Module 3: Memory Management and Threading

(Tutorial / Lab)

Subjects: Systems Programming

Minor: For all bachelor programs

Lab Objective

Students will:

- Use low-level .NET APIs to interact with memory and data structures
- Understand the performance implications of value vs reference types
- Apply memory and speed optimization strategies
- Explore CLR internals with practical exercises

Question 1 – Stack vs Heap and Object Lifetime

Write a C# program that:

- Declares both value types (`int`, `struct`) and reference types (`class`)
- Allocates objects inside and outside a method scope

Tasks:

- Identify which variables are stored on the stack and which are stored on the heap.
- Explain when each object becomes eligible for garbage collection.
- Use comments to describe object lifetime and scope.

Trả lời: Link Question 1

Question 2 – Garbage Collection and Memory Pressure

Create a console application that:

- Allocates a large number of objects inside a loop
- Forces garbage collection using `GC.Collect()` (for observation only)

Tasks:

- Observe memory usage before and after garbage collection.
- Explain why forcing garbage collection is discouraged in production code.
- Describe the role of GC generations (Gen 0, Gen 1, Gen 2) in this scenario.

Trả lời: Link Question 2

Dựa trên kết quả thực thi chương trình:

- **Memory Before (1,126,176 bytes):** Đây là mức tiêu thụ bộ nhớ khởi điểm của ứng dụng (bao gồm CLR overhead và các biến môi trường).
- **Memory Allocated (2,899,928 bytes):** Bộ nhớ tăng lên khoảng 1.77 MB.
- **Memory After GC (408696 bytes):** Sau khi ép buộc *GC.Collect()*, bộ nhớ giảm sâu xuống còn khoảng 0.4 MB.

Không nên ép GC trong Production vì:

- **Gây trễ nghiêm trọng:** *GC.Collect()* làm dừng toàn bộ ứng dụng (stop-the-world), dễ gây lag, đặc biệt với web và realtime system.
- **Phá cơ chế tự tối ưu của GC:** GC .NET vốn tự điều chỉnh theo hành vi cấp phát; can thiệp thủ công làm nó hoạt động kém hiệu quả về lâu dài.
- **Tăng chi phí dọn rác:** Gọi sai thời điểm có thể đẩy object lên Gen cao hơn, khiến GC tốn tài nguyên hơn nhiều.

GC .NET chia bộ nhớ thành 3 thế hệ để tối ưu hiệu suất:

- **Gen 0:** Chứa đối tượng ngắn hạn. Trong thí nghiệm, các obj được tạo rồi mất tham chiếu ngay, nên được dọn rất nhanh ở Gen 0.
- **Gen 1 & Gen 2:** Dành cho đối tượng sống lâu. Nếu không ép GC, phần lớn object sẽ bị xóa ngay ở Gen 0 và không bị đẩy lên các thế hệ cao hơn.

Question 3 – Writing Memory-Efficient Code

Given a program that heavily uses `List<T>`, LINQ, and object allocations:

Tasks:

- Refactor the code to reduce memory allocations (e.g., use arrays, `Span<T>`, or object pooling where appropriate).
- Identify at least **three memory-saving techniques** applied.
- Explain the trade-offs between readability, performance, and memory usage.

Trả lời: Link Question 3

Các kỹ thuật tiết kiệm bộ nhớ đã dùng:

- Loại bỏ LINQ và iterator: Tránh tạo delegate, iterator và List trung gian, giúp giảm allocation từ 1992 bytes xuống còn 256 bytes.
- Sử dụng mảng và vòng lặp for: Xử lý trực tiếp dữ liệu, không sinh collection phụ trong quá trình xử lý.
- Sử dụng ReadOnlySpan<char>: Truy cập trực tiếp vùng nhớ của chuỗi gốc, tránh Split và Substring nên giảm allocation xuống còn 208 bytes.

Đánh đổi giữa readability, performance, và memory usage:

Method	**Time (ms)**	**Allocated (bytes)**
LINQ + List	3	1992
Array + for loop	0	256
ReadOnlySpan	7	208

- **Dễ đọc:** LINQ có code ngắn gọn và rõ ý định nhất. Array + for dài hơn do phải quản lý vòng lặp và điều kiện. Span khó đọc nhất vì cần xử lý index, slice và kiến thức về bộ nhớ.
- **Hiệu năng:** Kết quả đo cho thấy array + for loop nhanh nhất với 0 ms, trong khi LINQ mất 3 ms do overhead iterator. Span chậm hơn với 7 ms vì chi phí xử lý chỉ số và thao tác Slice trong dữ liệu nhỏ.
- **Bộ nhớ:** LINQ tiêu tốn nhiều bộ nhớ nhất (1992 bytes). Array giảm mạnh allocation (256 bytes). Span dùng ít bộ nhớ nhất (208 bytes), gần như không tạo object trên heap.

