Złap współbieżnie latające muchy

Wstęp

Artykuł przeznaczony jest dla osób znających podstawy programowania w C#, a chcących dodatkowo zapoznać się z tajnikami tworzenia programów wielowątkowych. W niniejszym artykule postaram się na przykładzie prostej gry przedstawić część mechanizmów związanych z wielowątkowością. Zaznaczam jednak, iż należy traktować tę pracę jako wprowadzenie nie wyczerpujące w pełni złożonego zagadnienia współbieżności.

Wielowątkowość

Na pewno każdy słyszał o wielozadaniowości, która polega na jednoczesnym wykonywaniu wielu programów. Dzięki wielozadaniowości możemy na przykład słuchać muzyki przy równoczesnym odbieraniu poczty i pisaniu dokumentów. Jednakże w komputerach z pojedynczym procesorem ta równoczesność jest wyłącznie złudzeniem, ponieważ w każdej chwili procesor może wykonywać tylko jedno zadanie. Zatem jak to się dzieje, iż mamy wrażenie jednoczesnego wykonywania zadań? Otóż z pomocą przychodzi nam system operacyjny, który zarządza pracą procesora. To on przydziela czas procesora kolejnym zadaniom, a ponieważ dzisiejsze procesory potrafią wykonywać ponad miliard rozkazów maszynowych na sekundę, nawet nie jesteśmy w stanie zauważyć, że wykonywanie jakiegoś zadania zostało na chwilę przerwane.

Zastanówmy się zatem czym jest i co daje nam wielowątkowość. Jak sama nazwa wskazuje, związana jest ona z „jednoczesnym” wykonywaniem wielu wątków. Każdy program składa się z co najmniej jednego wątku. Natomiast program złożony z więcej niż jednego wątku nazywamy programem wielowątkowym. Każdy wątek niezależnie od innych może wykonywać przydzielone mu instrukcje. Jednakże w przypadku, gdy wiele wątków wykonuje pewne operacje na tym samym obiekcie, wprowadzane są dodatkowo metody synchronizujące dostęp do tego obiektu. Obiekt taki nazywamy zasobem współdzielonym, a synchronizacja zapewnia nieprzerwane wykonanie całej operacji na nim. Brak synchronizacji w takich przypadkach mógłby doprowadzić do nieprzewidywalnych sytuacji. Na przykład gdyby dwa wątki miały wykonać operację x = x+1 na tej samej zmiennej, mógłby zostać wywołany następujący ciąg instrukcji:

- wątek 1 pobiera wartość zmiennej x z pamięci, po czym dodaje do niej jeden i zostaje przerwany;
- wątek 2 zostaje wznowiony, dzięki czemu też pobiera wartość zmiennej x (jeszcze nie zmienionej) z pamięci, po czym dodaje do niej jeden, przypisuje nową wartość do zmiennej x i kończy swoje zadanie;
- wątek 1 zostaje wznowiony i może teraz przypisać obliczoną wcześniej wartość do zmiennej x.

Jak widać po wykonaniu tych operacji wartość zmiennej x ostatecznie została powiększona tylko o 1, a nie o 2, tak jak oczekiwaliśmy. Oczywiście ciąg takich wywołań nie zawsze będzie miał miejsce, ale jest on prawdopodobny i może wywoływać błędy w programie.

Wypuśćmy pierwsze muchy

Przejdźmy wreszcie od słów do czynów i zacznijmy analizować przykładową grę opartą na wielowątkowości. Gra ma polegać na łapaniu współbieżnie latających much na ekranie. Zatem każda mucha będzie w tym przykładzie osobnym wątkiem odpowiedzialnym za jej

przemieszczanie. Rozpocznijmy od wypuszczenia pierwszych much, które reprezentowane będą przez pojedyncze czarne kółko. Załóżmy więc, że już stworzyliśmy klasę *Mucha*, która zawiera podstawowe operacje takie jak *rysuj* rysująca czarne kółko w miejscu, w którym znajduje się mucha, *leci* przemieszczająca muchę o stały wektor oraz *poleMuchy* zwracające prostokąt w którym znajduje się mucha. (Pełna implementacja wszystkich operacji Muchy znajduje się w pliku Mucha.cs).

