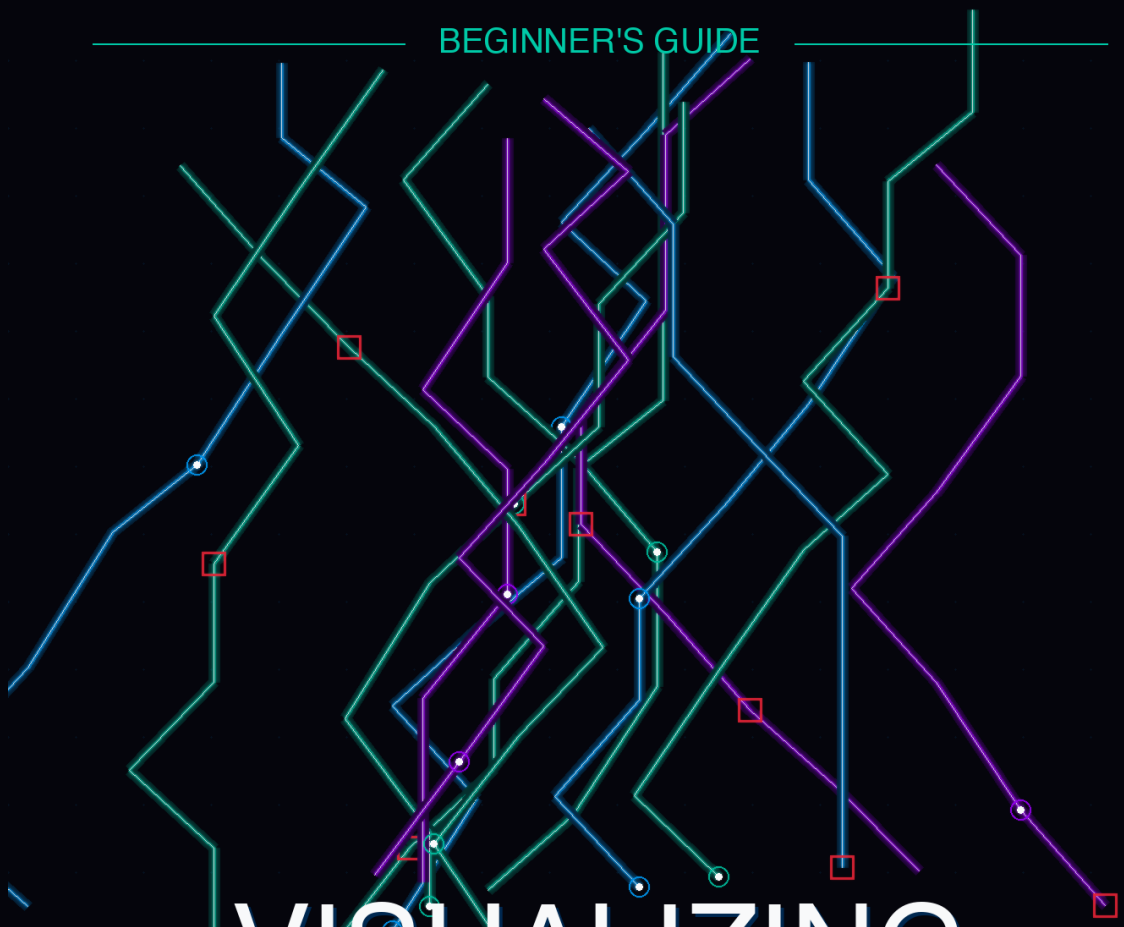
BEGINNER'S GUIDE



VISUALIZING C# CONCURRENCY

A Research-Driven Guide

> “Jangan hanya menulis async/await — pahami apa yang sebenarnya terjadi di memori saat thread berjalan.”

First Edition

Afandy Lamusu

Senior Backend Engineer & Software Architect

.NET 9 | C# 13

Daftar Isi

BEGINNER'S GUIDE

Visualizing C# Concurrency

A Research-Driven Guide

> *"Jangan hanya menulis async/await — pahami apa yang sebenarnya terjadi di memori saat thread berjalan."*

```
new Thread() ──▶ OS thread ──▶ ~1 MB stack
   └─ ThreadPool ──▶ reuse · hill-climb · work-steal

async/await ──▶ state machine ──▶ continuation
   └─ SynchronizationContext · ConfigureAwait

lock ── Monitor ── SpinLock ── Mutex
   └─ Channel<T> · SemaphoreSlim · RWLockSlim

Race · Deadlock · Starvation ──▶ detect & fix
```

First Edition

Afandy Lamusu

SENIOR BACKEND ENGINEER & SOFTWARE ARCHITECT

.NET 9 C# 13

Halaman Hak Cipta

Visualizing C# Concurrency: A Research-Driven Guide

Edisi Pertama, Februari 2026

Copyright & Disclaimer

AI-Generated Content Disclosure

Karya ini dikembangkan dengan bantuan teknologi kecerdasan buatan (termasuk Claude oleh Anthropic). Penulis, **Afandy Lamusu**, bertindak sebagai arsitek utama, dan validator teknis untuk seluruh konten. Semua konsep teknis, logika kode, dan pengaturan struktural diarahkan serta disempurnakan oleh penulis untuk memastikan akurasi dan ekspresi orisinal.

