Programowanie asynchroniczne i równoległe

Bartosz Miąskowski

Wątki

Wątek (*thread*) to **najmniejsza jednostka wykonywania programu**. Każdy działający program składa się z przynajmniej **jednego wątku głównego**, który **wykonuje instrukcje po kolei** (*synchronicznie*).

Pomyśl, że wątek jest pracownikiem, który:

- dostaje zadania do wykonania,
- wykonuje je jedno po drugim,
- nigdy nie robi dwóch rzeczy na raz.

Jeżeli dasz mu zadanie, które trwa długo – np. pobieranie pliku – **pracownik będzie czekał, w tym czasie nie zrobi nic innego.**

Jeżeli pracownikiem jest wątek UI – aplikacja się zawiesi.

Wątek UI

Bez względu na framework (*Windows Forms*, *WPF*, *MAUI*) **UI zawsze działa na jednym wątku**.

Wątek UI obsługuje:

- kliknięcia,
- renderowanie okien,
- animacje,
- odświeżanie interfejsu,
- sterowanie kontrolkami,
- bindowanie danych.

Jeżeli go **zablokujesz**, aplikacja nie może reagować – okno przestaje odpowiadać.

Wątki

Co się dzieje?

- Program wypisuje „1”.
- Wątek czeka 3 sekundy (*jest zajęty*).
- Dopiero potem wypisuje „2”.

Przez 3 sekundy nie może zrobić niczego innego.

```
Console.WriteLine("1");
Thread.Sleep(3000); // wątek stoi 3 sekundy
Console.WriteLine("2");
```

Kiedy aplikacja się zawiesza?

Kiedy aplikacja się zawiesza, użytkownik widzi, że:

- UI nie reaguje.
- Przyciski są zamrożone.

Technicznie jest to spowodowane tym, że **wątek UI jest zajęty zbyt długo**, i **nie może obsługiwać zdarzeń**.
Domyślnie wszystkie operacje wykonywane są w tym samym wątku, w którym działa interfejs graficzny.

Operacje, które potrafią trwać bardzo długo

- Pobieranie plików.
- Zapytania do bazy danych.
- Żądania do API.
- Operacje na plikach.

I jeszcze długo można by wymieniać…

Powyższe operacje, nie powodują zwiększonego zapotrzebowania na zasoby procesora – **wymuszają konieczność czekania**. Jeżeli wykonamy je **synchronicznie**, **zablokujemy cały wątek podczas oczekiwania na wynik**.

Zadania obliczeniowe v I/O

Zadania I/O bound

(ograniczeniem jest czekanie)

Operacje I/O wymagają komunikacji z zewnętrznym źródłem:

- Pobieranie pliku z Internetu.
- Zapytania HTTP.
- Zapytania do bazy danych.
- Operacje na dysku.
- Czekanie – `Task.Delay()`

Przy tego typu operacjach procesor się nudzi, on jedynie czeka na odpowiedź.

Zadania CPU bound

(ograniczeniem jest procesor)

Operacje, które intensywnie korzystają z mocy obliczeniowej procesora:

- Skomplikowane obliczenia (*całki*, *macierze*).
- Analiza dużych zbiorów danych.
- Kompresja, szyfrowanie.
- Renderowanie grafiki.
- Parsowanie dużych plików.

Tutaj procesor na nic nie czeka, przez cały czas pracuje – często bardzo ciężko.

Czekanie synchroniczne

W synchronicznym podejściu wątek:

- Zaczyna operację
- **Czeka** na jej zakończenie
- Dopiero potem robi coś dalej

W czasie „*czekania*”:

- wątek UI jest **zajęty**,
- interfejs nie działa,
- aplikacja wygląda na zawieszoną.

```
Thread.Sleep(3000); // Wątek stoi i niczego nie może robić
```

Czekanie asynchroniczne

W asynchronicznym podejściu:

- Zaczynasz operację (*np. pobieranie danych*)
- **Oddajesz wątek** – niech UI działa normalnie
- Gdy wynik jest gotowy – wykonujesz dalszy kod
- Nie ma blokady – UI żyje.

```
await Task.Delay(3000); // Wątek jest wolny, UI działa
```

Czekanie synchroniczne i asynchroniczne

Synchronicznie

Idziesz po kebaba, stoisz 10 minut w kolejce i patrzysz, jak robią.

- przez ten czas **nie możesz nic zrobić**
- nic innego się nie dzieje (*blokada*)

Asynchronicznie

Zamawiasz kebaba, dostajesz elektronicznego pikacza i siadasz przy stoliku.

- **Nie czekasz aktywnie** – możesz robić inne rzeczy.
- Gdy jest gotowe – **pikacz pika** (*kończysz czekanie*)

Asynchroniczność

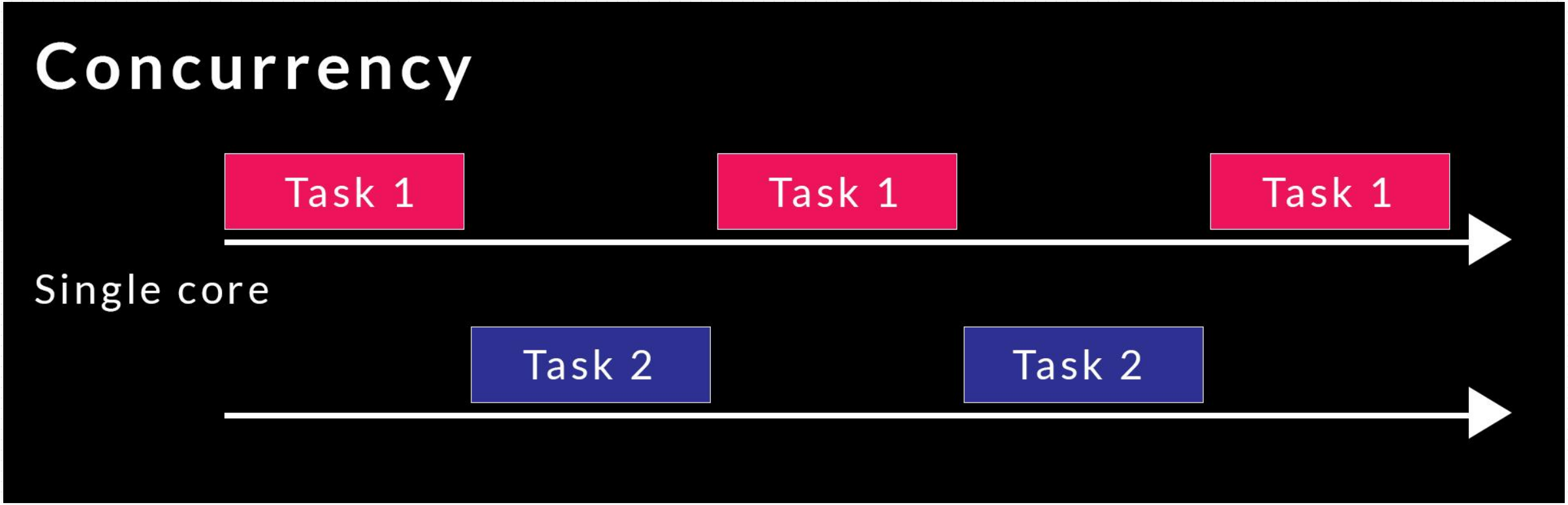
Wyobraź sobie, że jesteś kasjerem w Żabce. W kolejce stoją dwie osoby: **Ania** i **Jan**.