Warto jednak zatrzymać się na chwilę przy funkcji *leci*:

```
public void leci ()
{
	(...)// operacje zwiazane z przemieszczeniem wspolrzednych muchy

	Thread.Sleep(rand.Next(10)+10);

	(...)// operacje zwiazane z odswiezeniem ekranu po przemieszczeniu
}
```

Znalazła się tutaj statyczna funkcja *Sleep(int)* klasy *Thread* z przestrzeni nazw *System.Threading*. Przestrzeń nazw *System.Threading* zawiera wiele klas związanych z realizacją programów wielowątkowych. Najważniejszą z nich jest właśnie klasa *Thread*, dzięki której możemy stworzyć nowy wątek w programie oraz sterować nim poprzez ustawianie jego właściwości i wywoływanie funkcji odpowiedzialnych za jego zachowanie. Jedną z takich funkcji jest *Sleep(int)* za pomocą której możemy „uśpić” wątek wywołujący tę metodę. Argumentem tej funkcji jest nieujemna liczba milisekund określająca długość czasu na jaką wątek jest blokowany. Wartość 0 przekazanego argumentu oznacza, że wątek ma zostać w tej chwili zawieszony, aby umożliwić wykonanie innym wątkom czekającym na przydzielenie im czasu procesora. Stała wartość *Infinite* argumentu blokuje wątek na czas nieokreślony. Metoda ta zmienia stan wątku na *WaitSleepJoin* (o stanach wątku już za chwileczkę). „Uśpienie” wątku zastosowane w funkcji *leci* nadaje muchom opóźnienie związane z ich przemieszczaniem po ekranie.

Stwórzmy teraz klasę *MuchaThread* wraz z funkcją *start*, którą nowy wątek będzie wykonywał na rzecz konkretnego obiektu tej klasy. Wątek powiązany z tym obiektem będzie sterował muchą.

```
public class MuchaThread
{
	private Mucha m;

	public MuchaThread(FormGra form, Random rand, int wielkoscMuchy)
	{
		m = new Mucha (form, rand, wielkoscMuchy );
	}

	public void start ()
	{
		while ( Thread.CurrentThread.IsAlive )
		{
			m.leci();
		}
	}

	public void rysuj ( Graphics g, SolidBrush pedzelek )
	{
		m.rysuj(g,pedzelek);
	}
}
```

W funkcji *start* zastosowana została właściwość *IsAlive* na rzecz wątku wykonującego tę metodę dzięki właściwości *CurrentThread*. *IsAlive* zwraca *true* jeśli wątek został uruchomiony i nie został jeszcze zakończony. Natomiast *CurrentThread* zwraca obecnie działający wątek. W tym przypadku zapewnione mamy działanie „nieskończonej" pętli do momentu przerwania wątku z zewnątrz.

Myśląc przyszłościowo stwórzmy od razu klasę *MuchyThread* wraz z funkcją *start*, którą nowy wątek będzie wykonywał na rzecz konkretnego obiektu tej klasy. Wątek powiązany z tym obiektem będzie miał za zadanie wypuszczać muchy na ekran.

```
public class MuchyThread
{
	private FormGra form;
	private int wielkoscMuchy;
	private int liczbaMuch;
	private ArrayList muchy;

	public MuchyThread(FormGra form, int liczbaMuch, int wielkoscMuchy)
	{
		this.form = form;
		this.wielkoscMuchy = wielkoscMuchy;
		this.liczbaMuch = liczbaMuch;
		muchy = new ArrayList();
	}

	public void start ()
	{
		Random rand = new Random();
		for ( int i=0; i<liczbaMuch; i++ )
		{
			MuchaThread mt =new MuchaThread(form,rand,wielkoscMuchy);
			muchy.Add( mt );

			Thread t = new Thread( new ThreadStart ( mt.start ));
			t.Name = "Mucha " + i;
			t.IsBackground = true;
			t.Start();
		}
	}

	public void rysuj (Graphics g, SolidBrush pedzelek)
	{
		for ( int i=0; i<muchy.Count; i++ )
		{
			((MuchaThread)muchy[i]).rysuj(g,pedzelek);
		}
	}
}
```