Technical Disclaimer

Informasi dalam buku ini dijual “apa adanya” dan tanpa jaminan. Meskipun setiap tindakan pencegahan telah diambil dalam penyusunan karya ini, baik penulis maupun penerbit tidak akan bertanggung jawab kepada orang atau entitas mana pun sehubungan dengan kerugian atau kerusakan yang disebabkan atau diduga disebabkan secara langsung atau tidak langsung oleh instruksi yang terkandung dalam buku ini atau oleh produk perangkat lunak dan perangkat keras yang dijelaskan di dalamnya.

Kode Sumber Pendamping

Seluruh kode C# yang muncul dalam buku ini tersedia di repositori GitHub dan dirilis di bawah **MIT License**.

Teknologi Referensi

- .NET 9
- C# 13
- BenchmarkDotNet 0.14.0
- ASP.NET Core 9.0

Kontak Penulis

- **Afandy Lamusu**
- LinkedIn: linkedin.com/in/afandylamusu
- Email: afandy.lamusu@gmail.com
- Website: [Link Website Anda]

Diterbitkan secara digital · Indonesia · 2026

Kata Pengantar

"*Concurrency is hard.*"

Kita semua pernah mendengar kalimat itu. Namun, di era *cloud computing* dan prosesor *multi-core* saat ini, menulis kode yang berjalan secara sekuensial bukan lagi sebuah pilihan—itu adalah batasan. C# telah memberikan kita keajaiban berupa kata kunci `async` dan `await` yang membuat kode asinkron terlihat semudah kode sinkron. Namun, justru di situlah letak bahayanya.

Banyak pengembang terjebak dalam pola "menghafal sintaksis" tanpa memahami apa yang sebenarnya terjadi di balik layar. Mereka tahu cara menggunakan `Task.Run`, tapi mungkin tidak menyadari beban *context switching* yang ditimbulkannya. Mereka menggunakan `lock`, tapi tidak bisa memvisualisasikan bagaimana sebuah *deadlock* perlahan-lahan mencekik aplikasi di lingkungan produksi.

Buku ini lahir dari kegelisahan saya sebagai seorang arsitek perangkat lunak. Saya sering melihat sistem yang gagal bukan karena logika bisnis yang salah, melainkan karena kebocoran *thread*, *race conditions* yang sulit direproduksi, atau *thread starvation* yang membuat sistem melambat tanpa alasan yang jelas.

Mengapa kita harus peduli pada mekanisme internal?

Karena abstraksi selalu bocor (*abstractions leak*). Saat aplikasi Anda melambat di bawah beban berat, `async` dan `await` tidak akan memberi tahu Anda mengapa *thread pool* mencapai batasnya. Hanya pemahaman mendalam tentang bagaimana .NET mengelola memori, bagaimana *state machine* bekerja, dan bagaimana CPU menjadwalkan tugas yang dapat menyelamatkan Anda.

Dalam buku ini, saya mengajak Anda untuk berhenti sejenak dari sekadar mengetik kode dan mulai **memvisualisasikan** aliran data. Kita akan menggunakan pendekatan *Research-Driven*—tidak ada klaim yang tidak kita buktikan dengan *benchmark* atau *profiling*. Kita akan melihat langsung bagaimana instruksi kita memengaruhi memori dan CPU.

Tujuan saya sederhana: Saya ingin Anda memiliki "mata arsitek". Saat Anda menulis satu baris kode concurrency, saya ingin Anda bisa melihat benang-benang *thread* yang saling bersilangan, titik-titik sinkronisasi yang terbentuk, dan potensi hambatan yang mungkin muncul sebelum kode itu mencapai *server*.

Concurrency memang sulit, tapi ia tidak harus menjadi misteri. Mari kita mulai perjalanan ini dengan satu komitmen: **Jangan hanya menulis kode—pahami apa yang sebenarnya terjadi.**

Afandy Lamusu
Senior Backend Engineer & Software Architect

Chapter 1: Apa Itu Thread, Sebenarnya?

Subtitle: *Dari Prosesor Fisik ke Managed Thread di .NET*

📋 *Case Study: Startup yang Tidak Bisa Scale*

Sebuah startup fintech membangun payment processing service dalam C#. Di awal, saat load rendah, semua berjalan lancar. Saat pengguna bertambah, tim menambah fitur — termasuk kode yang membuat `new Thread()` *baru untuk setiap transaksi yang masuk.*

Tiga bulan kemudian: di peak hours, server mereka kehabisan memory dan crash. Investigasi menunjukkan ratusan thread hidup bersamaan — setiap thread mengonsumsi ~1MB stack. Dengan 500 transaksi/detik, itu setara dengan 500MB memory hanya untuk stack thread!

