

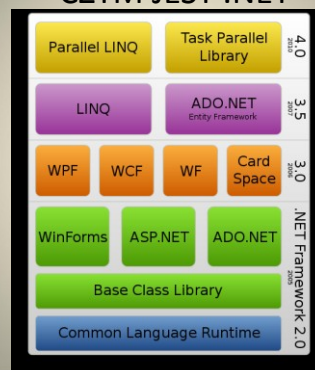
CZYM JEST .NET

- Konkurent technologii Java VM
- Technologia maszyny wirtualnej
- Środowisko opisane standardami
- Niezależne sprzętowo
- Obiektowy sprzęg do systemu/aplikacji
- Można rozszerzać dodając nowe podsystemy
- Dostępne są różne języki tworzenia aplikacji
- Dostępni są różni producenci środowisk, także *OPEN SOURCE*: **MONO**

CZYM JEST .NET

- Technologia maszyny wirtualnej pozwala:
 - Tworzyć aplikacje samodzielne, przenośne między różnymi platformami sprzętowymi i OS (*exe, dll, ...*),
 - Tworzyć aplikacje systemowe (*usługi, exe*),
 - Tworzyć komponenty (i tzw. wtyczki) – *dll, ...*
 - Tworzyć aplikacje webowe (*ASP.NET*),
 - Tworzyć usługi sieciowe (*web services*),
 -

CZYM JEST .NET



Źródło: MSDN

Common Language Infrastructure

Otwarta specyfikacja, standaryzowana przez ISO i ECMA

SKŁADOWE:

- Common Type System (CTS) - standaryzacja danych
- Common Language Specification (CLS) - wspólny język CIL (Common Intermediate Language)
- Virtual Execution System (VES) – wykonanie kodu, kontrola sterowania, obsługa wyjątków

Języki kompilowalne do CLI (.NET)

Wśród kilkudziesięciu języków:

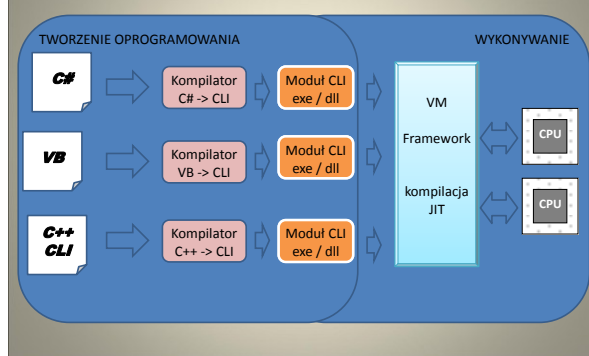
MS:

- C/C++ dla CLI,
- C# - główny język,
- F# - wzorowany na OCaml, do opisu funkcji,
- VB,
- J# - Java, projekt zarzucony,

inne:

- A# (Ada),
- Fortran .NET,
- L# - Lisp,
- P# - Prolog,
- IronPython,
- IronRuby

TECHNOLOGIA MASZINY WIRTUALNEJ

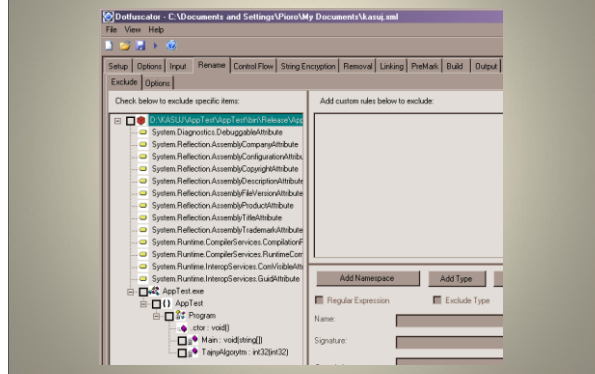


Asemlacja, deasemlacja i obfuskacja

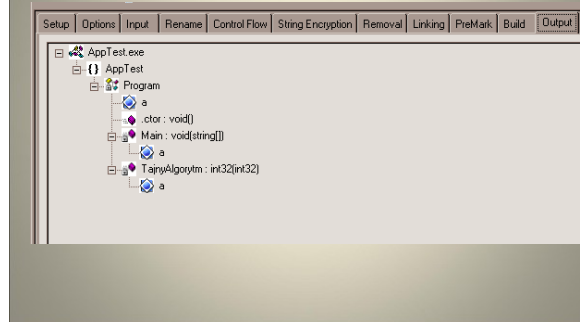
```
using System;
using System.Collections.Generic;
using System.Text;

namespace AppTest
{
    class Program
    {
        static int TajnyAlgorytm(int wpłata)
        {
            int podatek = (int)((double)wpłata*0.23);
            return podatek;
        }
        static void Main(string[] args)
        {
            Console.WriteLine("Wpłata 1000, VAT:"+TajnyAlgorytm(1000) );
            Console.ReadKey();
        }
    }
}
```