Question 4 – Multithreading and the Thread Pool

Write a program that processes 100 work items using:

- `Thread`
- `ThreadPool`
- `Task`

Tasks:

- Print the managed thread ID for each execution.
- Compare performance and resource usage.
- Explain why the thread pool and tasks are generally preferred over manually creating threads.

Trả lời: Link Question 4

Method	**Time(ms)**	**Unique Threads**
Thread	567	100
ThreadPool	154	13
Task	125	13

Kết quả đo cho thấy:

- **Thread** có thời gian thực thi cao nhất *(567 ms)* và sử dụng tới *100 thread* khác nhau, tương ứng với việc mỗi work item tạo một thread riêng. Việc tạo nhiều thread gây tốn bộ nhớ stack, chi phí khởi tạo cao và làm tăng context switching của hệ điều hành.
- **ThreadPool** có thời gian thực thi giảm mạnh xuống còn *154 ms* trong khi chỉ sử dụng *13 thread*. Điều này cho thấy các work item được phân phối lên một số lượng thread giới hạn và được tái sử dụng thay vì tạo mới liên tục, giúp tiết kiệm tài nguyên hệ thống.
- **Task** thời gian chạy tiếp tục giảm xuống còn *125 ms* và số thread sử dụng vẫn giữ nguyên ở *13 thread*, cho thấy Task thực chất chạy trên ThreadPool. Tuy nhiên, nhờ cơ chế Task Scheduler, việc phân phối công việc hiệu quả hơn nên đạt hiệu năng tốt nhất trong ba phương pháp.

Lý do ThreadPool và Task được ưu tiên

Việc tạo thread thủ công gây tiêu tốn nhiều tài nguyên do mỗi thread cần bộ nhớ stack riêng và chi phí tạo lớn. Với 100 work item, chương trình phải quản lý *100 thread* khác nhau, dẫn đến hiệu năng thấp và khó mở rộng.

ThreadPool khắc phục hạn chế này bằng cách giới hạn số thread hoạt động đồng thời và tái sử dụng chúng cho nhiều tác vụ khác nhau. Nhờ đó, cùng một khối lượng công việc nhưng chỉ cần *13 thread*, giúp giảm chi phí hệ thống và cải thiện hiệu năng rõ rệt.

Task được ưu tiên hơn vì cung cấp mức trừu tượng cao hơn ThreadPool, cho phép runtime tự động lập lịch, cân bằng tải và tối ưu song song. Kết quả đo cho thấy Task đạt thời gian tốt nhất (*125 ms*) mà không cần tăng số thread sử dụng, nên đây là cách tiếp cận hiệu quả và an toàn nhất trong lập trình đa luồng .NET.

Question 5 – Async/Await and Common Pitfalls

Create an asynchronous method that simulates a long-running operation using `Task.Delay()`.

Tasks:

- Call the method using `await` and explain how it improves CPU utilization.
- Demonstrate why `async void` should be avoided.
- Explain the problems caused by using `Task.Result` or `Task.Wait()`.

Trả lời: Link Question 5

Khi gọi phương thức bất đồng bộ bằng *await*, thread đang chạy không bị chặn trong thời gian chờ *Task.Delay()*. Thay vì giữ *CPU* để đợi hoàn thành, thread được trả về ThreadPool để thực hiện các công việc khác. Khi tác vụ hoàn tất, phần còn lại của phương thức sẽ được tiếp tục thực thi trên một thread khác nếu cần. Nhờ cơ chế này, CPU không bị idle một cách lãng phí và số lượng thread cần thiết để xử lý nhiều tác vụ đồng thời được giảm đáng kể, đặc biệt hiệu quả trong các ứng dụng I/O-bound.

Lý do nên tránh dùng async void

Phương thức *async void* không trả về *Task*, vì vậy caller không thể *await*, không thể bắt *exception* và không thể biết khi nào phương thức hoàn thành. Trong ví dụ trên, exception được ném ra trong *AsyncVoidMethod* nhưng không thể bị bắt bởi khối try–catch bên ngoài.

Điều này khiến lỗi runtime khó kiểm soát, dễ làm crash ứng dụng và không thể quản lý luồng thực thi.

Vấn đề khi sử dụng Task.Result hoặc Task.Wait()