Solusinya sederhana setelah mereka memahami ThreadPool: tidak perlu buat thread baru untuk setiap request. ThreadPool sudah ada untuk mengelola ini dengan efisien. ***Memahami biaya nyata thread creation*** *adalah langkah pertama menuju aplikasi yang scalable.*

Part I — Fondasi ini membangun mental model yang mencegah kesalahan seperti di atas.

Pendahuluan

Sebelum kita menulis satu baris kode concurrency, kita perlu membangun **mental model** yang benar tentang thread. Banyak developer pemula menulis `Task.Run()` atau `async/await` tanpa benar-benar memahami apa yang terjadi di balik layar. Hasilnya? Bug yang sulit direproduksi, aplikasi yang tiba-tiba hang, atau performa yang justru lebih buruk dari kode single-thread.

Chapter ini akan membangun fondasi itu. Kita mulai dari analogi sederhana, lalu turun ke level hardware, dan akhirnya naik lagi ke managed thread di .NET. Di akhir chapter, kamu bisa menjawab pertanyaan: "*Apa sebenarnya yang terjadi saat kamu menulis* `new Thread()`?"

1.1 Analogi Dunia Nyata: Dapur Restoran

Bayangkan sebuah restoran dengan dapur yang sibuk.

- **Restoran** = Program (proses) kamu
- **Dapur** = Memori program (RAM)
- **Kompor** = CPU core
- **Koki** = Thread

Saat restoran hanya punya **satu koki**, dia harus mengerjakan semua pesanan secara berurutan: masak sup, lalu masak steak, lalu masak dessert. Satu kompor, satu koki — sederhana, tapi lambat saat pesanan menumpuk.

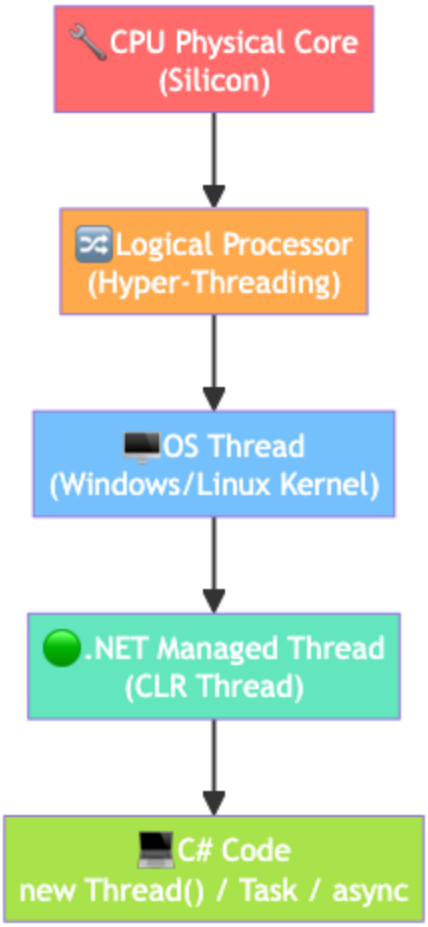
Saat restoran menambah **koki kedua**, keduanya bisa memasak di atas kompor yang berbeda secara bersamaan. Throughput naik. Tapi sekarang muncul masalah baru: bagaimana jika dua koki butuh bumbu yang sama pada saat yang sama? Siapa yang duluan? Apa yang terjadi kalau mereka saling menunggu sambil masing-masing memegang bahan yang dibutuhkan koki lainnya?

Itulah concurrency: **kekuatan sekaligus bahaya**.

> 💡 ***Insight Kunci:*** *Thread adalah unit eksekusi independen. Seperti koki, setiap thread punya "tangan"-nya sendiri (register, stack), tapi berbagi "dapur" yang sama (memori proses).*

1.2 Dari Hardware ke Software: Hierarki Thread

Untuk benar-benar memahami thread, kita perlu menelusuri jalur dari hardware ke kode C# kamu.



Level 1: CPU Physical Core

CPU modern memiliki beberapa **physical core** — unit komputasi independen yang bisa mengeksekusi instruksi secara benar-benar paralel. Laptop kamu mungkin punya 4, 8, atau 16 physical core.

Setiap physical core memiliki: - **ALU** (Arithmetic Logic Unit) — untuk komputasi - **Register file** — penyimpanan data sementara super cepat (nanoseconds) - **L1/L2 Cache** — memori lokal per-core (microseconds) - **Branch predictor** — untuk memprediksi alur eksekusi

Level 2: Logical Processor (Hyper-Threading)

Intel memperkenalkan **Hyper-Threading** (AMD: SMT — Simultaneous Multi-Threading). Satu physical core bisa menjalankan **dua thread secara semu-paralel** dengan cara berbagi unit eksekusi.