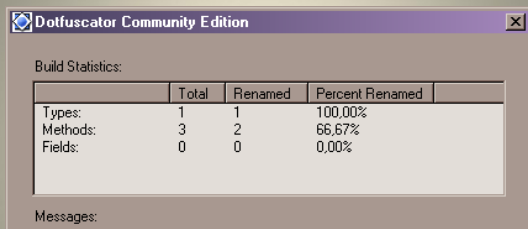

Asemlacja, deasemlacja i obfuskacja



Asemlacja, deasemlacja i obfuskacja

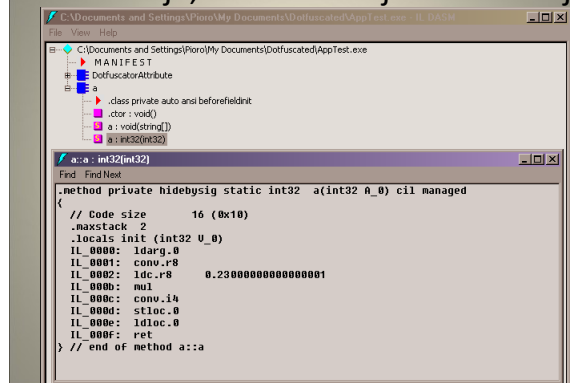


Asemlacja, deasemlacja i obfuskacja

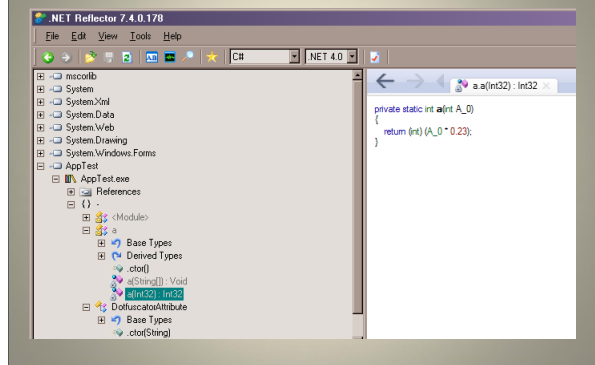


Dotfuscator – patent na indukcyjne nadawanie tej samej nazwy zmiennym, które są rozróżniane w oparciu o kontekst użycia,
 Inne obfuskatory:
 - zmieniają przebieg wykonania programu,
 - nadają nazwy identyfikatorom spośród słów kluczowych języka

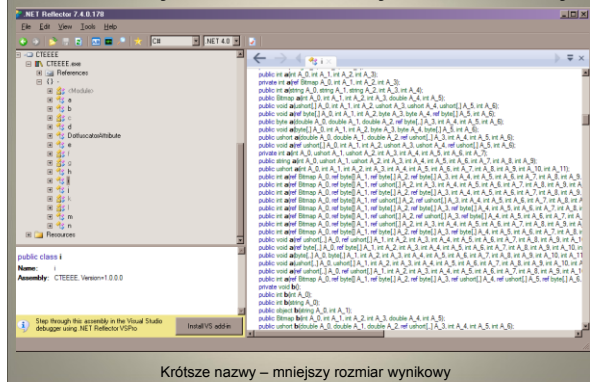
Asemlacja, deasemlacja i obfuskacja



Asemlacja, deasemlacja i obfuskacja



Asemlacja, deasemlacja i obfuskacja



Krótsze nazwy – mniejszy rozmiar wynikowy

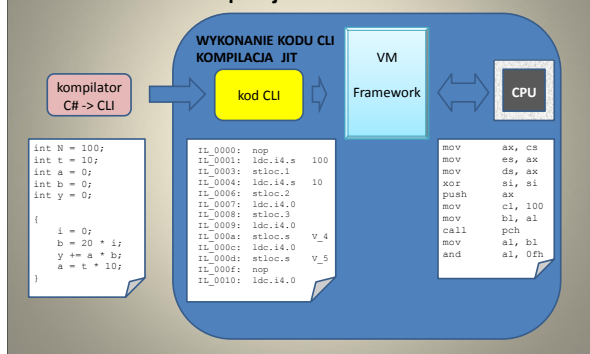
Asemlacja, deasemlacja i obfuskacja

```
public int a(ref Bitmap A_0, ref byte[] A_1, ref byte[] A_2, ref byte[] A_3, int A_4, int A_5, int A_6, int A_7, int A_8, int A_9, int
{
    int num3 = 0;
    int num4 = 0;
    try
    {
        int num;
        int num2;
        byte num6;
        if (A_11 == 0)
        {
            if (this.m == 0)
            {
                this.c(ref this, ref A_2, this.f, this.g, this.h, A_4, A_5, A_4 / 2, A_5, 0, (num3 - (A_4 / 2)) + this.a[A_10], num4 - (A_
            }
            else
            {
                this.c(ref this, ref A_2, this.f, this.g, this.h, A_4, A_5, A_4 / 2, A_5, 0, (num3 - (A_4 / 2)) + this.a[A_10], num4 - (A_5 - th
            }
        }
        else if (this.m == 0)
        {
            this.a(ref this, ref A_2, this.f, this.g, this.h, A_4, A_5, A_4 / 2, A_5, 0, (num3 - (A_4 / 2)) + this.a[A_10], num4 - (A_5
        }
        else
        {
            this.a(ref this, ref A_2, this.f, this.g, this.h, A_4, A_5, A_4 / 2, A_5, 0, (num3 - (A_4 / 2)) + this.a[A_10], num4 - (A_5 - this
        }
    }
    if (A_12 > 0 && this.p > 0)
    {
        if (this.m == 0)
        {

```

TECHNOLOGIA MASZYNY WIRTUALNEJ

Kompilacja Just-in-Time



TECHNOLOGIA MASZINY WIRTUALNEJ

Klasyczne optymalizacje kodu

MS: (Java VM : opcje -client i -server)

Optymalizacje wykonywane na etapie kompilacji z kodu źródłowego do CIL.

