phức tạp, chậm hơn do synchronization
- Interlocked: atomic ở mức CPU, nhanh, chỉ dùng cho phép toán đơn giản
- Counter, statistics dùng Interlocked; critical section lớn dùng lock

Question 2 – Task Coordination and Synchronization

Develop a program where multiple tasks must complete before the program continues.

Tasks:

1. Create 3 independent tasks that simulate work (e.g., sleep for random time).
2. Use **one** of the following mechanisms to wait for completion:
    - Task.WhenAll
    - CountdownEvent
    - ManualResetEventSlim
3. Print a message only after all tasks are finished.

Discussion:

- Which synchronization mechanism is easiest to use?
- When would low-level synchronization be preferred over Task.WhenAll?

Trả lời: link Question_02

Task.WhenAll là cơ chế dễ dùng nhất. Ngắn gọn, rõ nghĩa, ít lỗi. Không cần quản lý counter hay signal thủ công. Phù hợp hầu hết bài toán async.

Cần đồng bộ mức thấp khi muốn phối hợp giữa thread và code không async. Cần kiểm soát chính xác signal / trạng thái. Tích hợp với API cũ, callback-based hoặc hệ thống realtime.

Question 3 – Thread-Safe File Access

Multiple threads need to write logs to the same file.

Tasks:

1. Create a program where multiple tasks write text to the same log file.
2. Run the program **without synchronization** and observe the output.
3. Make the file writing thread-safe using:
    - lock
    - OR a dedicated logging task with a queue

Discussion:

- What problems occur without synchronization?
- Why is file I/O considered a shared system resource?

Trả lời: link Question_03

Vấn đề khi không đồng bộ: Nhiều thread cùng ghi file gây ghi chồng dữ liệu, mất dòng log hoặc file bị corrupt. File system không đảm bảo atomic cho nhiều thao tác ghi đồng thời.

Vì sao file I/O là shared system resource: Do File được quản lý ở mức OS, không thuộc riêng thread nào. Mọi thao tác ghi đều truy cập cùng một handle và buffer hệ thống, nên bắt buộc phải đồng bộ để tránh xung đột.

Question 4 – File Monitoring and Concurrent Processing

Create a program that monitors a directory and processes files concurrently.

Tasks:

1. Use FileSystemWatcher to detect new files in a folder.
2. When a new file appears:
    - Read its contents
    - Compress it
    - Save the compressed version
3. Ensure thread safety if multiple files arrive at the same time.

Discussion:

- What concurrency issues can arise with file system events?
- How can event storms be handled safely?

Trả lời: link Question_04

Concurrency issues với FileSystemWatcher: Một file có thể phát nhiều event (*Created, Changed*) cho cùng một file. File có thể bị bắt sự kiện khi còn đang được ghi, dẫn tới lỗi đọc file hoặc file chưa hoàn chỉnh. Khi nhiều file xuất hiện cùng lúc, callback có thể chạy song song gây quá tải I/O.

Cách xử lý event storm an toàn: Giới hạn số tác vụ xử lý đồng thời bằng *SemaphoreSlim* để tránh bắn quá nhiều task cùng lúc. Trì hoãn ngắn (*Task.Delay*) trước khi đọc file để đảm bảo OS ghi xong. Đẩy xử lý nặng sang background task thay vì chạy trực tiếp trong event handler. Do FileSystemWatcher chỉ nên dùng để phát hiện, không dùng để xử lý nặng. Việc tách event và xử lý giúp chương trình ổn định khi nhiều file đến đồng thời và tránh crash trong lúc demo.

Question 5 – Secure and Efficient File Storage

Design a small utility that safely stores sensitive data.

Tasks:

1. Read text data from a file.
2. Encrypt the data using a symmetric encryption algorithm.
3. Compress the encrypted data.
4. Save the final output to disk.
5. Reverse the process (decompress + decrypt).

Discussion:

- Why is encryption usually done before compression?
- What are the performance trade-offs of encryption and compression?

Trả lời: link Question_05

Dữ liệu được mã hóa trước khi nén nhằm đảm bảo tính bảo mật ngay từ đầu quá trình xử lý. Khi dữ liệu đã được mã hóa, nội dung gốc không còn ở dạng rõ ràng, giúp giảm rủi ro rò rỉ thông tin nếu dữ liệu trung gian bị truy cập trái phép. Tuy nhiên, dữ liệu sau khi mã hóa có độ entropy rất cao, gần như ngẫu nhiên, nên khả năng nén giảm mạnh.

Đánh đổi hiệu năng giữa mã hóa và nén:

- **Mã hóa:** Tiêu tốn CPU để thực hiện các phép toán mật mã. Không làm giảm kích thước dữ liệu. Là bước bắt buộc để đảm bảo bảo mật.
- **Nén:** Tốn CPU để phân tích và loại bỏ dữ liệu dư thừa. Giảm dung lượng lưu trữ và I/O. Hiệu quả phụ thuộc vào tính cấu trúc của dữ liệu.
- **Tổng quát:** Dùng cả hai bước làm tăng thời gian xử lý và mức sử dụng CPU. Đổi lại là dữ liệu an toàn hơn và tiết kiệm không gian lưu trữ. Trong hệ thống hiệu năng cao, có thể phải chọn chỉ nén hoặc chỉ mã hóa tùy yêu cầu.