Ketika satu thread sedang menunggu data dari cache/memori, unit eksekusi bisa dipakai thread lain. Hasilnya: satu physical core terlihat seperti **dua logical processor** di operating system.

```
CPU dengan 4 Physical Cores + Hyper-Threading:
├── Core 0 → Logical CPU 0, Logical CPU 1
├── Core 1 → Logical CPU 2, Logical CPU 3
├── Core 2 → Logical CPU 4, Logical CPU 5
└── Core 3 → Logical CPU 6, Logical CPU 7

Total: 8 logical processors
```

Level 3: OS Thread

OS Thread adalah abstraksi yang dibuat oleh kernel operating system. OS yang bertanggung jawab untuk: - Mengalokasikan thread ke logical processor yang tersedia - Menjadwalkan kapan setiap thread mendapat giliran CPU (**scheduling**) - Menyimpan dan memulihkan state thread saat **context switch**

Setiap OS thread memiliki: - **Thread ID** — identifikasi unik - **Stack** — ~1MB di Windows (default), menyimpan local variables dan call frame - **Thread Control Block (TCB)** — metadata thread di kernel - **Register state** — snapshot semua register saat thread di-pause

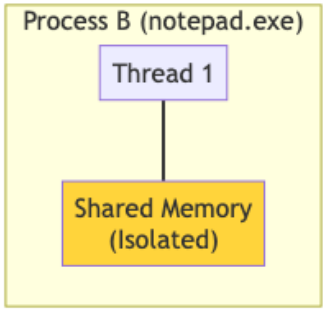
Level 4: .NET Managed Thread

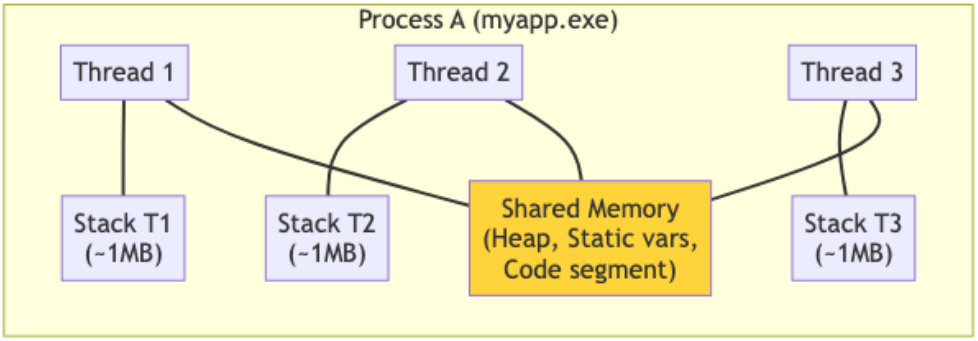
CLR (Common Language Runtime) membungkus OS thread dalam **managed thread**. CLR menambahkan: - **Thread-local storage** — data yang unik per thread - **Exception handling state** - **GC interaction** — thread harus bisa di-suspend untuk garbage collection (safepoints) - **AppDomain affinity** (untuk .NET Framework)

Di .NET, kamu berinteraksi dengan managed thread melalui class `System.Threading.Thread`.

1.3 Thread vs Process

Perbedaan fundamental antara **process** dan **thread** adalah **kepemilikan memori**.





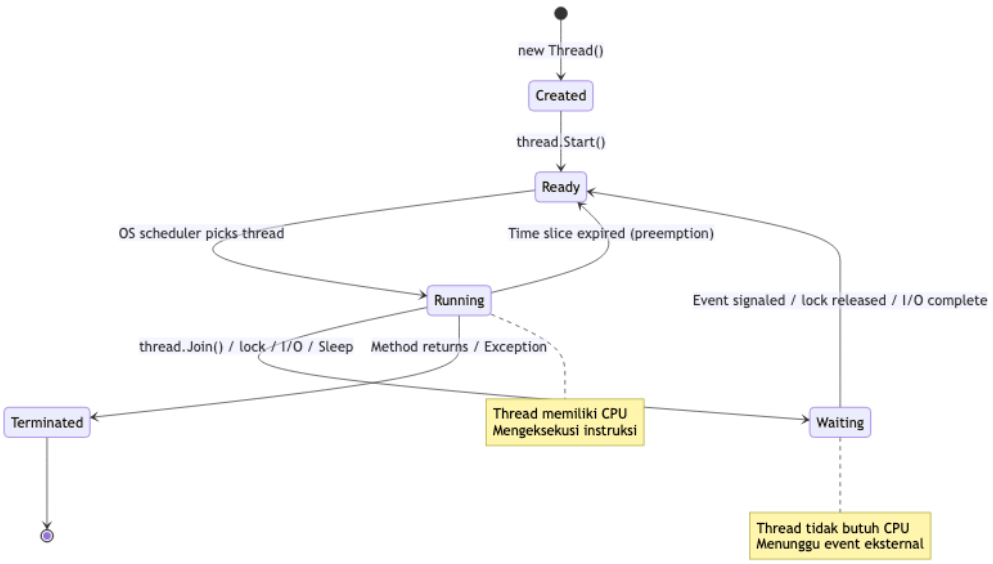
Process: - Memiliki address space sendiri yang **terisolasi** - Komunikasi antar process butuh IPC (Inter-Process Communication): pipes, sockets, shared memory - Crash di satu process tidak crash process lain

Thread: - Berbagi address space dengan semua thread dalam process yang sama - Komunikasi antar thread: **langsung** — share variable (inilah sumber masalah concurrency!) - Crash di satu thread (unhandled exception) bisa crash seluruh process

> ⚠️ ***Implikasi Penting:*** *Karena thread berbagi memori, dua thread bisa* ***membaca dan menulis variabel yang sama secara bersamaan****. Ini sangat efisien, tapi juga sangat berbahaya jika tidak dikelola dengan benar.*

1.4 Thread Lifecycle: State Machine

Setiap thread mengalami siklus hidup yang didefinisikan dengan baik. Memahami siklus ini krusial untuk debugging.