CSC /o (lub /optimize)

Proste optymalizacje, takie jak np. zwijanie stałych

MONO: <http://www.mono-project.com/AOT>

Liczne opcje optymalizacji, zarówno dotyczące kompilacji JIT, jak i AOT.

--aot	Peephole postpass	tailc	Tail recursion and tail calls
peephole	Branch optimizations	loop	Loop related optimizations
branch	Inline method calls	fcov	Fast x86 FP compares
inline	Constant folding	leaf	Leaf procedures optimizations
cfold	Constant propagation	aot	Usage of Ahead Of Time compiled code
constexpr	Copy propagation	precomp	Precompile all methods before executing
copyprop	Dead code elimination	abcrem	Array bound checks removal
deadcode	Linear scan global reg allocation	ssa	SSA based Partial Redundancy Elimination
linearsc	Conditional moves	exception	Optimize exception catch blocks
cmov	Emit per-domain code	ssa	Use plain SSA form
shared	Instruction scheduling	treeprop	Tree propagation
sched	Intrinsic method implementations	sse2	SSE2 instructions on x86
intrinsic		gshared	Share generics

TECHNOLOGIA MASZINY WIRTUALNEJ

Profile-guided optimization (PGO)

- optymalizacja w trakcie wykonania,
- przeprowadzana w wybranych momentach,
- na bieżąco kod jest dostosowywany/optymalizowany do środowiska wykonawczego,
- optymalizacja odbywa się w oparciu o heurystyki i statystyki pozyskane z uprzednio wykonanych części kodu,
- stałe obciążenie systemu sporządzaniem statystyk przyczynia się do zmniejszenia wydajności

(Java VM, .NET, Visual C++ !!! ☺, etc)

TECHNOLOGIA MASZINY WIRTUALNEJ

Kompilacja Just-in-Time a kompilacja Ahead-of-Time

JIT:

- optymalizacja dotyczy kodu pośredniego,
- optymalizowane są głównie pętle oraz fragmenty kodu, które są wykonywane najczęściej,
- kompilacja odbywa się etapami (krokowo),
- na bieżąco kod jest dostosowywany/optymalizowany do środowiska wykonawczego,
- optymalizacja odbywa się w oparciu o heurystyki i statystyki pozyskane z uprzednio wykonanych części kodu,
- pełna optymalizacja może nie być możliwa ze względu na czasochłonność

AOT: (MS: ngen install testy.exe, MONO: mono --aot testy.exe)

- technika stosowana do kodu pośredniego,
- optymalizacja dla danego środowiska wykonawczego,
- możliwość dokonania pełnej, czasochłonnej optymalizacji, niewykonalnej dla JIT
- wykonanie kodu w środowisku docelowym może być szybsze, ALE NIE MUSI,
- czasem JIT widząc kompilację przez NGen nie wykonuje wszystkich sprawdzeń,
- wykonanie kodu w innych środowiskach niż docelowe może być dużo wolniejsze,
- często nie wykłucza optymalizacji JIT – np. w przypadku uruchomienia w innym środowisku lub w przypadku pewnych optymalizacji realizowanych tylko przez JIT

TECHNOLOGIA MASZINY WIRTUALNEJ

Metody optymalizacji prędkości wykonania kodu

Ogólne:

- rozwijanie funkcji (inline),
- globalna alokacja rejestrów,
- usuwanie martwego kodu,
- redukcja mocy,
- łączenie operacji,
- zwijanie stałych,
- propagacja stałych,
- upraszczanie wyrażeń,
- usuwanie podwyrażeń wspólnych,
- propagacja kopii

Pętle:

- wektoryzacja pętli,
- zwijanie pętli,
- rozwijanie pętli,
- eliminacja zmiennych indukcyjnych,
- przeniesienie kodu poza pętlę

Więcej: wykład Kompilacja i Kompilatory

TECHNOLOGIA MASZINY WIRTUALNEJ

SSA – Single Static Assignment

Forma kodu zakładająca pojedyncze przypisanie do danej zmiennej, transformacja do postaci SSA polega na dodaniu nowych zmiennych

wejście	wersja SSA
a = 1;	a1 = 1;
b = 2;	b1 = 2;
a = b + 3;	a2 = b1 + 3;

Dla wersji SSA można dokonać analizy w postaci grafu przepływów sterowania i którego to grafu redukcja pozwala na dokonywanie takich operacji jak np. usuwanie martwego kodu, usuwanie podwyrażeń wspólnych, propagacja stałych etc.

