ZAAWANSOWANE PROGRAMOWANIE
LAB03: WIELOWĄTKOWOŚĆ

Data: 7.11.2023

LABORATORIUM

Projekt dostępny na stronie pod adresem:

http://www.cs.put.poznan.pl/mradom/teaching/laboratories/ZaawProg/WindowsFormsApp_Multithread.7z

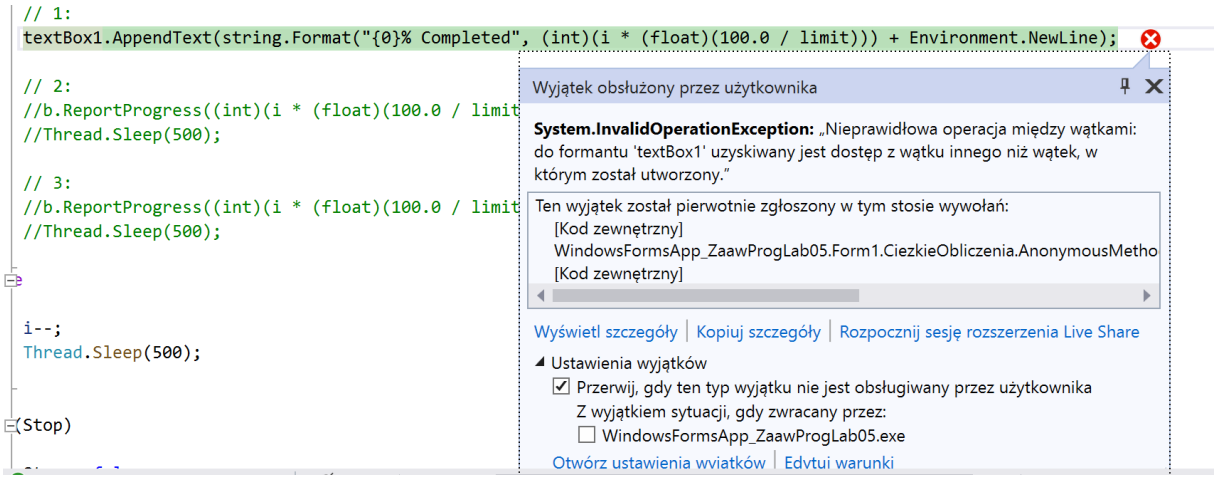
Program stanowi działający przykład jak wykorzystać jeden z wielu mechanizmów wielowątkowości (w tym wypadku obiekt klasy **BackgroundWorker**) w swoim programie.

To laboratorium najlepiej robić w ten sposób, że rozpakowują sobie Państwo projekt, a uruchamiacie dopiero wtedy, jak w tym tekście będzie mowa o uruchomieniu czy zmienianiu czegoś.

Zacznijmy od początku, czyli od tego, że projekt się Państwu bez problemu skompiluje, zapewne wszyscy klikniecie przycisk w lewym dolnym rogu „Udawaj że ciężko pracujesz" i program się wywali z komunikatem wyjątku. Klasyk. Problem stanowi niewinna w sumie linia:

```
textBox1.AppendText(string.Format("{0}% Completed", (int)(i * (float)(100.0 / limit)))
+ Environment.NewLine);
```

której jedynym celem życiowym jest dodawanie do textBox1 pewnej liczby (mniejsza o sam wzór). W normalnych (jednowątkowych) warunkach ta linia nic złego by nie spowodowała. Tym razem efekt jest taki:



Wyjątek dość dobrze tłumaczy problem: są dwa wątki i taka bezpośrednia komunikacja jak dotąd w ramach jednego okna nie jest już możliwa. Zanim opiszę tutaj jak to wszystko działa, proszę zakomentować tą nieszczęsną linię w kodzie i odkomentować dwie dalsze:

```
b.ReportProgress((int)(i * (float)(100.0 / limit)));
Thread.Sleep(500);
```

Zanim przejdziemy jednak do opisu tego kodu, proszę sprawdzić na świeżo uruchomionym programie jak działa prawy dolny przycisk „Uprzywilejowany progressBar”. Jego kod jest prosty:

```
progressBar1.Maximum = 200;
progressBar1.Minimum = 0;
for (int i = 0; i < 200; i++)
{
	Thread.Sleep(20);
	progressBar1.Value = i + 1;
}
```

Pętla od 1 do 200 kroku nakazuje zasnąć wątkowi tego kodu na 20 milisekund, a następnie aktualizuje stan paska postępu o krok *i* (Sleep(…) działa w milisekundach, czyli dopiero Thread.Sleep(1000) nakaże „spać” wątkowi na 1 sekundę). Wrócimy do tego później.