Penjelasan Setiap State

State	Deskripsi	Contoh
Created	Thread dibuat tapi belum dimulai	`new Thread(method)`
Ready	Siap dieksekusi, menunggu giliran CPU	Setelah `Start()`, atau setelah I/O selesai
Running	Sedang mengeksekusi di CPU	Thread sedang di-assign ke logical processor
Waiting	Diblokir, tidak butuh CPU	`Thread.Sleep()`, menunggu `lock`, I/O
Terminated	Selesai eksekusi	Method thread selesai atau throw exception

Di .NET: `ThreadState` Enum

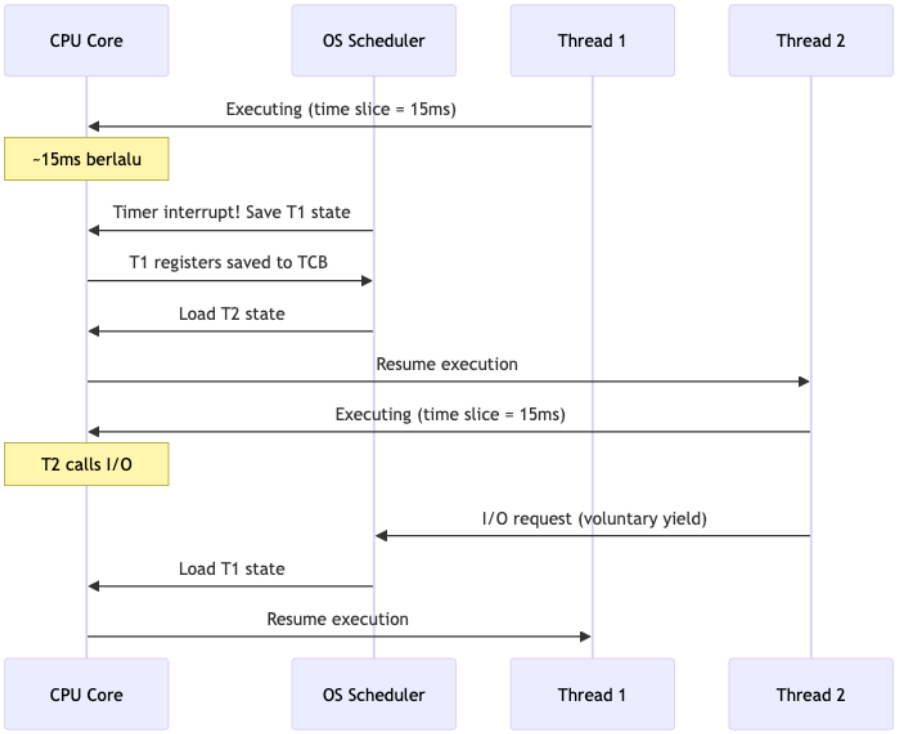
.NET menyediakan `System.Threading.ThreadState` untuk memeriksa state thread:

```
var thread = new Thread(() =>
{
    Thread.Sleep(2000);
});

Console.WriteLine(thread.ThreadState); // Unstarted

thread.Start();
Thread.Sleep(100); // Beri waktu thread untuk start
Console.WriteLine(thread.ThreadState); // WaitSleepJoin (karena Sleep)

thread.Join();
Console.WriteLine(thread.ThreadState); // Stopped
```

> 📌 ***Catatan:*** *Di production,* `ThreadState` *lebih berguna untuk debugging daripada logic control — nilainya bisa berubah sewaktu-waktu secara concurrent.*

1.5 Context Switching: Apa yang Sebenarnya Terjadi?

Context switching adalah proses OS memindahkan CPU dari satu thread ke thread lain. Ini terjadi ribuan kali per detik dan merupakan salah satu "biaya tersembunyi" dari multi-threading.



Apa yang Disimpan Saat Context Switch?

Setiap kali OS menghentikan sebuah thread, ia harus menyimpan **seluruh state** thread tersebut ke dalam **Thread Control Block (TCB)**:

```
TCB (Thread Control Block):
├── General Purpose Registers (RAX, RBX, RCX, RDX, ...)  ← ~120 bytes
├── Stack Pointer (RSP)
├── Instruction Pointer (RIP) — di mana thread berhenti
├── Flags Register (RFLAGS)
├── Segment Registers
├── FPU/SSE State (floating point, SIMD) ← bisa ratusan bytes
└── Thread metadata (priority, affinity, ...)
```

Ketika thread dilanjutkan, semua nilai ini **di-restore** ke register CPU. Proses ini memakan waktu **1–10 microseconds** tergantung hardware.