Usuwanie martwego kodu – usunięcie przypisań do zmiennych, które nie są używane.

TECHNOLOGIA MASZINY WIRTUALNEJ

Przetwarzanie potokowe a prędkość kodu

(=> predykcja odgałęzień - branch)

```

0: iload_1
1: iload_2
2: if_icmpge 7
5: iconst_m1
6: ireturn
7: iload_1
8: iload_2
9: if_icmple 14
12: iconst_1
13: ireturn
14: iconst_0
15: ireturn

```

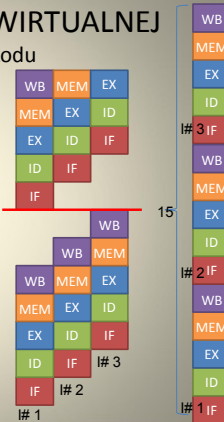
Problemy:

- instrukcje skoków warunkowych
- wywołania funkcji (stos)

=> branch optimization

RISC:

- instruction fetch
- instruction decode
- execute
- memory access
- write back



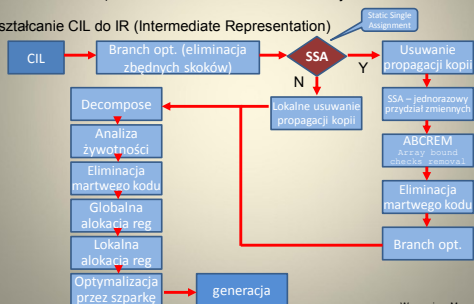
TECHNOLOGIA MASZINY WIRTUALNEJ

Optymalizacja podczas kompilacji Just-in-Time

MS .NET, Java VM ? ;)

Mono => dokumentacja :D

Przekształcanie CIL do IR (Intermediate Representation)



W oparciu o Mono summit 2006

TECHNOLOGIA MASZINY WIRTUALNEJ

Optymalizacja podczas kompilacji Just-in-Time

Etapy kompilacji JIT:

- CIL -> IR
- Decompose-Long-Opts (dekompozycja zmiennych dłuższych niż 32 bity @CPU 32)
- Local Copy/Constant Propagation
- Branch Optimizations
- Handle-Global-Vregs
- Local Dead Code Elimination
- Decompose VType Opts
- SSA Optimizations
- Liveness Analysis
- Global Register Allocation
- Allocate Vars
- Spill Global Vars

http://www.mono-project.com/Linear_IL

TECHNOLOGIA MASZINY WIRTUALNEJ

Optymalizacja podczas kompilacji **Just-in-Time**

Optymalizacja przez szparkę odbywa się dla ograniczonego bloku kodu i obejmuje:

- zwijanie stałych,
- redukcję mocy (instrukcji),
- usuwanie martwego kodu (operacji bezproduktywnych),
- łączenia operacji – jedna operacja w miejsce kilku równorzędnych,
- upraszczanie wyrażeń w oparciu o prawa algebraiczne,
- wykorzystanie specjalnych instrukcji procesora,
- dobór metod adresowania (upraszczanie odniesień do pamięci)

TECHNOLOGIA MASZINY WIRTUALNEJ

JVM

Java HotSpot Client Compiler

The Java HotSpot Client Compiler is a simple, fast three-phase compiler. In the first phase, a platform-independent front end constructs a high-level intermediate representation (HIR) from the bytecodes. The HIR uses static single assignment (SSA) form to represent values in order to more easily enable certain optimizations, which are performed during and after IR construction. In the second phase, the platform-specific back end generates a low-level intermediate representation (LIR) from the HIR. The final phase performs register allocation on the LIR using a customized version of the linear scan algorithm, does peephole optimization on the LIR and generates machine code from it.

Emphasis is placed on extracting and preserving as much information as possible from the bytecodes. The client compiler focuses on local code quality and does very few global optimizations, since those are often the most expensive in terms of compile time.

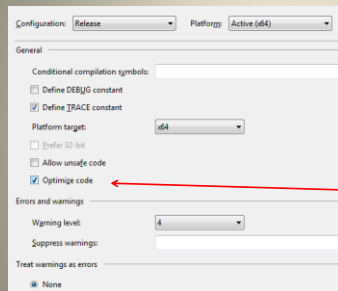
Java HotSpot Server Compiler

The server compiler is tuned for the performance profile of typical server applications. The Java HotSpot Server Compiler is a high-end fully optimizing compiler. It uses an advanced static single assignment (SSA)-based IR for optimizations. The optimizer performs all the classic optimizations, including dead code elimination, loop invariant hoisting, common subexpression elimination, constant propagation, global value numbering, and global code motion. It also features optimizations more specific to Java technology, such as null-check and range-check elimination and optimization of exception throwing paths. The register allocator is a global graph coloring allocator and makes full use of large register sets that are commonly found in RISC microprocessors. The compiler is highly portable, relying on a machine description file to describe all aspects of the target hardware. While the compiler is slow by JIT standards, it is still much faster than conventional optimizing compilers, and the improved code quality pays back the compile time by reducing execution times for compiled code.