Ania jest pierwsza i chce kupić **hot-doga**. Przygotowanie hot-doga trwa jednak dość długo – trzeba poczekać, aż podgrzeje się bułka i parówka. W tym czasie kasjer **nie musi nic robić**, bo proces podgrzewania dzieje się sam.

Drugą osobą w kolejce jest Jan, który chce kupić **butelkę wody**. Ponieważ kasjer i tak tylko czeka na podgrzanie składników hot-doga, może w tym czasie obsłużyć Jana i sprzedać mu wodę.

Gdy bułka i parówka będą gotowe, kasjer wraca do Ani i sprzedaje jej gotowego hot-doga.

Asynchroniczność



Słowa kluczowe **async** i **await**

async

- Umieszczamy przy **sygnaturze metody**
- Oznacza: *ta metoda może zawierać await*
- **Nie** sprawia, że metoda działa w tle
- **Nie** tworzy nowego wątku

async pozwala metodzie na zatrzymanie się i kontynuację później.

```
private async Task TestAsync()
{
    Console.WriteLine("Start");
    await Task.Delay(2000); // asynchroniczne czekanie
    Console.WriteLine("Koniec");
}
```

await

- Używamy przed operacją, która **zwraca Task**
- Mówi: *poczekaj na zakończenie tej operacji, ale NIE blokuj wątku*
- To tutaj dzieje się „*magia*”:
  - metoda się zatrzymuje,
  - wątek jest oddany,
  - po zakończeniu operacji wykonywany jest dalszy kod.

async – metoda może zrobić przerwę w pracy
await – tu zrób przerwę i wróć później

Blokowanie wątku v czekanie asynchroniczne

`Thread.Sleep()` – blokowanie wątku

Co to robi?

- Zatrzymuje **wątek**, w którym zostało wywołane.
- Wątek **nie może** w tym czasie wykonać niczego innego.
- Jeśli to wątek UI – aplikacja się **zamraża**
- jeśli to wątek serwera – request się **blokuje**

To jest **czekanie aktywne** – wątek stoi i nic nie robi (*kosztuje czas i zasoby*).

Efekty uboczne:

- zamrożenie UI
- brak reakcji aplikacji
- brak możliwości anulowania
- brak możliwości przeplotu innych zadań

```
Thread.Sleep(3000);
```

`Task.Delay()` – czekanie asynchroniczne

Co to robi?

- metoda robi przerwę
- **ale wątek zostaje zwolniony!**
- UI działa normalnie
- serwer może obsługiwać inne requesty
- po 3 sekundach wykonywany jest dalszy kod (*kontynuacja*)

To jest **czekanie nieblokujące**.

Efekty:

- brak zamrożenia UI
- brak blokady wątku
- inne zadania mogą być wykonywane

```
await Task.Delay(3000);
```

Metody asynchroniczne - `Task`

`Task` to **obiekt reprezentujący operację, która jeszcze się nie zakończyła**.

- To nie jest wątek.
- To nie jest kod działający w tle.
- To opis trwającej operacji.

`Task` mówi nam „*Hej, zacząłem robić tę rzecz. Dam Ci znać, kiedy skończę*”.

Metoda z przykładu to po prostu **obietnica, że coś się wykona**. Zwraca ona `Task`, ponieważ:

- Wykonanie może trwać.
- Trzeba mieć możliwość oczekiwania bez blokowania (`await`)
- Dzięki obiektowi `Task`, **`await` wie dokładnie kiedy operacja zostanie wykonana**.

```
async Task SaveAsync()
{
    await Task.Delay(1000); // np. zapis pliku
}
```

Metody asynchroniczne – `Task<T>`

Jeżeli chcemy, aby metoda asynchroniczna zwróciła wartość używamy `Task<T>`. Jest to **obietnica, że kiedyś dostaniemy wynik typu T**.

```
async Task<string> GetDataAsync()
{
    await Task.Delay(500);
    return "gotowe";
}
```

Metody asynchroniczne – `ValueTask<T>`

`ValueTask<T>` to bardziej zaawansowany typ do scenariuszy, kiedy:

- Wyniki są czasem dostępne **natychmiast**.
- Czasem wymagają asynchronicznego czekania.
- Najczęstszy przykład: **cache**.

Różnica między `Task<T>` a `ValueTask<T>`:

- **`Task<T>`** - zawsze tworzy obiekt `Task`.
- **`ValueTask<T>`** - może nie tworzyć obiektu (*ze względu na wydajność*). Jest strukturą.