Nareszcie doszliśmy do miejsca, w którym wypuszczamy na ekran współbieżnie latające muchy. Wszystko dzieje się w metodzie *start*. W każdej iteracji pętli *for* tworzone są kolejno: obiekt klasy *MuchaThread* oraz obiekt klasy *Thread* wraz z przekazaniem do niego adresu funkcji, którą ma wykonywać poprzez delegacje *ThreadStart*. Przekazana funkcja musi być typu *public void funkcja()*. Ponieważ przekazana funkcja należy do obiektu klasy *MuchaThread* obiekt ten został wcześniej utworzony. Należy pamiętać, że utworzenie obiektu klasy *Thread* nie wiążę się z rozpoczęciem jego wykonywania. Wątek taki znajduje się początkowo w stanie *Unstarted* i jest w nim dopóki nie wywołana zostanie funkcja *Start()*, która zmieni stan wątku na *Running* (wątek rozpocznie wykonywanie swoich zadań). Każdy wątek po utworzeniu ma domyślnie ustawiany priorytet wykonywania na *Normal* (o priorytetach już za chwileczkę). W powyższym kodzie można zauważyć również, iż tuż przed

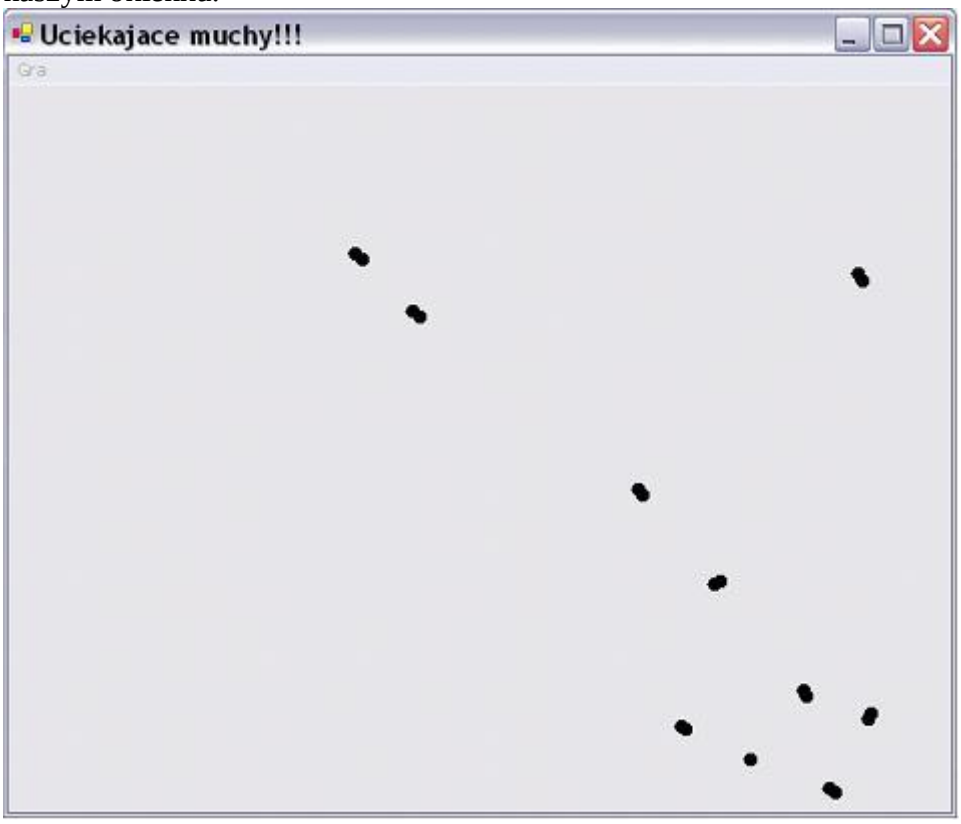
wystartowaniem wątku ustawiane są jego pewne właściwości. Właściwość *Name* pobiera lub ustawia nazwę wątku. Natomiast właściwość *IsBackground* pobiera lub ustawia wartość logiczną mówiącą o tym, czy dany wątek jest wątkiem drugoplanowym (*true*) czy też nie (*false*). Wykonywanie wątków drugoplanowych (*background*) jest automatycznie kończone w przypadku, gdy ostatni z wątków pierwszoplanowych (*foreground*) kończy działanie. Zatem wraz z zakończeniem działania programu kończone są wszystkie wątki drugoplanowe.