Gdy teraz ponownie uruchomią Państwo program, proszę znowu wcisnąć lewy dolny przycisk „Udawaj że ciężko pracujesz”. Program się nie „wywali”, zacznie co chwila (co pół sekundy, vide: Thread.Sleep(500);) wyświetlać kolejne wartości co 5%. Można go nawet spauzować czy zatrzymać odpowiednimi przyciskami. Jak to wszystko działa?

Całość metody CiężkieObliczenia() to jeden osobny wątek, realizowany klasą **BackgroundWorker**. Tutaj taka dygresja: ta metoda (CiezkieObliczenia) to po prostu zwykła metoda działania przycisku (komponent Button) jaką znamy – po prostu zmieniłem jej nazwę. Stąd jej argumenty. Chodzi bardziej o jej zawartość.

Linijki:

```
BackgroundWorker bw = new BackgroundWorker();
bw.WorkerReportsProgress = true; //umożliwia powiadamianie
```

tworzą nowy obiekt BackgroundWorker, druga z nich ustawia flagę możliwości powiadomień **OD** tego wątku **DO** reszty programu (i jego wątku głównego).

Następna linia wkłada w zdarzenie **DoWork** naszego obiektu bw (klasy **BackgroundWorker**) nowe zdarzenie (Event, linia 32) realizowany delegatem (linia 33). Treść delegata to linie 34-55. To co otwiera linia 32 zamyka nawias w linii 56 (patrz: rysunek na następnej stronie, numery linii po lewej).

Active i **Stop** to zmienne globalne boolowskie w klasie Form1, ich wartości jak się można domyślić ustawiają odpowiednie przyciski okna.

```
	volatile bool Active = true;
	volatile bool Stop = false;
```

Bez zbędnej teorii: warto do takich flag do celów wielowątkowości używać rozkazu `volatile` przed typem zmiennej. W skrócie zapewnia to szybsze i stabilniejsze przesyłanie zmienionego stanu takiej zmiennej między wątkami przez mechanizmu kontrolne .NET.

```
1 odwołanie
private void CiezkieObliczenia(object sender, EventArgs e) {
    BackgroundWorker bw = new BackgroundWorker();
    bw.WorkerReportsProgress = true; //umożliwia powiadamianie
    bw.DoWork += new DoWorkEventHandler(            // what to do in the backgr
        delegate (object o, DoWorkEventArgs args) {
            BackgroundWorker b = o as BackgroundWorker;
            int limit = 20;
            for (int i = 1; i <= limit; i++) {
                if (Active) {
                    // 1:
                    //textBox1.AppendText(string.Format("{0}% Completed", (int)(
                    // 2:
                    b.ReportProgress((int)(i * (float)(100.0 / limit)));
                    Thread.Sleep(500);
                    // 3:
                    //b.ReportProgress((int)(i * (float)(100.0 / limit)), "I wan
                    //Thread.Sleep(500);
                } else {
                    i--;
                    Thread.Sleep(500);
                }
                if (Stop) {
                    Stop = false;
                    break;
                }
            }
        }
    );
```

Pod tym kawałkiem kodu z rysunku, są jeszcze tworzone dwa dodatkowe zdarzenia – dla informowania o czymkolwiek z wątku **BackgroundWorker** do okna programu oraz zdarzenie uruchamiane, gdy wątek **BackgroundWorker** się zakończy (w naszym wypadku – pętla for dojdzie do limitu). Pierwszy z tych dwóch z kolei jest uruchamiany z wnętrza pętli for, za każdym razem gdy odwołujemy się tam do metody **ReportProgress** (linia: `b.ReportProgress((int)(i * (float)(100.0 / limit)));` - którą zaraz na początku trzeba było odkomentować, aby program się nie zawieszał).

```
// co zrobić gdy zmienił się wynik
bw.ProgressChanged += new ProgressChangedEventHandler(
delegate (object o, ProgressChangedEventArgs args) {
       label1.Text = string.Format("{0}% Completed", args.ProgressPercentage);
       textBox1.AppendText(string.Format("{0}% Completed", args.ProgressPercentage) +
Environment.NewLine);

//3:
//textBox1.AppendText((args.UserState as String) + Environment.NewLine);
});

// zdarzenie uruchamiane po zakończeniu obliczeń
bw.RunWorkerCompleted += new RunWorkerCompletedEventHandler(
delegate (object o, RunWorkerCompletedEventArgs args) {
       label1.Text = "Finished!";
       textBox1.AppendText("Finished!" + Environment.NewLine);
});
```