Generalnie prawie zawsze używamy `Task<T>`.

```
ValueTask<string> GetCachedAsync()
{
    if (_cache.TryGetValue("key", out var value))
        return new ValueTask<string>(value); // wynik gotowy od ręki

    return new ValueTask<string>(LoadFromDiskAsync());
}
```

Pułapki związane z `ValueTask<T>`

- **Nigdy nie używaj** `await` **więcej niż raz** na tej samej instancji `ValueTask`. Obiekt ten może zostać zresetowany po pierwszym odczycie wyniku.
- **Nie używaj współbieżnego dostępu.** Nie możesz bezpiecznie odczytywać tego samego `ValueTask` z wielu wątków jednocześnie.
- **Utrudniona kompozycja.** Nie możesz łatwo użyć `ValueTask` w `Task.WhenAll` czy `Task.WhenAny`. Wymaga to konwersji `.AsTask()`, co z kolei alokuje nowy obiekt `Task`, negując zysk wydajnościowy.

Dlaczego **`async`** musi zwrócić **`Task`**?

Cała koncepcja async i await opiera się na tym, że:

- **`await` potrzebuje czegoś, na co może „*czekać*” bez blokowania**.
- `Task` mówi: „*wywołaj kontynuację, gdy skończę*”.
- `Task` ma stany:
  - `Running`
  - `Completed`
  - `Faulted`
  - `Canceled`

Wątek nie jest blokowany, zamiast tego czekamy na stan `Task`.

Kryminał – `async void`

W większości przypadków **`async void`** to **najgorsze możliwe** rozwiązanie w programowaniu asynchronicznym.

- Nie zachowuje się jak zwykła `async` metoda.
- Nie da się go oczekiwać (*używać `await`*).
- Nie da się złapać wyjątków.

Niestety, istnieją scenariusze w których istnieje konieczność skorzystania z `async void`.

Brak oczekiwania – `async void`

W przypadku metod `async void`, słowo kluczowe **`await` po prostu nie działa**...

Oznacza to, że:

- Nie możesz poczekać, aż operacja się zakończy.
- Nie możesz napisać poprawnego kodu, który zależy od wykonania danej metody.
- Nie możesz zapewnić kolejności działań (*operacji*).

```
async void DoWorkAsync() { ... }

await DoWorkAsync(); // BŁĄD – nie da się
```

Brak wyjątków – `async void`

Każdy wyjątek z `async void`:

- trafia poza Twój kod,
- rozbija się o synchronizację,
- **może zamknąć aplikację**.

```
async void BadAsync()
{
    await Task.Delay(1000);
    throw new Exception("BUM!");
}
```

Do przykładu:

- Jeżeli wywołanie `async void` jest w `try`, **nic się nie wydarzy** (*nie zostanie przechwycony*).
- Wyjątek z `async void` poleci globalnie – zginie gdzieś, wśród innych wyjątków globalnych.
- Może ubić całą aplikację bądź pojedynczy request (*jeżeli ASP.NET Core*)

Dlatego też się mówi, że async void jest toksyczne. ☠️

Dlaczego **`async void`** istnieje?

Istnieje **jeden przypadek**, w którym dozwolone jest użycie `async void` – **`EventHandler`**, który **musi mieć sygnaturę `void`**. W przeciwnym wypadku, nie dałoby się używać `await` w zdarzeniach UI.

```
private async void Button_Click(object sender, EventArgs e)
{
    await Task.Delay(1000);
}
```

```
async void OnClick(...)
async void OnLoaded(...)
async void OnTextChanged(...)
```

Dozwolone jest użycie `async void` w EventHandlerach.

Windows Forms, WPF, MAUI

Czego nigdy nie robić z **`async void`**?

- Nie tworzyć metod pomocniczych `async void`.
- Nie robić logiki biznesowej `async void`.
- Nie wywoływać `async void` tylko dla „fire and forget”.
- **Nigdy nie używać `async void` do operacji ciężkich obliczeniowo.**

Używamy `async Task` zawsze, gdzie jest to możliwe *(gdy potrzebujemy asynchroniczności)*.

Używamy `async void` tylko wtedy, gdy piszemy event handler – **nigdzie indziej**.

Odrzucanie Tasków

Odrzucenie taska oznacza, że uruchamiamy coś asynchronicznie, ale nie interesuje nas, kiedy to się skończy ani co się stanie (*fire and forget*). Zwykle jest **to bardzo zły pomysł**.

- `Task` odrzucony (bez `await`) **nie zwróci wyjątku**, może ubić aplikację podobnie jak z `async void`.
- **Nie wiemy czy operacja w ogóle się powiodła** – czy trwa, czy się skończyła.
- Tworzymy chaos – mogą wykonywać się, tworzyć w **nieznanej kolejności**.

```
public void SendEmail()
{
    SendEmailAsync(); // BŁĄD – task jest odrzucony!
}
```

Źle ❌

```
try
{
    await SendEmailAsync();
}

catch (Exception ex)
{
    // obsługa błędu
}
```

Dobrze ✅

Używanie **`await`**

Najprostszy przypadek – **bezpośrednie użycie `await`** oraz otrzymanie wyniku.

- `Task` jest uruchamiany od razu.
- Od razu czekamy na wynik.
- Wątek jest od razu zwalniany (*na czas czekania*).
- Jest to najczęściej stosowana forma.

```
var res = await DoAsync();
```

Używanie `await`

Możliwe jest uruchomienie taska wcześniej, a dopiero później odczekanie na wynik.

Dzięki temu:

- Zadanie rozpoczyna się wykonywać od razu.
- W tym czasie, można robić coś innego.
- Używasz `await` dopiero, kiedy wynik jest potrzebny.

```
var t1 = DoAsync();   // 1. Uruchamiamy zadanie
// ... kod wykonywany w międzyczasie ...
var res1 = await t1;  // 2. Czekamy na zakończenie
```

```
DoSomethingElse(await t1);
```

W tym przypadku, zadanie **nie jest** wykonywane drugi raz:

1. Kończy się wykonywanie taska pod `t1`.
2. Wynik trafia jako argument do metody.

Jeżeli `Task` się wykonał przed użyciem `await t1`, **wynik jest natychmiastowy – na nic nie musimy czekać.**

Kontekst synchronizacji

Kiedy używamy `await` w aplikacji z UI (*WinForms*, *WPF*, *MAUI*), dzieje się coś bardzo ważnego:

- Kod **po `await`** domyślnie wraca na **ten sam kontekst**, z którego został wywołany. W aplikacjach UI to jest **wątek UI**.
- Za to zachowanie odpowiada obiekt o nazwie **`SynchronizationContext`**.