<http://www.oracle.com/technetwork/java/whitepaper-135217.html#client>

TECHNOLOGIA MASZINY WIRTUALNEJ

.NET – optymalizacja na etapie kompilacji do IL



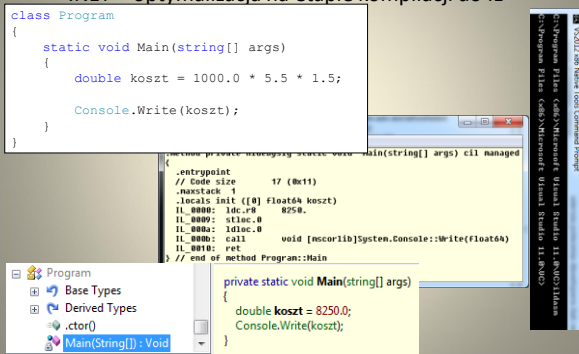
*constant folding,
copy propagation,
dead code elimination**

C:\Windows\Microsoft.NET\Framework\v2.0.50727>csc /o test.cs

* <http://blogs.msdn.com/b/jmstall/archive/2006/03/13/dead-code-elimination.aspx>

TECHNOLOGIA MASZINY WIRTUALNEJ

.NET – optymalizacja na etapie kompilacji do IL



TECHNOLOGIA MASZINY WIRTUALNEJ

.NET – optymalizacja na etapie kompilacji do IL

```
byte rgb_r, rgb_g, rgb_b;
//byte rgb_r = 0, rgb_g = 0, rgb_b = 0;

for (int i = 0; i < sx; i++)
    for (int j = 0; j < sy; j++)
    {
        rgb_r = R[i, j];
        rgb_g = G[i, j];
        rgb_b = B[i, j];
    }
```

```
for (int i = 0; i < sx; i++)
{
    for (int j = 0; j < sy; j++)
    {
        byte rgb_r = R[i, j];
        byte rgb_g = G[i, j];
        byte rgb_b = B[i, j];
    }
}
```

```
//byte rgb_r, rgb_g, rgb_b;
byte rgb_r = 0, rgb_g = 0, rgb_b = 0;

for (int i = 0; i < sx; i++)
    for (int j = 0; j < sy; j++)
    {
        rgb_r = R[i, j];
        rgb_g = G[i, j];
        rgb_b = B[i, j];
    }
```

```
byte rgb_r = 0;
byte rgb_g = 0;
byte rgb_b = 0;

for (int i = 0; i < sx; i++)
{
    for (int j = 0; j < sy; j++)
    {
        rgb_r = R[i, j];
        rgb_g = G[i, j];
        rgb_b = B[i, j];
    }
}
```

TECHNOLOGIA MASZINY WIRTUALNEJ

.NET – optymalizacja na etapie wykonania przez JIT

The following is a list of the profile-guided optimizations:

- **Inlining** – For example, if there exists a function A that frequently calls function B, and function B is relatively small, then profile-guided optimizations will inline function B in function A.
- **Virtual Call Speculation** – If a virtual call, or other call through a function pointer, frequently targets a certain function, a profile-guided optimization can insert a conditionally-executed direct call to the frequently-targeted function, and the direct call can be inlined.
- **Register Allocation** – Optimizing with profile data results in better register allocation.
- **Basic Block Optimization** – Basic block optimization allows commonly executed basic blocks that temporally execute within a given frame to be placed in the same set of pages (locality). This minimizes the number of pages used, thus minimizing memory overhead.
- **Size/Speed Optimization** – Functions where the program spends a lot of time can be optimized for speed.
- **Function Layout** – Based on the call graph and profiled caller/callee behavior, functions that tend to be along the same execution path are placed in the same section.
- **Conditional Branch Optimization** – With the value probes, profile-guided optimizations can find if a given value in a switch statement is used more often than other values. This value can then be pulled out of the switch statement. The same can be done with if/else instructions where the optimizer can order the if/else so that either the if or else block is placed first depending on which block is more frequently true.
- **Dead Code Separation** – Code that is not called during profiling is moved to a special section that is appended to the end of the set of sections. This effectively keeps this section out of the often-used pages.
- **EH Code Separation** – The EH code, being exceptionally executed, can often be moved to a separate section when profile-guided optimizations can determine that the exceptions occur only on exceptional conditions.
- **Memory Intrinsics** – The expansion of intrinsics can be decided better if it can be determined if an intrinsic is called frequently. An intrinsic can also be optimized based on the block size of moves or copies.