Aby ujrzeć nasze dzieło w akcji musimy jeszcze dorzucić do głównego formularza (*FormGra*) pozycję w menu realizującą wyświetlenie latających much wraz z przesłonięciem metody *OnPaint*:

```
private void menuGraStart_Click(object sender, System.EventArgs e)
{
	liczbaMuch=10;
	wielkoscMuchy=10;
	muchyThread = new MuchyThread(this,liczbaMuch,wielkoscMuchy);
	watekMuch = new Thread( new ThreadStart ( muchyThread.start ) );
	watekMuch.IsBackground = true;
	watekMuch.Start();
}
protected override void OnPaint(PaintEventArgs e)
{
	base.OnPaint (e);
	Graphics g = CreateGraphics();
	SolidBrush pedzelek = new SolidBrush(Color.Black);

	if ( muchyThread!=null)
		muchyThread.rysuj(g,pedzelek);
}
```

W ten oto sposób otrzymaliśmy szkielet gry wielowątkowej. Każda mucha jest sterowana przez osobny wątek. Zatem zgodnie z powyższym kodem 10 much fruwa równocześnie w naszym okienku.



Stan wątku

Właściwość *ThreadState* obiektu klasy *Thread* zawiera aktualny stan wątku. Wszystkie stany w jakich może znajdować się wątek określa typ wyliczeniowy *ThreadState*.

Zestawienie wartości typu wyliczeniowego *ThreadState*		
Wartość	Opis	Wartość liczbowa
Aborted	W wyniku wywołania metody *Abort()* wątek został zakończony.	256
AbortRequested	W wyniku wywołania metody *Abort()* zażądano zakończenia wykonywania wątku.	128
Background	Wątek wykonywany jest jako wątek drugoplanowy.	4
Running	Wątek został wystartowany i nie jest zablokowany – jest aktywny	0
Stopped	Wątek został zakończony.	16
StopRequested	Zażądano zakończenia wykonywania wątku (wartość tylko do użytku wewnętrznego).	1
Suspended	Wątek został zawieszony	64
SuspendRequested	Zażądano zawieszenia wątku	2
Unstarted	Wątek nie został jeszcze wystartowany metodą *Start()*	8
WaitSleepJoin	Wątek jest zablokowany w wyniku wywołania jednej z metod *Wait(), Sleep()* lub *Join()*	32

Warto wiedzieć, że wątek znajduje się w co najmniej jednym stanie, co oznacza, że może także w tym samym czasie znajdować się w większej liczbie stanów. Na przykład jeśli wątek jest zablokowany w skutek wywołania metody *Join()* a inny wątek wywołuje funkcję *Abort()* na zablokowanym wątku, to ten zablokowany wątek będzie znajdował się zarówno w stanie *WaitSleepJoin* jak i *AbortRequested*. W tym przypadku jak tylko wątek powróci z wywołania *Join()* lub zostanie przerwany przez inny wątek metodą *Interrupt()* otrzyma on wyjątek *ThreadAbortedException*. Każdy stan ma swoją maskę bitową (wartość liczbowa), a właściwość *ThreadState* wątku może być nałożeniem kilku masek.

Priorytet wątku

Priorytet wątku określony jest względem wszystkich innych wątków w ramach tego samego programu. Oznacza to, że w ramach jednego programu wątki o priorytecie najwyższym wykonywane są najczęściej w pierwszej kolejności. Jak już wcześniej wspomniałem domyślnie, po utworzeniu obiektu klasy Thread, obiekt ten ma priorytet *Normal*, który oczywiście można zmienić dzięki właściwości *Priority*. Jednakże wszelkie zmiany priorytetu powinny być dobrze przemyślane, ponieważ może się zdarzyć taka sytuacja, w której wątki o niższych priorytetach nigdy nie zostaną wywołane z powodu ciągłego wykonywania wątków o wyższych priorytetach. Właściwość *Priority* zawiera wartość typu wyliczeniowego *ThreadPriority*.

Zestawienie wartości typu wyliczeniowego *ThreadPriority*	
Wartość	Opis
Highest	Najwyższa wartość priorytetu.
AboveNormal	Priorytet wyższy niż normalny.

Normal	Priorytet normalny – ustawiany domyślnie.
BelowNormal	Priorytet niższy niż normalny.
Lowest	Najniższa wartość priorytetu.