`SynchronizationContext` działa nieco inaczej w zależności od typu aplikacji:

- **Aplikacje UI** – pilnuje, aby wszystko co dotyczy UI, było wykonywane na wątku UI.
- **ASP.NET Core** – brak UI, kod po `await` może wrócić do dowolnego wątku z puli (`ThreadScheduler`).

Kontekst synchronizacji

W aplikacjach z UI (*WinForms*, *WPF*, *MAUI*):

- **Kontrolki nie są thread-safe**.
- Nie możesz ich **modyfikować z innego wątku niż wątek UI**.
- Gdyby `await` wznowił metodę w innym wątku **zaczęłyby lecieć wyjątki** typu „*Cross-thread operation not valid*”.

```
private async void Button_Click(object sender, EventArgs e)
{
    //1. Jesteśmy w wątku UI (korzystamy z niego)
    label1.Text = "Ładowanie...";

    //2. Zwalniamy wątek UI (może robić coś innego)
    await Task.Delay(2000);

    //3. Wracamy automatycznie do wątku UI (dzięki SynchronizationContext)
    label1.Text = "Gotowe!";
}
```

`Wait()` i `.Result`

Dzięki użyciu `await` **wątek nie jest blokowany**, inaczej to wygląda, kiedy chcemy **wywołać metodę asynchroniczną z metody synchronicznej** (*bez async w sygnaturze*) – wtedy czasami możemy użyć `.Wait()` (jeżeli nie oczekujemy wyniku) i `.Reslut`, które blokują wątek na sztywno. **Prowadzi to często do powstania deadlocka**, czyli **zakleszczenia wątków, którego nie da się rozwiązać**.

Poniższy przykład powoduje deadlocka w aplikacjach z UI:

```
private void button1_Click(object sender, EventArgs e)
{
    var res = GetHelloAsync().Result;
    MessageBox.Show(res);
}

private async Task<string> GetHelloAsync()
{
    await Task.Delay(1000);
    return "Hello, World!";
}
```

`ConfigureAwait(false)`

Nie zawsze możemy chcieć wrócić do wątku UI, wtedy z pomocą przychodzi `.ConfigureAwait(false)`, które sprawia, że **operacja może być kontynuowana na dowolnym wątku z puli**.

```
await SomeAsyncMethod().ConfigureAwait(false);

//Jak to działa?
//1. Domyślnie - ConfigureAwait(true)
[await] -> oddaj wątek -> zakończ operację -> wróć na UI thread

//2. Z ConfigureAwait(false)
[await] -> oddaj wątek -> zakończ operację -> kontynuuj na dowolnym wątku
```

ConfigureAwait(false)

Jeżeli ustawimy `ConfigureAwait(false)` w kodzie z przykładu o deadlockach, uda nam się tego **deadlocka ominąć** – jednak, nadal nie powinniśmy tego robić w ten sposób.

```
private void button1_Click(object sender, EventArgs e)
{
    var res = GetHelloAsync().Result;
    MessageBox.Show(res);
}

private async Task<string> GetHelloAsync()
{
    //Nie wracamy do wątku UI po await
    await Task.Delay(1000).ConfigureAwait(false);
    return "Hello, World!";
}
```

Anulowanie operacji asynchronicznej

Czasami operacja asynchroniczna może trwać bardzo długo (zapytania HTTP, operacje na bazie danych itd.). Do tego służy `CancellationToken`. **Pozwala on na przerwanie operacji asynchronicznej w kontrolowany sposób** – bez zabijania wątku, bez wyjątków systemowych i bez bałaganu.

```
private async Task LoadDataAsync(CancellationToken cts)
{
    for (var i = 0; i < 1_000_000 && !cts.IsCancellationRequested; i++)
    {
        //Tu możemy robić coś ciężkiego...
        await Task.Delay(500, cts);
    }
}

//Tworzymy CancellationTokenSource
var cts = new CancellationTokenSource();
var t = LoadDataAsync(cts.Token); // <-- Wysyłamy token

//Okej, przerywamy akcję
cts.Cancel();
```

Asynchroniczne wyrażenia lambda

Funkcje anonimowe i wyrażenia lambda również mogą być asynchroniczne.

```
async x => await DoAsync(x);
```

```
Func<int, Task<string>> f =
    async x => {
        await Task.Delay(1000);
        return $"Wynik: {x}";
    };
```

```
Action a = async () => await DoAsync();  // BŁĄD
```

Tutaj utworzy się async void! ❌

```
EventHandler h = async (s, e) =>
{
    await Task.Delay(1000);
};
```

Legalny przypadek - `EventHandler`

Przetwarzanie równoległe

Równoległość (*parallelism*) to **wykonywanie wielu operacji jednocześnie, fizycznie, przy użyciu wielu rdzeni procesora lub wielu maszyn**.

Kiedy CPU posiada wiele rdzeni (*4*, *8*, *12*, *16*) **każdy rdzeń może wykonywać jedno zadanie – niezależnie od pozostałych**. Wrzucenie bardzo ciężkich operacji na wiele rdzeni (ciężkie obliczenia, szyfrowanie, kompresja, rendering) często **kilkukrotnie przyspiesza obliczenia**.

Oczywiście jest tutaj **pewne uproszczenie**, i mówimy o rdzeniach wirtualnych (*wątkach*). Współczesne procesory z *hyper-threadingiem* posiadają różne konfiguracje:

- **Intel Core i5 1240p** – 12 rdzeni, 16 wątków (P + E).
- **Intel Core Ultra 9 285K** – 24 rdzenie, 24 wątki. (*bez hyper-threading*)
- **AMD Ryzen 9 9950X3D** – 16 rdzeni, 32 wątki.