Biaya Tambahan: TLB Flush

Jika OS memindahkan thread ke CPU core yang berbeda, **TLB (Translation Lookaside Buffer)** — cache yang memetakan virtual address ke physical address — harus di-flush. Ini menyebabkan **page table walk** yang lebih lambat untuk beberapa waktu setelah perpindahan.

Voluntary vs Involuntary Context Switch

Tipe	Penyebab	Dampak
Voluntary	Thread sendiri yield (I/O, Sleep, lock wait)	Lebih efisien, thread sudah tidak butuh CPU
Involuntary	Time slice habis, thread dipaksa berhenti	Kurang efisien, thread mungkin di tengah-tengah kerja

```
# Di Linux, cek context switch statistics:
vmstat 1

# Output:
# procs -----------memory---------- ---swap-- -----io---- -system-- ------
        cpu-----
# r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy
        id wa st
# 2  0      0 1234567  12345 678901    0    0     0     1  234 5678  5  2
        93  0  0
#                                                          ^--- cs =
        context switches/sec
```

1.6 Biaya Nyata Membuat Thread

Thread bukan resource yang "gratis". Setiap thread yang kamu buat membawa overhead nyata:

Memory Overhead

```
Per Thread Memory Usage:
├── Stack (default 1MB di Windows, 512KB di Linux 64-bit)
│   └── Untuk local variables, method call frames
├── Thread Environment Block (TEB) ≈ 4KB
├── Kernel Thread Object ≈ beberapa KB
└── CLR Thread Object + Thread-Local Storage

Total: ~1MB+ per thread
```

Implikasi: Jika kamu membuat 1000 thread, itu berarti ~1GB RAM hanya untuk stack thread! Ini sebelum program kamu melakukan apapun.

Waktu Creation

Membuat thread tidak instant. Prosesnya melibatkan: 1. Alokasi memory untuk stack 2. Inisialisasi kernel thread object 3. System call ke OS 4. CLR initialization untuk managed thread 5. Scheduling oleh OS scheduler

Proses ini bisa memakan waktu **ratusan microseconds** hingga beberapa milliseconds.

Benchmark Awal: Thread Creation Cost

```
// Ilustrasi perbedaan waktu — bukan benchmark formal
// Lihat Lab 1 untuk angka yang akurat

var sw = Stopwatch.StartNew();

// Membuat dan menunggu 100 thread selesai
var threads = new Thread[100];
for (int i = 0; i < 100; i++)
{
    threads[i] = new Thread(() => Thread.Sleep(10));
    threads[i].Start();
}
foreach (var t in threads) t.Join();

sw.Stop();
Console.WriteLine($"100 threads: {sw.ElapsedMilliseconds}ms");
// Pada mesin tipikal: ratusan ms karena overhead creation
```

🔬 Lab 1: Mengukur Thread Creation

Tujuan: Membuktikan secara empiris biaya membuat thread dibandingkan alternatifnya.

Setup Project

```
dotnet new console -n Chapter01.ThreadBasics
cd Chapter01.ThreadBasics
dotnet add package BenchmarkDotNet
```

Kode Benchmark

Buat file `ThreadCreationBenchmark.cs`:

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using BenchmarkDotNet.Order;

[MemoryDiagnoser]          // Tampilkan alokasi memori
[Orderer(SummaryOrderPolicy.FastestToSlowest)]
[RankColumn]
public class ThreadCreationBenchmark
{
    private const int N = 100; // Jumlah unit kerja

    // ── Skenario 1: Thread manual ──────────────────────
    [Benchmark(Description = "new Thread()")]
    public void CreateRawThreads()
    {
        var threads = new Thread[N];
        for (int i = 0; i < N; i++)
        {
            threads[i] = new Thread(DoWork);
            threads[i].Start();
        }
        foreach (var t in threads)
            t.Join();
    }

    // ── Skenario 2: ThreadPool ─────────────────────────
    [Benchmark(Description = "ThreadPool.QueueUserWorkItem")]
    public void UseThreadPool()
    {
        using var countdown = new CountdownEvent(N);
        for (int i = 0; i < N; i++)
        {
            ThreadPool.QueueUserWorkItem(_ =>
```

```
            {
                DoWork();
                countdown.Signal();
            });
        }
        countdown.Wait();
    }

    // ── Skenario 3: Task (TPL) ──────────────────────────────────

    [Benchmark(Description = "Task.Run", Baseline = true)]
    public void UseTasks()
    {
        var tasks = new Task[N];
        for (int i = 0; i < N; i++)
            tasks[i] = Task.Run(DoWork);
        Task.WaitAll(tasks);
    }

    // ── Skenario 4: Task dengan ValueTask ──────────────────────

    [Benchmark(Description = "Parallel.For")]
    public void UseParallelFor()
    {
        Parallel.For(0, N, _ => DoWork());
    }

    // Simulasi pekerjaan minimal — kita ukur overhead creation, bukan
        kerja
    private static void DoWork()
    {
        // Pekerjaan sangat minimal untuk isolasi overhead creation
        Thread.SpinWait(100);
    }
}