Zacznijmy wreszcie łapać współbieżnie uciekające muchy

Naszym zadaniem będzie teraz umożliwienie graczowi łapania much. Będzie on chciał wybić wszystkie muchy wykorzystując do tego lewy przycisk myszy. Każde wciśnięcie tego przycisku będzie wywoływało odpowiednią funkcję obsługującą powyższe zdarzenie. Funkcja ta ma za zadanie sprawdzić czy w obrębie wciśniętego punktu znajdowała się mucha. Jeżeli tak, to należy wątek z nią związany przerwać. Aby dobrze zrealizować to zagadnienie musimy zadać sobie pytanie: czy mamy pewność, że jeśli trafimy muchę to ona na pewno zginie? Zauważmy, że po każdym wciśnięciu myszy należy przejść przez całą kolekcję much i sprawdzić czy „pole muchy" pokrywa się z „polem ataku". Ponieważ muchy ciągle latają zmieniając wciąż swe położenie, nie możemy mieć pewności, że zanim nie skończymy przeglądania kolekcji much, to ona nie wyfrunie z naszego „pola ataku". Zatem zależy nam na tym, aby atak gracza wstrzymał na chwilę wszystkie muchy i dopiero po zakończeniu akcji znów je rozruszał. Wykorzystamy do tego klasę *ManualResetEvent* przestrzeni nazw *System.Threading*, która służy do zasygnalizowania jednemu lub wielu wątkom o zajściu zdarzenia. Trzy najważniejsze funkcje, które wystarczą nam do zrealizowania zadania to *Reset()*, *Set()* oraz *WaitOne*().

ManualResetEvent dotyczy zwykle zadań, w których jeden wątek musi coś zrobić w całości zanim inne wątki będą mogły kontynuować swoje zadania. Czyli atak na muchy musi zostać wykonany w całości zanim muchy będę mogły latać dalej. Jeżeli więc zostanie rozpoczęty atak, wątek który go wywołał wykorzysta funkcję *Reset()* aby umieścić *ManualResetEvent* w stanie *nonsignaled*. Wątki much, które wywołują funkcję *WaitOne()* zostaną zablokowane w oczekiwaniu na sygnał. Kiedy atak na muchy zostanie zakończony, wątek wywoła funkcję *Set()* aby zasygnalizować, że wszystkie czekające wątki much mogą latać dalej. W momencie gdy *ManualResetEvent* jest w stanie *signaled* wątki much wywołujące WaitOne() nie są w ogóle blokowane.

Przelejmy powyższe dywagacje do kodu:
Do głównego formularza wrzucamy parę dodatkowych zmiennych pomocniczych:

```
public ManualResetEvent pilnujMuchy;
private bool myszkaWcisnieta;
private Rectangle poleWcisniecia;
private const int SZEROKOSC_POLA = 30;
```

które zainicjujemy w konstruktorze formularza głównego:

```
pilnujMuchy = new ManualResetEvent(true);
myszkaWcisnieta = false;
poleWcisniecia = new Rectangle(0,0,SZEROKOSC_POLA,SZEROKOSC_POLA);
```

Tworzymy funkcję obsługującą zdarzenie wciśnięcia myszki na formularzu:

```
private void FormGra_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{
      if ( e.Button == MouseButtons.Left )
      {
            pilnujMuchy.Reset();

            myszkaWcisnieta = true;
```

```
            poleWcisniecia.X = e.X-SZEROKOSC_POLA/2;
            poleWcisniecia.Y = e.Y-SZEROKOSC_POLA/2;
            Invalidate();

            if ( liczbaMuch>0 )
            {
                  if ( muchyThread.atakuj(poleWcisniecia) )
                        liczbaMuch--;
                  if ( liczbaMuch==0 )
                        menuGra.Enabled = true;
            }

            pilnujMuchy.Set();
      }
}
```

oraz funkcję obsługujące zdarzenie odciśnięcia myszki:

```
private void FormGra_MouseUp(object sender,
System.Windows.Forms.MouseEventArgs e)
{
      myszkaWcisnieta = false;
      Invalidate();
}
```

Do przesłoniętej funkcji *OnPaint* dodajemy rysowanie pola ataku:

```
protected override void OnPaint(PaintEventArgs e)
{
      (...)// wszystko to co wczesniej
      if ( myszkaWcisnieta )
      {
            Pen piorko = new Pen(Color.Red,2);
            g.DrawRectangle(piorko,poleWcisniecia);
      }
}
```