Równoległość v asynchroniczność

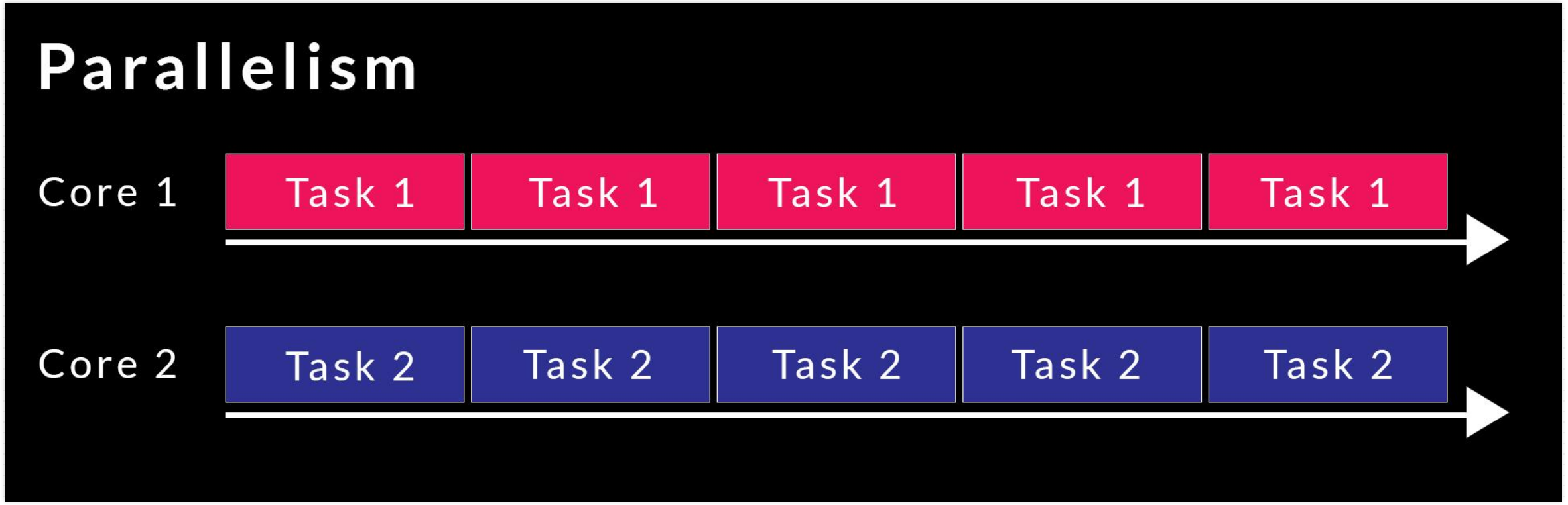
Asynchroniczność

- Nie tworzy nowych wątków.
- Nie używa dodatkowych rdzeni.
- Pozwala jedynie **na nieblokowanie wątku**.

Równoległość

- Uruchamia **wiele operacji jednocześnie**.
- Zużywa **więcej niż jeden rdzeń**.
- Jest przeznaczona do **obliczeń** (*CPU-bound*).

Przetwarzanie równoległe



Przetwarzanie równolegle

Wyobraźmy sobie ponownie Żabkę, ale tym razem mamy **dwóch kasjerów** i **dwa stanowiska**. Do sklepu przyszło również dwóch klientów:

- **Ania**, która chce kupić **hot-doga**,
- **Jan**, który chce kupić **wege hot-doga**.

Pierwszy kasjer zaczyna przygotowywać hot-doga dla Ani, a drugi kasjer w tym samym czasie przygotowuje wege hot-doga dla Jana.

Obaj pracują **równolegle** – każdy wykonuje swoją część pracy niezależnie od drugiego.

Pula wątków – `ThreadPool`

`ThreadPool` to **zarządzana przez .NET pula wątków**, które są ponownie używane, aby:

- nie tworzyć nowych wątków za każdym razem (*co jest drogie*),
- poprawić wydajność,
- lepiej wykorzystywać wszystkie rdzenie,
- uniknąć niekontrolowanej liczby wątków.

.NET sam dba o:

- Tworzenie nowych wątków tylko jeśli to konieczne.
- Usypianie nieużywanych wątków.
- Balansowanie obciążenia.

Pula wątków – **ThreadPool**

1. Najpierw wrzucasz zadanie (*do wykonania w osobnym wątku*).
2. Zadanie trafia do kolejki w puli wątków.
3. Wątek, który jest wolny, pobiera zadanie i je wykonuje.
4. Po wykonaniu zadania, wątek wraca do puli jako wolny i czeka bądź bierze kolejne zadanie.

Kluczowe jest, że wątki nie są tworzone i niszczone – ThreadPool zapewnia pewnego rodzaju recykling.

```
//Uruchamianie DoWork() w wątku z puli
Task.Run(() => DoWork());
```

Pula wątków – `ThreadPool`

Ile wątków ma ThreadPool? **To zależy** - .NET dobiera tę liczbę dynamicznie.

- W zależności od procesora.
- W zależności od potrzeb (*w trakcie działania programu*).