<https://msdn.microsoft.com/en-us/library/7k32f4k.aspx>

TECHNOLOGIA MASZINY WIRTUALNEJ

.NET – sterowanie ręczne optymalizacją na etapie wykonania przez JIT

`RuntimeHelpers.PrepareMethod()`

przygotowanie funkcji - wywołanie optymalizacji przez JIT
Constrained Execution Region

```
using System.Runtime.CompilerServices;

[MethodImplAttribute(MethodImplOptions.NoInlining)]
public void duza_petla_bez_inline()
{
    ...
}
```

`MethodImplOptions:`

AggressiveInlining	dana metoda ma być inline, jeśli to tylko możliwe
NoInlining	dana metoda nie może być inline
NoOptimization	metoda nie jest optymalizowana przez JIT ani NGEN - ze względu na debugowanie
Synchronized	tylko jeden wątek naraz może wykonywać metodę
Unmanaged	dla kodu niezarządzanego

TECHNOLOGIA MASZINY WIRTUALNEJ

.NET – sterowanie ręczne optymalizacją na etapie wykonania przez JIT

Uruchomione w pętli 10 mln razy:

```
[MethodImpl(MethodImplOptions.NoInlining)]
static int podatek(int p)
{
    int a = (p + 2) * (p + 2);
    p = a / 2;
    p--;
    return p * 10;
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
static int podatekinline(int p)
{
    int a = (p + 2) * (p + 2);
    p = a / 2;
    p--;
    return p * 10;
}
```

```
bez inline: 18 ms
bez inline: 18 ms
bez inline: 18 ms
bez inline: 18 ms
bez inline: 18 ms
bez inline: 18 ms
bez inline: 18 ms
bez inline: 18 ms
bez inline: 18 ms
z inline : 15 ms
z inline : 15 ms
z inline : 15 ms
z inline : 15 ms
z inline : 15 ms
z inline : 15 ms
z inline : 15 ms
z inline : 15 ms
z inline : 15 ms
z inline : 15 ms
```

Stosowanie `inline` uzależnione jest od rozmiaru funkcji, np. ograniczenie do 32 instrukcji CLR, bez try/catch, pętli, MarshalByRef, metody wirtualne etc.

W temacie....

1. Piórkowski, A., Żupnik, M.: Loop optimization in managed code environments with expressions evaluated only once. TASK Quarterly, Vol. 14, No. 4., 2010, s. 397-404.
2. Kras, S., Piórkowski, A.: Optymalizacja kodu przy wykorzystaniu algorytmu SSA w środowiskach maszyn wirtualnych. Studia Informatica, 37(1), 2016, s. 97-111.
3. Olshevskyy Y., Piórkowski A.: Oszacowanie wydajności optymalizacji przy wykorzystaniu liniowej reprezentacji pośredniej kodu. Studia Informatica, 2018 ...
4. Aleksander R., Piórkowski A.: Dekompozycja instrukcji rozgałęziających w procesie optymalizacji kodu. ?

KOD ZARZĄDZANY

Platforma wykonująca kod zarządzany nadzoruje:

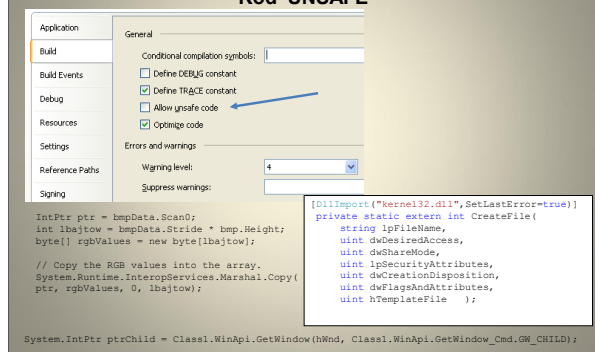
- sterowanie wykonaniem programu,
- obsługę wyjątków,
- zarządzanie pamięcią.

Cechy charakterystyczne

- brak wskaźników,
- położeniem obiektów w pamięci zarządza platforma – może je przesuwać,
- platforma przydziela miejsce w pamięci dla obiektów,
- obiekty nie są natychmiast usuwane, gdy przestają być używane,
- platforma zlicza referencje do obiektów,
- zarządza odzyskiwaniem pamięci

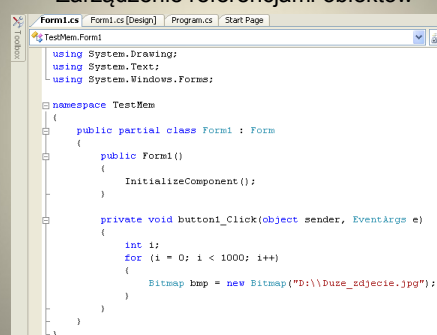
KOD ZARZĄDZANY

Kod UNSAFE



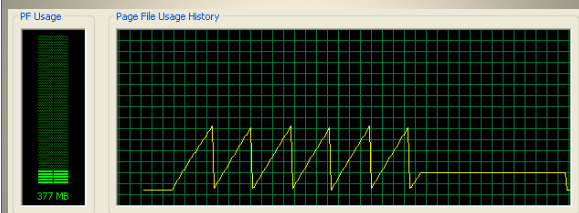
KOD ZARZĄDZANY

Zarządzanie referencjami obiektów



KOD ZARZĄDZANY

Zarządzanie referencjami obiektów



Max: 1,6 GB

KOD ZARZĄDZANY

Fragmentacja zewnętrzna

1



2

Obszar scalony zajmuje mniej stron pamięci => mniej pamięci L2 => szybszy dostęp

3

Wycieki pamięci – typowe dla kadry niskokwalifikowanej ;)