Do klasy *MuchyThread* dorzucamy jeszcze jedną zmienną pomocniczą, w której będziemy przechowywać wątki much:

```
private Hashtable watkiMuch = new Hashtable();
```

do funkcji *start* przed wywołaniem *t.Start()* dodajemy wypełnianie powyższej zmiennej wątkami:

```
watkiMuch.Add( mt, t);
```

dodamy jeszcze funkcję obsługującą atak:

```
public bool atakuj (Rectangle poleAtaku)
{
      for ( int i=0; i<muchy.Count; i++ )
      {
            MuchaThread mt = ((MuchaThread)muchy[i]);
            if ( mt.atakuj(poleAtaku) )
            {
                  ((Thread)watkiMuch[mt]).Abort();
                  muchy.Remove(mt);
                  return true;
            }
      }
      return false;
}
```

Ostatecznie do klasy *MuchaThread* dodajemy również funkcję sprawdzającą czy atak na muchę się powiódł:

```
public bool atakuj ( Rectangle poleAtaku )
{
      Rectangle poleMuchy = m.poleMuchy();
```

```
        poleAtaku.Intersect(poleMuchy);

        return (poleAtaku.Width*poleAtaku.Height!=0?true:false) ;
}
```

oraz zmodyfikujemy nieco funkcję *start*, ponieważ może wystąpić wyjątek *ThreadAbortException* rzucony przez system w momencie wywołania funkcji *Abort()* z funkcji *atakuj* klasy *MuchyThread*.

```
public void start ()
{
        try
        {
                while ( Thread.CurrentThread.IsAlive )
                {
                        form.pilnujMuchy.WaitOne();
                        m.leci();
                }
        }
        catch ( ThreadAbortException )
        {
                Thread.ResetAbort();
        }
}
```

Aby sprawdzić rzeczywiste działanie wprowadzonego tu obiektu *pilnujMuchy* klasy *ManualResetEvent* wystarczy skomentować linijkę *pilnujMuchy.Set();* i uruchomić grę. Podczas gry zauważymy, że po wciśnięciu myszki, żadna mucha już się nie ruszy, wiecznie czekając na sygnał. Konstruktor *ManualResetEvent* wywołany z parametrem *true* ustawia sygnał, tak że muchy mogą spokojnie latać do momentu pierwszego wciśnięcia myszki. Z kolei funkcja *ResetAbort()* klasy *Thread* powoduje anulowanie żądania przerwania przez funkcję *Abort()* i umożliwia dalsze wykonanie kodu poza sekcją *try{…}catch(…){…}*.

Wypuśćmy więcej współbieżnie latających much, ale ograniczmy ich liczbę na ekranie (wprowadzenie semafora licznikowego)

Do ograniczenia liczby much latających współbieżnie w jednej chwili na ekranie doskonale posłuży nam zaimplementowanie prostego semafora licznikowego. Semafor licznikowy jest zmienną całkowitą przyjmującą wartości nieujemne. Na semaforze S definiujemy dokładnie dwie operacje, które muszą być atomowe (niepodzielne):

- opusc(S) – opuszcza semafor:
  if S=0 then wstrzymaj wykonywanie wątku
  S:=S-1
- podnies(S) – podnosi semafor, tzn. jeśli są jakieś wątki wstrzymane przez semafor S, to wznawia jeden z nich, a następnie *S:=S+1*

Aby rozwiązać to zadanie wystarczy zainicjalizować semafor licznikowy maksymalną liczbą much, które mogą zostać wpuszczone na ekran. Zarządca much (*MuchyThread*) przed wypuszczeniem każdej muchy na ekran będzie opuszczał semafor. Zapełnienie ekranu maksymalną liczbą much spowoduje w końcu zablokowanie zarządcy. Natomiast każda trafiona mucha będzie podnosić semafor. Jeżeli zarządca będzie zablokowany, to podniesienie semafora przez muchę sprawi jego odblokowanie.