Pula wątków – **ThreadPool**

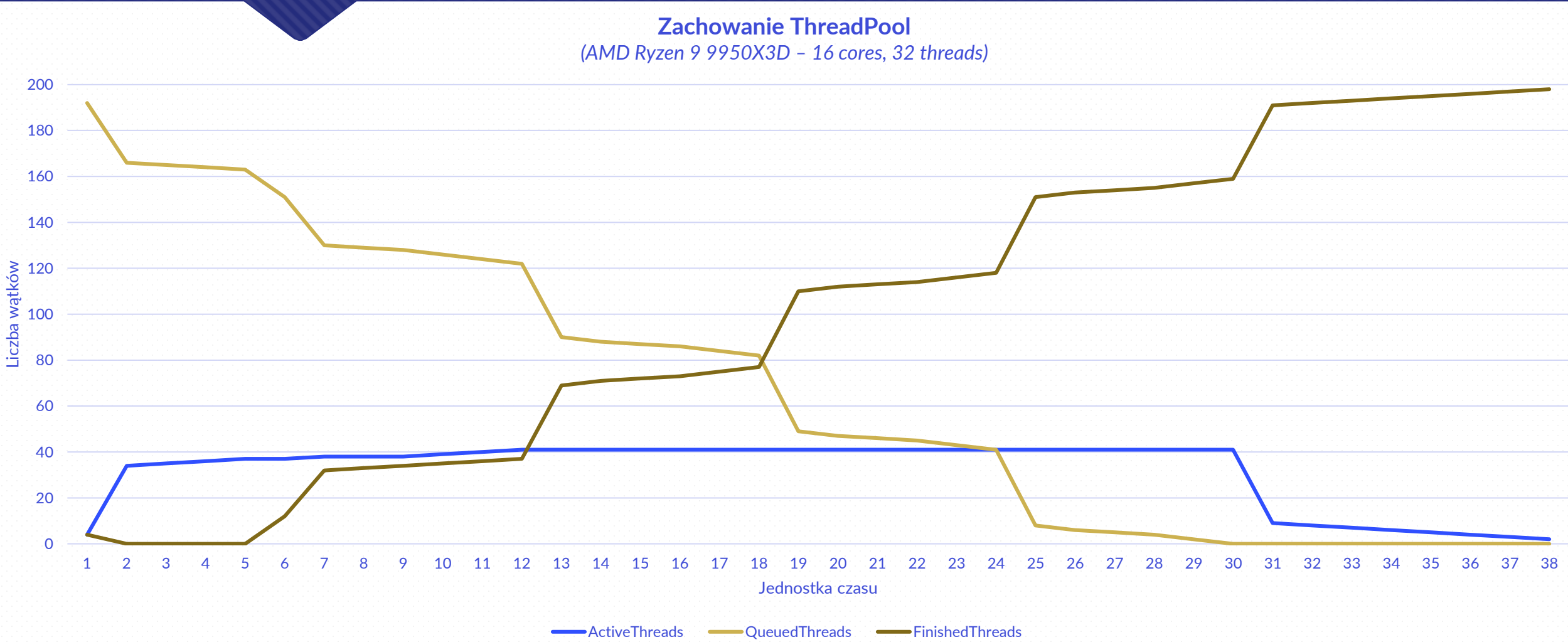
```
Console.WriteLine($"Startujemy {TASKS_COUNT} zadań blokujących...");
var sw = Stopwatch.StartNew();

List<Task> tasks = new();
for (int i = 0; i < TASKS_COUNT; i++)
{
    int id = i;
    tasks.Add(Task.Run(() =>
    {
        //Symulacja ciężkiej, blokującej operacji
        Thread.Sleep(5000);
    }));
}
```

Przeprowadźmy pewien eksperyment...

TASKS_COUNT = 200

Pula wątków – `ThreadPool`



Pula wątków – `ThreadPool`

Co trafia do puli wątków?

- `Task.Run()`
- Operacje z biblioteki Parallel – `Parallel.For`, `Parallel.ForEach`.
- Parallel LINQ (PLINQ)

`ThreadPool` **nie gwarantuje kolejności wykonywania zadań**, nie obsługuje priorytetów.

Wątki tworzone ręcznie

W .NET wątki można tworzyć ręcznie, jednakże jest to:

- **Drogie**.
  - Zajmuje kilka MB pamięci stosu.
  - Jest zasobem systemowym.
  - Stworzenie go zajmuje dużo czasu procesora.
  - Nie podlega optymalizacjom .NET.
- **Trudne w zarządzaniu**.
- **Nie współpracują z `async` i `await`**.
- **Możesz ich utworzyć bardzo dużo (*brak kontroli*)** – może prowadzić do stworzenia zbyt wielu wątków.
- Niepotrzebne w **99%** przypadków – *tak, są przypadki w których to podejście działa najlepiej*.

```
var t = new Thread(() => DoWork());
t.Start();
```

Uruchamianie wątku – `Task.Run()`

Najprostszy przykład – wykonanie **operacji ciężkiej dla procesora** (*CPU bound*)

Co tutaj się dzieje?

- Kod nie wykonuje się w wątku UI.
- .NET wrzuca zadanie do puli wątków (`ThreadPool`).
- Jeżeli są dostępne rdzenie – zadanie wykona się równolegle.
- Dzięki `await` powrócimy do poprzedniego wątku, po zakończeniu obliczeń.

```
await Task.Run(() => ExpensiveCalculation());
```

Uruchamianie wątku – `Task.Run()`

Wątek (*podobnie jak to było w przypadku asynchroniczności*) możemy przypisać do zmiennej, dzięki temu:

- **`Task` rozpoczyna pracę natychmiast** (*o ile są wolne wątki w puli*).
- Możesz w tym czasie **wykonywać inne czynności**.
- Używasz **`await`**, dopiero **kiedy potrzebujesz wyniku** – jeżeli go nie ma, **poczekasz na niego**.

To jest podstawa **współbieżnego** uruchamiania wielu obliczeń.

```
var task = Task.Run(() => ExpensiveCalculation());
// ... tu możesz robić inne rzeczy
await task; // czekasz na wynik
```

Uruchamianie wątku – `Task.Run()`

Jeżeli chcesz, aby operacje **zostały wykonane w innym wątku**, ale **potrzebujesz wyniku**. `Task.Run()` pozwala również na **odebranie wyniku obliczeń z innego wątku**.

```
int result = await Task.Run(() =>
{
    return CalculateValue();
});
```

```
int result = await Task.Run(CalculateValue);
```

Biblioteka `Parallel`

Biblioteka `Parallel` z przestrzeni nazw `System.Threading.Tasks` pozwala wykonywać wiele operacji **fizycznie równolegle**, wykorzystując **wszystkie dostępne rdzenie CPU**.

To idealne narzędzie do **operacji ciężkich dla CPU**: obliczeń, przetwarzania danych, symulacji, kompresji itd.

Metoda `Parallel.For()`

Metoda `Parallel.For` to równoległa wersja zwykłej pętli `for`.

- Elementy (*0…9*) są przetwarzane równocześnie.
- .NET dynamicznie przydziela zadania do rdzeni.
- Działa znacznie szybciej przy dużych pętlach CPU bound.

Kolejność wykonania zadań w pętli nie jest gwarantowana!

```
Parallel.For(0, 10, i =>
{
    ProcessItem(i);
});
```

Metoda `Parallel.ForEach()`

```
Parallel.ForEach(items, item =>
{
    Process(item);
});
```

Tutaj nie będziemy się rozpisywać... metoda działa tak samo jak `Parallel.For()`

Biblioteka `Parallel`

`Parallel` używa **ThreadPoola**, który:

- mierzy obciążenie procesora,
- monitoruje opóźnienia zadań,
- dynamicznie zwiększa lub zmniejsza liczbę równoległych wątków.

