Wrocław, 22 października 2024

Programowanie **r**ównoległe i **r**ozproszone

Laboratorium 1

Programowanie równoległe w `C#`

prowadzący: *Dr inż. Radosław Idzikowski*

1 Wprowadzenie

Celem laboratorium jest przypomnienie podstaw programowania równoległego w języku programowania `C#` na przykładzie prostej symulacji pracy górników. Czas przewidziany na wykonanie zadania to 1 termin wraz z ocenieniem pracy. Praca będzie oceniana na bieżąco w trakcie zajęć wraz z postępem programowania kolejnych przykładów. **Po ukończeniu każdego zadania należy zawołać prowadzącego w celu zaakceptowania etapu i odnotowania postępów.**

2 Zadania

W ramach zajęć należy w zespołach wykonać następujące zadania:

1. Podstawowa symulacja z wykorzystaniem klasy `Task` oraz semaforów.
2. Dodanie podglądu stanu symulacji `on-line`
3. Pomiar przyśpieszenia i efektywności.

*. Zwiększenie liczby kopalni.

Za wykonanie zadania nr 1 jest ocena dostateczna, za każde kolejne zadanie jest +1 do oceny. Na ocenę bardzo dobrą (5.0) należy wykonać wszystkie trzy zadania. Zadanie oznaczone gwiazdką jest dla chętnych. Po zakończeniu zajęć należy kody źródłowe napisanych programów przesłać do prowadzącego.

3 Opis zadań

3.1 Zadanie 1

Zadanie polega na stworzeniu programu symulującego równoległe wydobycie, a następnie transport węgla ze wspólnego złoża do magazynu. Każdy górnik (wątek) podjeżdża do złoża, gdzie wydobywa węgiel aż do zapełnienia swojego pojazdu (**wydobycie odbywa się po jednej jednostce węgla z założoną prędkością**). Po zapełnieniu pojazdu, górnik transportuje węgiel do magazynu, a następnie go rozładowuje. Program kończy się, gdy cały węgiel zostanie przeniesiony ze złoża do magazynu. Synchronizacja dotyczy zarówno złoża, gdzie w jednym momencie może wydobywać tylko jednocześnie dwóch górników, jak i magazynu, gdzie na raz może rozładowywać się maksymalnie tylko jeden pojazd. Program ma pokazywać, jak zwiększenie liczby górników wpływa na czas przeniesienia węgla, uwzględniając problemy z dostępem do współdzielonych zasobów, co powoduje zatory przy załadunku i rozładunku. To zadanie pozwoli zobaczyć, jak liczba górników wpływa na wydajność systemu i jak zarządzanie współdzielonymi zasobami może mieć kluczowe znaczenie dla optymalizacji procesów równoległych. Jako dane wejściowe należy przyjąć następujące założenia:

- rozmiar źródła: 2000,

- pojemność pojazdu: 200,
- czas pozyskania jednej jednostki węgla: 10 ms,
- czas rozładowania jednej jednostki węgla: 10 ms,
- czas przejazdu między kopalnią a magazynem: 10 s.

W ramach tego zadania w pełni wystarczające będzie wyjście w formie logów informujących o zmianie statusu górnika. Poniżej przedstawiono przykładowe komunikaty:

```
Górnik 3 wydobył 200 jednostek węgla. Pozostało w złożu: 1600 jednostek.
Górnik 1 wydobył 200 jednostek węgla. Pozostało w złożu: 1600 jednostek.
Górnik 3 transportuje węgiel do magazynu...
Górnik 1 transportuje węgiel do magazynu...
Górnik 1 rozładowuje węgiel...
Górnik 2 wydobył 200 jednostek węgla. Pozostało w złożu: 1200 jednostek.
Górnik 4 wydobył 200 jednostek węgla. Pozostało w złożu: 1200 jednostek.
Górnik 4 transportuje węgiel do magazynu...
Górnik 2 transportuje węgiel do magazynu...
Górnik 4 rozładowuje węgiel...
Górnik 5 wydobył 200 jednostek węgla. Pozostało w złożu: 812 jednostek.
Górnik 5 transportuje węgiel do magazynu...
Górnik 1 wydobył 200 jednostek węgla. Pozostało w złożu: 800 jednostek.
```

```
using System;
using System.Threading;
using System.Threading.Tasks;
namespace Semaphore
{
    class Program
    {
        static int sharedResource = 0;
        static SemaphoreSlim semaphore = new SemaphoreSlim(2, 2);
        static object lockObject = new object();
        static void Main(string[] args)
        {
            Task[] tasks = new Task[5];
            for (int i = 0; i < tasks.Length; i++)
            {
                tasks[i] = Task.Run(() => AccessSharedResource());
            }
            Task.WaitAll(tasks);
            Console.WriteLine("Wartość współdzielonego zasobu na końcu: " +
    sharedResource);
        }
        static void AccessSharedResource()
        {
            for (int i = 0; i < 5; i++)
            {
                semaphore.Wait();
                Thread.Sleep(100);
                lock (lockObject)
                {
                    sharedResource++;
                    Console.WriteLine($"Wątek {Task.CurrentId} zmodyfikował zasób
     na: {sharedResource}");
                }
                semaphore.Release();
            }
        }
    }
}
```