Ostatnia linia w całym kodzie metody „CiezkeObliczenia” to uruchomienie kodu w **BackgroundWorker**:

```
bw.RunWorkerAsync();
```

Przedostatnia rzecz do zmienienia w kodzie. Proszę odkomentować linie pod komentarzem //3: w kodzie pętli **for**, zakomentować te spod dwójki. Chodzi o odkomentowanie linii:

```
b.ReportProgress((int)(i * (float)(100.0 / limit)), "I want to say something");
Thread.Sleep(500);
```

A także linii w delegacie eventu już omawianego (odkomentowanie linii na czerwono tutaj):

```
bw.ProgressChanged += new ProgressChangedEventHandler(
delegate (object o, ProgressChangedEventArgs args) {
	label1.Text = string.Format("{0}% Completed", args.ProgressPercentage);
	textBox1.AppendText(string.Format("{0}% Completed", args.ProgressPercentage) +
Environment.NewLine);

	//3:
	textBox1.AppendText((args.UserState as String) + Environment.NewLine);
});
```

Program w zasadzie działa teraz to samo, tylko dodatkowo do **ReportProgress** wysyłamy łańcuch znaków z wątku pobocznego. Ten drugi argument to może być cokolwiek, obiekt dowolnej klasy. Pierwszym argumentem dla **ReportProgress** może być tylko **int**. Ten zwykły int trochę mało, często nam zależy, aby wątek mógł wysyłać do okna także inne dane. W tym przykładzie wysyła łańcuch znaków string, więc w czerwonej linijce mamy `(args.UserState as String)`

Jednak pole **UserState** może być czymkolwiek – samo jest typu Object. To pozwala nam na wysyłanie z wątku BackgroundWorker dowolnej paczki danych ukrytej jako drugi argument wywołania metody **ReportProgress** (tej w pętli **for**).

Teraz po uruchomieniu dodatkowo na ekranie będzie się wyświetlać tekst.

Ostatnia sprawa, wracamy do paska postępu na dole okna i przycisku w prawym dolnym rogu okna. Proszę uruchomić program raz jeszcze i najpierw wcisnąć lewy dolny przycisk (uruchamianie **BackgroundWorker**) a następnie przycisku „Uprzywilejowany progressBar”.

Teraz dopiero objawi się powód, dla którego tak nazwałem ten przycisk. Otóż okazuje się, że gdy rusza pętla w przycisku zapełniającym progressBar (jest to główny wątek okna), to informacje, które spływają z **BackgroundWorker** (wątek drugi, poboczny) są ignorowane. One wszystkie „hurtem” pojawią się w oknie, ale dopiero wtedy, gdy **ProgressBar** do końca się zapełni. Na szczęście można to łatwo naprawić. W pętli kierującej wypełnianiem paska postępu, pod jej dwoma linijkami należy dopisać trzecią:

```
Application.DoEvents();
```

Całość ma wyglądać tak:

```
private void ProgressBar(object sender, EventArgs e) {
	progressBar1.Maximum = 200;
	progressBar1.Minimum = 0;
	for (int i = 0; i < 200; i++) {
		Thread.Sleep(20);
		progressBar1.Value = i + 1;
```

```
                Application.DoEvents();
        }
}
```

Dzięki tej zmianie, jeśli teraz uruchomimy nasz program po raz kolejny, najpierw wciśniemy lewy dolny przycisk a potem prawy dolny, zarówno komunikaty **BackgroundWorker** jak i pasek postępu będą równocześnie odświeżane. Zapewnia do linijka Application.DoEvents(); której każde wywołanie zmusza dany wątek (tutaj: główny wątek okna) to zatrzymania się na chwilkę i sprawdzenia, czy w międzyczasie nie spłynęły jakieś żądania zmiany czegoś w oknie – np. właśnie ze strony naszego wątku pobocznego – i oczywiście ich zrealizowania (tu: wyświetlenia komunikatów w **TextBox**).