Dzięki temu:

- równoległość nie zabija systemu,
- program nie tworzy niepotrzebnych wątków,
- wydajność jest optymalna.

Parallel LINQ

PLINQ to rozszerzenie klasycznego LINQ, które pozwala wykonywać zapytania **równolegle na wielu rdzeniach**. Wystarczy dodać **`AsParallel()`**, aby LINQ zaczęło wykorzystywać `ThreadPool` i równoległość CPU.

Co się dzieje?

- kolekcja jest dzielona na partycje,
- każda partycja jest przetwarzana na innym rdzeniu,
- wyniki są scalane na końcu.

Jeżeli operacje wykonywane przez PLINQ są wymagające dla CPU, przyniesie to ogromny wzrost wydajności.

```
var results = numbers
    .AsParallel()
    .Select(n => HeavyComputation(n)) // <-- Ciężkie dla CPU!
    .ToList();
```

Stopień równoległości

Stopień równoległości (*degree of parallelism*) określa **ile wątków równocześnie może wykonywać dane zadania**.

Domyślnie .NET używa maksymalnej liczby dostępnych rdzeni, ale czasami **chcemy to ograniczyć** – przykładowo, by nie obciążyć całego systemu.

```
var results = data
    .AsParallel()
    .WithDegreeOfParallelism(4) // <-- Maksymalnie 4 wątki
    .Select(x => HeavyCompute(x))
    .ToList();
```

PLINQ z ustawionym stopniem równoległości

```
Parallel.For(0, items.Length,
    new ParallelOptions { MaxDegreeOfParallelism = 4 },
    i =>
    {
        Process(items[i]);
    }
);
```

`Parallel.For` z ustawionym stopniem równoległości

Synchronizacja w przetwarzaniu równoległym

Równoległość działa świetnie, gdy zadania są **niezależne**. Ale jeśli pracujące równolegle wątki muszą korzystać ze **wspólnego zasobu**, trzeba je zsynchronizować, aby nie doszło do błędów. Synchronizacja to narzucenie wątkom **kolejności lub wyłączności** na wykonywanie pewnych fragmentów kodu w danym czasie.

Dlaczego w ogóle jest synchronizacja?

- Wątki wykonują się równocześnie.
- Nie ma gwarancji, co do kolejności uruchomienia i zakończenia wątku.
- Wątki mogą się spierać o dostęp do zasobów.

```
int counter = 0; // <-- wynikiem nigdy nie będzie 1.000.000
Parallel.For(0, 1_000_000, i =>
{
    counter++; // tu powstaje bałagan
});
```

Synchronizacja – kiedy jest potrzebna?

Synchronizacja jest niezbędna, kiedy:

- Wątki modyfikują wspólną zmienną (*np. licznik, suma, indeks itd.*).
- Zapisują dane do wspólnej kolekcji (*np. lista czy słownik*).

Nie potrzebujemy synchronizacji, kiedy:

- Operacje są od siebie niezależne (*np. każdy wątek operuje na innym pliku*).
- Gdy każdy wątek pracuje na własnej kopii danych.
- Kiedy korzystamy z kolekcji bezpiecznych wątkowo.

Synchronizacja – słowo kluczowe `lock`

Słowo kluczowe `lock` chroni fragment kodu **przed jednoczesnym dostępem z wielu wątków**. W danym momencie **tylko jeden wątek** może wejść do sekcji objętej lockiem. To najprostszy i najczęściej używany mechanizm synchronizacji w .NET.

```
private readonly object _lockObj = new object();

lock (_lockObj)
{
    // tylko jeden wątek naraz może wejść tutaj
    sharedCounter++;
}
```

Dobra praktyka – zawsze tworzymy osobny obiekt dla każdego locka

`lock` i deadlock

Nie wolno wywoływać metod asynchronicznych z wnętrza sekcji `lock`, prowadzi to do deadlocka.

- `lock` blokuje wątek.
- `await` oddaje wątek.
- Inny wątek może spróbować zdobyć lock – **deadlock**.

Do synchronizacji operacji równoległych, które korzystają z asynchroniczności korzystamy z `SemaphoreSlim` – nigdy z `lock`.

```
lock (_obj)
{
    await SomethingAsync(); // BŁĄD!
}
```

Synchronizacja – semafory

`SemaphoreSlim` pozwala ograniczyć **ile wątków jednocześnie** może wejść do danej sekcji kodu – najczęściej 1 (*czyli działa jak „asynchroniczny lock”*).

```
private readonly SemaphoreSlim _sem = new SemaphoreSlim(1, 1);

public async Task DoWorkAsync()
{
    await _sem.WaitAsync(); //Wejście do sekcji krytycznej

    try
    {
        await SomethingAsync(); //Bezpieczne async operacje
    }

    finally
    {
        _sem.Release(); //MUSI być w finally
    }
}
```

Efekt:

- tylko 1 wątek na raz **zostanie przepuszczony przez semafor**.
- **`await`** nie blokuje wątku
- nie powoduje deadlocków w `async`

Na semafor można również czekać synchronicznie - `_sem.Wait()`. Jednakże jest to podejście bardzo bezsensowne, na wielu poziomach… **Dlaczego?**

Parametry:

- **currentCount** – jeden wątek może wejść. (*ilość wolnych miejsc*)
- **maxCount** – nigdy więcej niż jeden wątek. (*górny limit currentCount*)

Kolekcje bezpieczne wątkowo

W klasycznych kolekcjach (listy, słowniki, kolejki itd.) nie wolno jednocześnie zapisywać lub modyfikować danych z wielu wątków, prowadzi to między innymi do:

- Sytuacji „*kto pierwszy*” – nie mamy pewności, czy wartości zostaną dobrze ustawione.
- Wyjątków – `System.InvalidOperationException`.
- Utraty danych.

.NET oferuje specjalne kolekcje **Concurrent**, zaprojektowane do łatwej obsługi z wielu wątków.

ConcurrentDictionary<TKey, TValue>

`ConcurrentDictionary` to najczęściej używana kolekcja thread-safe.