// ── Program Entry Point ─────────────────────────────────────────

class Program
{
```

```
    static void Main(string[] args)
    {
        var summary = BenchmarkRunner.Run<ThreadCreationBenchmark>();
    }
}
```

Eksperimen Tambahan: 1000 Thread vs 1000 Task

Tambahkan file `ThreadScalingExperiment.cs`:

```
using BenchmarkDotNet.Attributes;

[MemoryDiagnoser]
public class ThreadScalingBenchmark
{
    [Params(10, 100, 500, 1000)]
    public int ThreadCount { get; set; }

    [Benchmark(Description = "Raw Threads")]
    public void ScaleWithRawThreads()
    {
        var threads = new Thread[ThreadCount];
        for (int i = 0; i < ThreadCount; i++)
        {
            threads[i] = new Thread(() => Thread.SpinWait(1000));
            threads[i].Start();
        }
        foreach (var t in threads) t.Join();
    }

    [Benchmark(Description = "Tasks", Baseline = true)]
    public void ScaleWithTasks()
    {
        var tasks = new Task[ThreadCount];
        for (int i = 0; i < ThreadCount; i++)
```

```
            tasks[i] = Task.Run(() => Thread.SpinWait(1000));
        Task.WaitAll(tasks);
    }
}
```

Cara Menjalankan

```
# Build dalam Release mode (WAJIB untuk benchmark yang akurat)
dotnet run --configuration Release
```

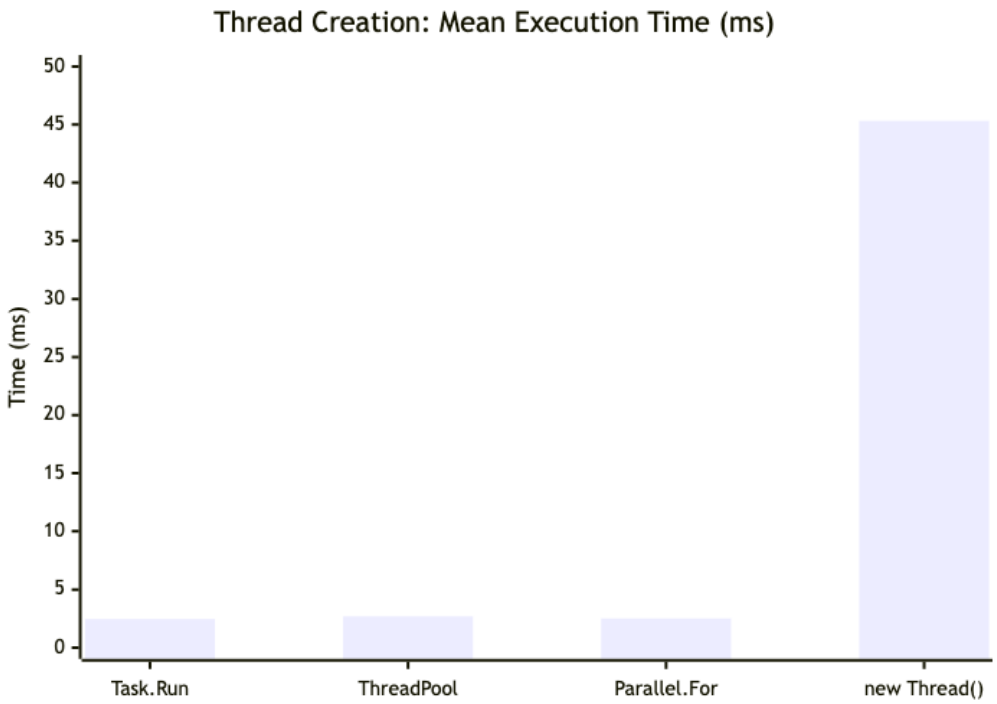
> ⚠️ ***Penting:*** *Jangan jalankan benchmark dalam Debug mode. Debug build memiliki overhead yang signifikan dan hasil tidak akan representatif.*

Hasil yang Diharapkan

Tabel hasil benchmark kamu akan terlihat seperti ini (angka bervariasi per mesin):

Method	Mean	Error	StdDev	Ratio	Allocated
Task.Run (baseline)	2.450 ms	0.048 ms	0.045 ms	1.00	42.3 KB
ThreadPool.QueueUserWorkItem	2.680 ms	0.052 ms	0.049 ms	1.09	38.1 KB
Parallel.For	2.510 ms	0.049 ms	0.046 ms	1.02	15.2 KB
new Thread()	45.320 ms	0.891 ms	0.834 ms	18.50	102.8 MB

Analisis Hasil



Temuan utama dari Lab 1:

1. `new Thread()` **adalah yang paling lambat** — ~18x lebih lambat dari `Task.Run` untuk 100 unit kerja. Setiap panggilan membuat OS thread baru dari nol.

2. `Task.Run` **dan** `ThreadPool` **hampir sama** — keduanya menggunakan ThreadPool di bawahnya. Task hanya menambahkan abstraksi tipis di atasnya.

3. **Alokasi memory berbeda drastis** — Raw thread mengalokasikan ratusan MB (stack per thread), sementara Task menggunakan thread yang sudah ada di pool.

4. **Skalabilitas**: Coba dengan `ThreadCount = 1000`. Kamu akan melihat bahwa 1000 raw thread bisa menghabiskan >1GB RAM dan menyebabkan sistem menjadi sangat lambat, sementara 1000 Task dieksekusi efisien melalui ThreadPool.

💡 Key Outcome

Setelah chapter ini, kamu seharusnya bisa:

- **Menggambar** diagram hierarki dari CPU core → logical processor → OS thread → .NET managed thread
- **Menjelaskan** perbedaan antara process dan thread dalam konteks shared memory
- **Melukiskan** thread lifecycle state machine dan menyebutkan setiap transisi
- **Mendeskripsikan** apa yang terjadi di dalam CPU saat context switch (register save/restore)
- **Mengukur** dan membandingkan biaya membuat thread vs menggunakan ThreadPool menggunakan BenchmarkDotNet
- **Memahami** mengapa `new Thread()` untuk setiap request adalah anti-pattern

---

🔥 Common Mistakes

Mistake 1: `new Thread()` untuk Setiap Request