KOD ZARZĄDZANY

Garbage Collector

Charakterystyka:

- zarządzanie dużymi obiektami,
- Każdy proces ma przydzieloną swoją przestrzeń adresową (user) na dane (2GB),
- rozwiązanie problemu fragmentacji zewnętrznej w obszarze adresowym użytkownika – scalanie i zsuwanie obszarów w jeden blok aby móc udostępnić jak największy blok,
- programista w systemie operacyjnym zarządza bezpośrednio blokami pamięci,
- platforma NET przydziela bloki pamięci (obiekty) na żądanie, zwalnia je jednak zbiorczo raz na jakiś czas (zwiększa to wydajność) gdy:
 - jest mało dostępnej pamięci fizycznej,
 - zaalokowany obszar zbliża się do progu (próg ten jest ustalany na bieżąco),
 - wywołano funkcję odzyskującą miejsce w pamięci,
- częstotliwość i czas trwania odświeżania zależy od rozmiaru zajętości i wolnego miejsca

KOD ZARZĄDZANY

Garbage Collector

Sterta przeznaczona na obiekty dzieli się na dwie części:

- obiekty duże (powyżej 84KB, tablice),
- obiekty małe.

ZASADY USUWANIA OBIEKTÓW:

- obiekty, które nie posiadają żadnego odwołania – nadają się do usunięcia,
- zmienne automatyczne i zmienne na stosie – należy sprawdzić, czy są potrzebne,
- Etap 1 – znalezienie wszystkich 'żywych' obiektów (mających aktualne odwołania),
- Etap 2 – wyznaczenie nowych adresów obiektów,
- Etap 3 – przenoszenie obiektów (kompaktowanie)

Zliczanie referencji – problem dwóch obiektów niedostępnych, referujących się wzajemnie

- GC ma strategię, która ma zapewnić:

- utrzymać zestaw roboczy pamięci na odpowiednim (nie za dużym) poziomie,
- pilnować, by czyszczenie nie trwało zbyt długo.

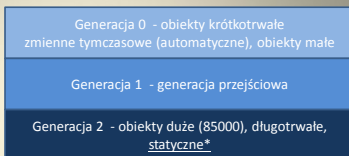
- kompaktowanie dużych obiektów może być kosztowne (czas i pamięć)

KOD ZARZĄDZANY

Garbage Collector

Idee:

- im dłużej obiekt jest na sterckie, tym prawdopodobnie dłużej będzie używany
- małe zmienne lokalne są używane przez krótki czas



- obiekty, które przetrwały czyszczenie – przechodzą do następnej generacji,
- gdy GC wykryje duży udział obiektów G2 – zwiększa próg alokacji dla generacji,

Przesuwanie dużych obiektów (w Generacji 2) jest kosztowne.

Pełne odświeżanie – proces GC obejmujący wszystkie generacje.
Częściowe odświeżanie – usunięcie obiektów.

* Obiekty statyczne – mogą być używane, choć nie mają aktywnych referencji

KOD ZARZĄDZANY

Garbage Collector

Generacje 0 i 1 – generacje efemeryczne (dla obiektów krótkotrwałych)

ich rozmiar:

	32 bit	64 bit
Workstation GC	16MB	256MB
Server GC	64MB	4GB
Server GC, 4+CPU (log)	32MB	2GB
Server GC, 8+CPU (log)	16MB	1GB

Generacja 2 może się składać z wielu segmentów – kompaktowanie dużych obiektów jest czasochłonne

KOD ZARZĄDZANY

Garbage Collector – tryby stacja robocza i serwer

Stacja robocza (Workstation):

- dwa tryby – współbieżny lub wyłączny,
- tryb domyślny dla maszyny z 1 CPU - w przypadku włączenia trybu serwer zostaje wyłączona współbieżność,
- wątki zarządzane zazwyczaj mają priorytet normalny, wątki GC mają też taki priorytet i współzawodniczą w pozyskaniu procesora,
- model dla licznych procesów,
- wątki kodu natywnego nie są zatrzymywane,

Serwer:

- dwa tryby – wyłączny lub w tle,
- priorytety wątków GC są najwyższe (15),
- każdy obszar pamięci (proces) ma swoje wątki GC do każdego CPU – dużo!
- model dla niewielkiej liczby procesów o dużym użyciu pamięci.

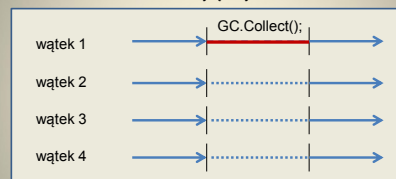
Od .NET 4.0 tryb współbieżny zostaje zastąpiony odświeżaniem w tle.