Zalety:

- atomowe dodawanie, aktualizowanie i usuwanie,
- idealna do liczników, cache, agregacji danych,
- działa dobrze w równoległości (Parallel, PLINQ),
- bardzo dobrze zoptymalizowana.

```
var dict = new ConcurrentDictionary<string, int>();

dict.TryAdd("A", 1);
dict.AddOrUpdate("A", 1, (key, old) => old + 1);
```

ConcurrentBag<T>

Dobrze działa, gdy wiele wątków dodaje i pobiera elementy **niezależnie**.

Zastosowania:

- tymczasowe, lokalne buforowanie wielu elementów,
- równoległe zbieranie wyników, gdy kolejność nie ma znaczenia.

Kolejność elementów w tej kolekcji jest nieprzewidywalna.

```
var bag = new ConcurrentBag<int>();

bag.Add(1);
bag.TryTake(out var x);
```

Czy kolekcje bezpieczne wątkowo są lepsze?

Kolekcje thread-safe służą wyłącznie do używania w scenariuszach, kiedy do kolekcji będzie odwoływał się więcej niż jeden wątek jednocześnie. Przez koszt synchronizacji, operacje na nich są bardziej wymagające niż w przypadku tradycyjnych kolekcji.

- Jeżeli możesz uniknąć operacji na jednej kolekcji z wielu wątków, użyj zwykłej kolekcji.
- Jeżeli kolekcja musi być współdzielona między wątkami, możesz użyć kolekcji thread-safe.

Oprócz `ConcurrentBag` i `ConcurrentDictionary` są jeszcze takie kolekcje jak `ConcurrentStack` czy `ConcurrentQueue`.

Operacje atomowe

Wszystkie **operacje atomowe** w .NET są bezpieczne wątkowo – często nie dają one gwarancji poprawności danych, ale w niektórych scenariuszach pozwalają **uniknąć konieczności synchronizacji kodu**.

Przykładem operacji atomowej w .NET jest **odczyt i zapis referencji** – jest to wykonywane **jednym, nieprzerywalnym krokiem CPU**. Odczyt i zapis prostych typów jak `int`, `double`, `bool`, `long` również są operacjami atomowymi (*inkrementacja już nie*).

```
private List<int> _data = new List<int>();

public void UpdateData()
{
    var newList = _data.ToList(); //kopia
    newList.Add(999);

    _data = newList; //ATOMOWA podmiana referencji
}
```

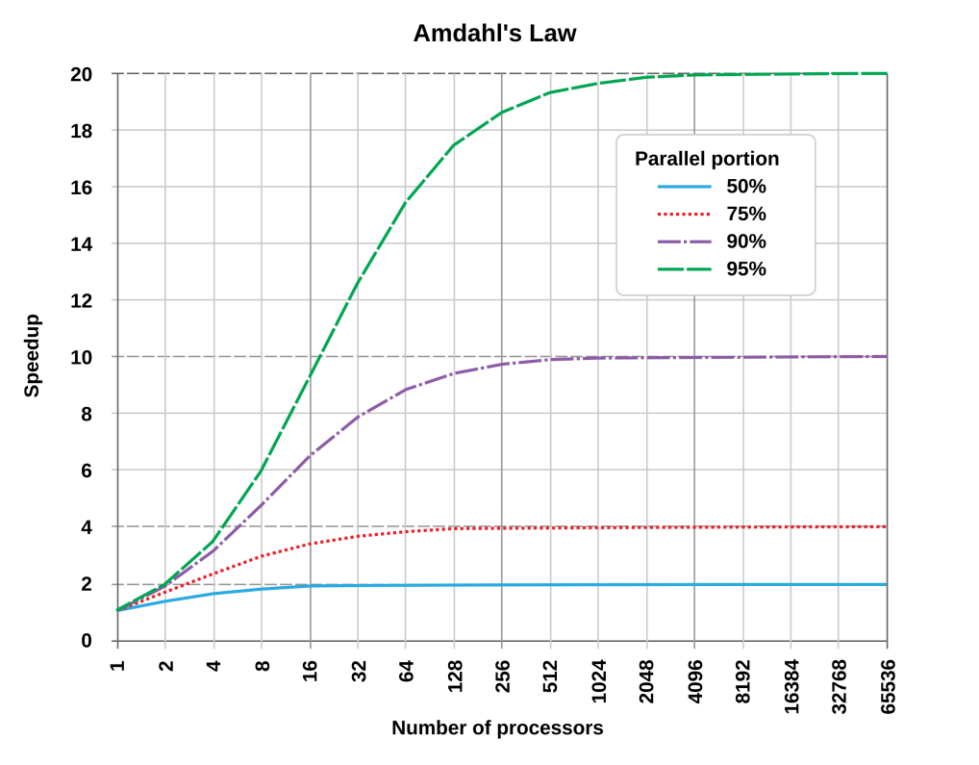
Prawo Amdahla

„Przyspieszenie programu zależy od tej części, którą da się zrównoleglić.

Fragment sekwencyjny ogranicza skalowanie całości."

Czyli nawet jeśli masz 128 rdzeni, program nie przyspieszy więcej, niż pozwala „*wąskie gardło*", które działa sekwencyjnie.

Prawo Amdahla



Masz program, w którym:

- 80% czasu można wykonać równolegle
- 20% musi być wykonane sekwencyjnie

Nawet na **nieskończonej liczbie rdzeni**, czas wykonania wyniesie min 20% oryginalnego czasu.

Maksymalne możliwe przyspieszenie to x5.

$$\text{Speedup} = \frac{\text{Performance for the entire task when enhancements are applied}}{\text{Performance for the same task without those enhancements}}$$

Podsumowanie

- Równoległość działa tylko tam, **gdzie da się pracować niezależnie**. Każda sekcja sekwencyjna ogranicza maksymalne przyspieszenie – i to właśnie opisuje Prawo Amdahla.
- Planując zrównoleglenie algorytmu, trzeba się liczyć często z kosztami synchronizacji – źle zoptymalizowana, może być bardzo kosztowna (*zniweluje boost otrzymany przez równoległość*).
- Zrównoleglając algorytmy należy pamiętać, że różni klienci mają różne procesory – inna ilość wątków. Stąd często sama optymalizacja jest bardzo trudna.