```
// ❌ JANGAN LAKUKAN INI
public async Task HandleRequest(HttpRequest request)
{
    var thread = new Thread(() => ProcessRequest(request));
    thread.Start();
    // Thread baru dibuat untuk setiap request!
    // Bayangkan 1000 concurrent request → 1000 thread → ~1GB RAM stack
}

// ✅ LAKUKAN INI
public async Task HandleRequest(HttpRequest request)
{
    await Task.Run(() => ProcessRequest(request));
    // ThreadPool mengelola thread secara efisien
}
```

Mengapa ini masalah? Setiap `new Thread()` mengalokasikan ~1MB stack dan membutuhkan system call ke OS. Di bawah load tinggi, ini menyebabkan memory exhaustion dan performa yang sangat buruk.

Mistake 2: Mengabaikan Overhead Context Switching

```
// ❌ Membuat lebih banyak thread dari core yang tersedia tidak selalu
        lebih cepat
int coreCount = Environment.ProcessorCount; // misal 8
var tasks = new Task[coreCount * 100]; // 800 tasks untuk CPU-bound work

for (int i = 0; i < tasks.Length; i++)
    tasks[i] = Task.Run(() => HeavyCpuWork());

// Untuk CPU-bound work, lebih dari Environment.ProcessorCount thread
```

```
// justru memperlambat karena context switching overhead

// ✅ Untuk CPU-bound work, batasi paralelisme:
var options = new ParallelOptions { MaxDegreeOfParallelism = Environ-
        ment.ProcessorCount };
Parallel.For(0, 800, options, i => HeavyCpuWork());
```

Mengapa ini masalah? Untuk pekerjaan CPU-intensive, memiliki lebih banyak thread dari CPU core menyebabkan context switching yang tidak perlu. Thread saling berebut CPU alih-alih bekerja.

Mistake 3: Mengira Thread = Parallelism Sempurna

```
// ❌ Asumsi yang salah
// "Saya punya 4 thread, jadi pekerjaan saya 4x lebih cepat"
// Kenyataannya:
// 1. Context switching overhead
// 2. Memory access contention
// 3. Cache invalidation
// 4. Synchronization overhead
// Seringkali hanya 2-3x lebih cepat, atau bahkan lebih lambat

// ✅ Selalu ukur dengan BenchmarkDotNet sebelum mengklaim optimasi
```

Hukum Amdahl: Hanya bagian kode yang bisa di-parallelize yang mendapat manfaat dari multi-threading. Bagian sekuensial membatasi speedup maksimum yang mungkin.

Ringkasan Chapter 1

```
Thread adalah unit eksekusi independen yang:
├── Dibuat oleh OS (OS thread) dan dibungkus CLR (managed thread)
├── Berbagi address space (heap) dengan thread lain dalam process
├── Memiliki stack pribadi (~1MB) untuk local variables
├── Mengalami siklus: Created → Ready → Running → Waiting → Terminated
└── Memiliki biaya nyata: ~1MB memory, ratusan µs creation time

Context Switch terjadi ketika:
├── Time slice thread habis (involuntary preemption)
└── Thread menunggu I/O, lock, atau Sleep (voluntary yield)
    └── Biaya: 1–10 µs per switch + TLB flush jika pindah core

Rekomendasi:
├── Gunakan ThreadPool/Task daripada new Thread()
├── Batasi thread count untuk CPU-bound work (≈ ProcessorCount)
└── Selalu ukur — jangan assume threading selalu lebih cepat
```

Referensi

- Microsoft Docs: Managed Threading Basics
- .NET Threading Model
- Windows Internals, Part 1 — Mark Russinovich — Chapter 4: Threads
- BenchmarkDotNet Documentation
- CLR via C# — Jeffrey Richter — Chapter 26: Thread Basics

Selanjutnya: Chapter 2 — ThreadPool dan Task: Pekerja yang Lebih Pintar