Implementacja semafora licznikowego przy wykorzystaniu klasy *Monitor* z przestrzeni nazw *System.Threading*:

```
public class Semafor
{
```

```
        private int sem;

        public Semafor()
        {
                sem = 0;
        }

        public Semafor ( int ini )
        {
                sem = ini;
        }

        public void opusc ()
        {
                Monitor.Enter(this);
                if ( sem==0 ) Monitor.Wait(this);
                sem--;
                Monitor.Exit(this);
        }

        public void podnies ()
        {
                Monitor.Enter(this);
                sem++;
                if ( sem==1 ) Monitor.Pulse(this);
                Monitor.Exit(this);
        }
}
```

Klasa *Monitor* dostarcza mechanizmów synchronizujących dostęp do obiektów. Wszystkie metody tej klasy są statyczne i nie dopuszcza się tworzenia obiektów dla niej. Wątek wywołujący funkcję *Monitora* blokującą dostęp do obiektu (referencyjnego), staje się równocześnie właścicielem tego obiektu i dopóki nie zdejmie z niego blokady, żaden inny wątek nie może z tego obiektu skorzystać. Dla każdego synchronizowanego przez *Monitor* obiektu są utrzymywane informacje na temat referencji do wątku aktualnie blokującego obiekt, referencji do kolejki wątków gotowych do przejęcia obiektu oraz referencji do kolejki wątków czekających na sygnał o zmianie stanu zablokowanego obiektu.

W przykładzie wywołanie funkcji *Monitor.Enter(this)* powoduje przejęcie obiektu klasy *Semafor* przez wątek i zablokowanie dostępu do niego dla innych wątków. Każdy wątek chcący przejąć obiekt poprzez wywołanie tej samej funkcji będzie blokowany i umieszczany w kolejce wątków gotowych do przejęcia, dopóki właściciel obiektu go nie zwolni. Właściciel może zwolnić obiekt po wywołaniu funkcji *Monitor.Exit(this)*. Instrukcje zawarte między wywołaniami *Enter* i *Exit* wchodzą w skład sekcji krytycznych, które wykonywane są tylko i wyłącznie przez jeden wątek. Funkcja *Monitor.Wait(this)* wywołana przez właściciela obiektu powoduje zwolnienie blokady obiektu i udostępnienie go innym wątkom. Jednocześnie wątek wywołujący *Wait* jest blokowany i wstawiany do kolejki wątków oczekujących na sygnał od metod *Pulse* lub *PulseAll* wywołanych przez innego właściciela obiektu. Kolejny wątek z niepustej kolejki wątków gotowych do przejęcia obiektu zakłada blokadę na obiekt i staje się jego właścicielem. Wywołanie funkcji *Monitor.Pulse(this)* powoduje wysłanie sygnału do kolejki wątków oczekujących, skąd pierwszy wątek zostaje zwolniony i przeniesiony do kolejki wątków gotowych do przejęcia obiektu.

Wykorzystanie powyższych funkcji *Monitora* gwarantuje nam poprawne działanie semafora włącznie z niepodzielnością operacji *opusc* i *podnies*.

Wykorzystajmy zatem tak zaimplementowany semafor do rozwiązania wcześniej zdefiniowanego zadania. Do klasy *MuchyThread* dołączamy nową zmienną

```
public static Semafor semafor;
```

i zmieniamy nieco konstruktor tej klasy (zmieniając jednocześnie jego wywołanie w formularzu głównym)

```
public MuchyThread(FormGra form, int liczbaMuch, int liczbaMuchLatajaych,
int wielkoscMuchy)
{
	this.form = form;
	this.liczbaMuch = liczbaMuch;
	this.wielkoscMuchy = wielkoscMuchy;

	semafor = new Semafor(liczbaMuchLatajaych);
	muchy = new ArrayList();
	watkiMuch = new Hashtable();
}
```

W metodzie *start* na początek każdej iteracji w pętli *for* dodajemy po prostu:

```
semafor.opusc();
```

Na koniec metody *start* klasy *MuchaThread* poza sekcją krytyczną *try* & *catch* dorzucamy linijkę:

```
MuchyThread.semafor.podnies();
```

W ten sposób zrealizowaliśmy nasz cel. Zgodnie z wcześniejszym omówieniem problemu, zarządca much (*MuchyThread*) przed każdym wpuszczeniem muchy na ekran opuszcza semafor. Po zapełnieniu ekranu maksymalną liczbą much zarządca jest blokowany. Natomiast mucha trafiona przez gracza znika z ekranu i podnosi semafor, dając tym samym możliwość wpuszczenia przez zarządcę kolejnej muchy na ekran.

Ograniczmy czas polowania na muchy

Ostatnim naszym zadaniem będzie wprowadzenie limitu czasowego polowania na muchy. Chcemy to tak zrealizować, aby gra kończyła się jednym z dwóch sposobów: albo gracz zdąży wybić wszystkie muchy, albo limit czasu zostanie przekroczony. W tym drugi przypadku trzeba będzie dopilnować, aby wszystkie pozostałe muchy zostały zniszczone.