Listing 1: Implementacją z wykorzystaniem podstawowych operacji na wątkach

Na listingu 1 przedstawiono prosty kod do obsługi wątków z wykorzystaniem klasy `Task` z dostępem no współdzielonych zasobów. Zadania `Task` tworzymy trochę analogicznie do standardowych wątków `Thread`, ale ich obsługa jest znacznie łatwiejsza. Podczas tworzenia zadania uruchamiany je za pomocą metody `Run`, gdzie jako argument możemy w postaci funkcji `lambda` przypisać funkcję, która ma być uruchamiana wielowątkowo. W celu wymuszenia czekania za zakończenie wszystkich zadań, należy skorzystać z funkcji statycznej `WaitlAll` z przestrzeni `Task`, jako parametr można podać pulę zadań w formie tablicy.

Do poprawnego działania programu jeszcze potrzebujemy kilka elementów zasób współdzielony (zmienna `sharedResource`), semafor (`SemaphoreSlim` – stanowi lekką alternatywę dla `Semaphore`, ograniczającą liczbę wątków, które mogą uzyskiwać dostęp do zasobu lub puli zasobów jednocześnie.) oraz obiekt do blokowania sekcji krytyczniej (zmienna `lockObject`). Funkcja `AccessSharedResource` za zadanie wykonać 5 czasochłonnych operacji, które mogą wykonywać się niezależnie dla każdego wątku, ale załóżmy, że naraz mogą być wykonywane jedynie dwie takie operacje. W tym celu został utworzony semafor, który ma domyślnie dwa dostępne sloty na dwa możliwe (`SemaphoreSlim (2 , 2)`). Metodą `Wait` dla konkretnego semafora, możemy zarezerwować slot i czekać do uzyskanie dostępu. Następnie są wykonywane kolejne instrukcji aż do zwolnienie zasobu metodą `Relase`. Częśc krytyczną, np. zmianę wartości zasobu współdzielonego należy zablokować z użyciem mechanizmu `lock`.

3.2 Zadanie 2

Zadanie polega na dodaniu podglądu statusu symulacji `on-line`. Do wyświetlania pomocne będzie polecenie odpowiedzialne za ustawienia pozycji kursora w konsoli, np. w pozycji początkowej `Console.SetCursorPosition(0, 0);`, pierwszy argument jest odpowiedzialny za nr znaku w wierszu, a drugi za numer wiersza. **UWAGA!** Ustawienie kursora oraz wypisanie odpowiedniego statusu musi być otoczone mechanizmem zamka, żeby tekst pojawił się w zamierzonym wierszu. Do aktualizacji stanu złoża i magazynu trzeba stworzyć osobny wątek. Przykład podglądu przedstawiono poniżej.

```
Stan złoża: 5 jednostek węgla
Stan magazynu: 110 jednostek węgla

Górnik 1 zakończył pracę.
Górnik 2: Wydobywa węgiel......
Górnik 3: Transportuje do magazynu...
Górnik 4: Rozładowuje węgiel...
Górnik 5: Transportuje do magazynu...
```

W celu ociągnięcia zamierzonego efektu niezbędne będzie wymuszenie stałej szerokości wyświetlanego `string`, co można wymusić stosując następującą konstrukcję

```
Console.WriteLine($"Stan złoża: {coalDeposit} jednostek węgla".PadRight(50));
```

3.3 Zadanie 3

Koeljnym zadaniem jest pomiar sprawności naszej symulacji poprzez sprawdzenie jak na pracę naszej kopalni wpływa dodawanie kolejnych górników. W tym celu należy sprawdzić czas każdej symulacji, następnie policzyć przyśpieszenie oraz efektywność. Przykładowe wartości dla uruchomionej symulacji przedstawiono poniżej.

```
liczba górników: 1, czas: 163,95 s przyśpieszenie: 1    efektywność: 1
liczba górników: 2, czas:  84,82 s przyśpieszenie: 1,93 efektywność: 0,96
liczba górników: 3, czas:  65,31 s przyśpieszenie: 2,51 efektywność: 0,83
liczba górników: 4, czas:  52,26 s przyśpieszenie: 3,13 efektywność: 0,78
liczba górników: 5, czas:  45,18 s przyśpieszenie: 3,62 efektywność: 0,72
liczba górników: 6, czas:  44,44 s przyśpieszenie: 3,68 efektywność: 0,61
```

3.4 Zadanie *

W tym zadaniu należy rozbudować symulację o kolejne złoże z węglem. Można też zwiększyć przepustowość magazynu. Drugie złoże ma być bardziej oddalone od pierwszego względem magazynu. Ponadto należny sprawdzić różne techniki wyznaczania celu dla górnika: (1) losowo, (2) tam gdzie jest więcej wolnych slotów. W celu sprawdzenia dostępnych slotów dla semefora należy użyć jego metody `CurrentCount`.