Tworzymy funkcję obliczającą pozostały czas do końca gry (metoda klasy *FormGra*):

```
public void liczCzas ()
{
	int i;
	for ( i=czasGry; i>=0; i-- )
	{
		if ( liczbaMuch>0 )
		{
			Thread.Sleep(1000);
		}
		else break;
	}
	if ( i<0 )
	{
		watekMuch.Interrupt();
		liczbaMuch=0;
		MessageBox.Show("Nie potrafisz jeszcze trzepac wspolbieznie
uciekajacych much. :P\n","Wyniki lapania much");
	}
	else
```

```
	{
		MessageBox.Show("Gratulacje!!! Udalo Ci sie wytrzepac wszystkie
wspolbieznie uciekajace muchy!!!\n","Wyniki lapania much");
	}
	menuGra.Enabled = true;
}
```

Wątek wywołujący funkcję *liczCzas* w każdej iteracji usypiany jest na 1s. Jeżeli czas zostanie przekroczony, tj. i<0 wywołana zostanie funkcja *watekMuch.Interrupt()*, która przerywa wątek *watekMuch* znajdujący się w stanie *WaitSleepJoin* i rzuca wyjątek *ThreadInterruptedException*.

Zgodnie z założeniami przekroczenie czasu miało dopilnować zniszczenie wszystkich pozostałych wątków much. Aby to zrealizować musimy nieco zmodyfikować funkcję *start* klasy *MuchyThread* zarządzającej wszystkimi muchami.

```
public void start ()
{
	try
	{
		(...)// wszystko to co bylo do tej pory

		IDictionaryEnumerator ie = watkiMuch.GetEnumerator();
		while ( ie.MoveNext() )
		{
			((Thread)ie.Value).Join();
		}

	}
	catch ( ThreadInterruptedException )
	{
		IDictionaryEnumerator ie = watkiMuch.GetEnumerator();
		while ( ie.MoveNext() )
		{
			((Thread)ie.Value).Abort();
		}
		for ( int i=muchy.Count-1; i>=0; i-- )
		{
			muchy.RemoveAt(i);
		}
		form.Invalidate();
	}
}
```

Wątek wykonujący metodę *start* klasy *MuchyThread* bardzo często będzie znajdował się w stanie *WaitSleepJoin*. Będzie miało to miejsce albo w momencie wypuszczania much i blokadzie na opuszczaniu semafora dzięki *Wait* albo też po wypuszczeniu wszystkich much trafiając do pętli *while* gdzie blokowany będzie dzięki *Join*. Wywołanie funkcji *[watek].Join()* spowoduje zablokowanie wątku ją wywołującego do momentu aż *[watek]* zostanie zakończony. Jeśli nasz wątek realizujący *start* przejdzie przez wszystkie możliwe blokady i zakończy się samodzielnie będzie to oznaczać sytuację, w której gracz zdążył wybić wszystkie muchy w określonym czasie. Jeżeli jednak wątek ten zostanie przerwany z powodu przekroczonego czasu, to będzie on musiał obsłużyć rzucony wyjątek *ThreadInterruptedException*. Jak widać w sekcji *catch* wątek ten najpierw przerwie wszystkie wątki much, a następnie usunie wszystkie muchy z kolekcji *muchy*.

Ostatecznie musimy wprowadzić jeszcze trzy linijki na koniec metody *menuGraStart_Click* klasy *FormGra* i możemy nacieszyć nasze oczy w pełni wielowątkową grą:

```
Thread czas = new Thread ( new ThreadStart( liczCzas ));
czas.IsBackground = true;
```

```
czas.Start();
```

Na zakończenie

Jak wspomniałem na samym początku artykuł miał być jedynie wprowadzeniem w tworzenie programów wielowątkowych. Problem współbieżności, szczególnie w działaniu z zasobami współdzielonymi jest o wiele bardziej złożony. Początkujący programista może napotkać wiele niemiłych niespodzianek na swojej drodze związanych z zakleszczaniem wątków czy nieodpowiedniej synchronizacji operacji na wspólnych danych. Nie warto się jednak zrażać, ponieważ wielowątkowość jest i będzie nieodłączną cechą wielu aplikacji i może się okazać wielce przydatna przy projektowaniu własnego oprogramowania.