Przełączanie między Workstation a Server do sprawdzenia: `GCSettings.IsServerGC`

KOD ZARZĄDZANY

Garbage Collector – wywołanie odświeżania

model wyłączny



Na czas odświeżania pozostałe wątki są zawieszane, aby nie odwoływać się do zmiennych w trakcie relokacji

KOD ZARZĄDZANY

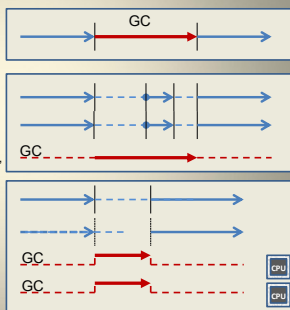
Garbage Collector – współbieżny GC

Workstation - model wyłączny
- działa dla G0 i G1

Model współbieżny

- zaimplementowany w osobnym wątku,
- blokuje na krócej ale częściej,
- dla aplikacji interaktywnych,
- pozwala alokować obiekty w trakcie działania,
- zużywa więcej zasobów CPU i RAM,
- działa dla G2 (G0 i G1 odświeża się szybko)

Model serwer
- szybki GC



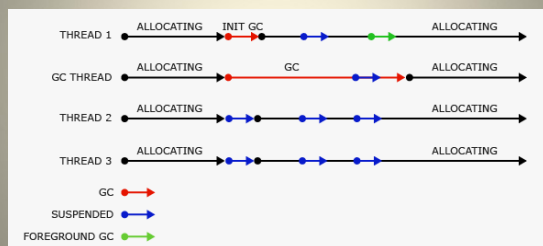
KOD ZARZĄDZANY

Garbage Collector – odświeżanie w tle

- zastępuje model współbieżny,
- odświeżanie w tle dla trybu workstation – od .NET 4.0,
- odświeżanie w tle dla trybu serwer – od .NET 4.5,
- działa na dedykowanym wątku (podobnie jak odświeżanie współbieżne),
- dotyczy generacji G2 (podobnie jak odświeżanie współbieżne),
- generacje efemeryczne odświeżane są w trybie wyłącznym,
- odświeżanie G0 i G1 uruchamiane jako pierwszoplanowe,
- uruchamiane w razie potrzeby lub: wątek GC sprawdza, kiedy uruchomić odświeżanie pierwszoplanowe,
- workstation – 1 wątek, serwer – N wątków (x I. CPU)

KOD ZARZĄDZANY

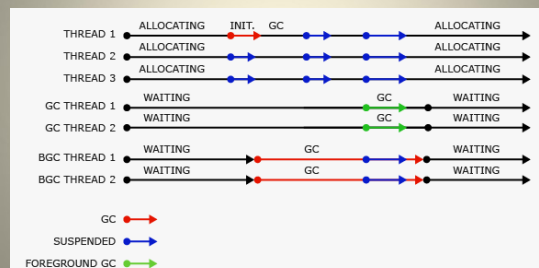
Garbage Collector – odświeżanie w tle tryb workstation, .NET 4.0



[https://msdn.microsoft.com/en-us/library/ee787088\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx)

KOD ZARZĄDZANY

Garbage Collector – odświeżanie w tle tryb serwer, .NET 4.5



[https://msdn.microsoft.com/en-us/library/ee787088\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx)

KOD ZARZĄDZANY

Garbage Collector – zarządzanie GC

Funkcje GC:

- `GC.KeepAlive()`
- `GC.Collect()` – wymuszenie pracy GC (odświeżenie) – np. przed utworzeniem dużych, wymagających zasobów
- `GetGeneration()` – podaje generację danego obiektu,
- `CollectionCount()` – liczba wywołań GC,
- `SupressFinalize()` – pominięcie 'finalizacji'
- `MaxGeneration()` – maksymalny poziom generacji w systemie,
- `AddMemoryPressure()/RemoveMemoryPressure()` – ustawianie poziomu pilności odświeżania

KOD ZARZĄDZANY

Zarządzanie czasem życia obiektów

Platforma sama zarządza czasem życia obiektów. W przypadku kodu zarządzanego (safe) programista właściwie nie potrzebuje zajmować się usuwaniem obiektów.

Inaczej jest w przypadku zasobów zewnętrznych na różnym poziomie, np:

- połączeń do kodu niezarządzanego (przykład `CreateFile()`, użycie `PInvoke()`)
- uchwytów systemowych, dostępu do pamięci niezarządzanej,
- połączeń z innymi systemami, np. bazodanowymi (pomijając ADO),
-

Przykład: konieczne jest resetowanie dostępu do urządzenia przy wyłączaniu programu (inaczej następna instancja nie będzie mogła podłączyć się)

Obiekty dzielą się na dwa rodzaje – typy **wartościowe** i typy **referencyjne**:

- typy wartościowe dziedziczą niejawnie po typie `ValueType`, są to typy liczbowe, ..., aż po struktury (grupa wartości); mogą być alokowane na stosie lub sterpie,
- typy referencyjne dziedziczą po typie `object`, odwołania do nich mają charakter referencyjny (np. kopiowanie obiektów klas powoduje kopiowanie referencji do składowych tablic; alokowane są na sterpie zarządzanej przez GC).

KOD ZARZĄDZANY

Zarządzanie czasem życia obiektów

```
class MojaKlasa
{
    public int a = 0;
}

static void Main(string[] args)
{
    MojaKlasa mk1 = new MojaKlasa();
    MojaKlasa mkr = null;

    {
        MojaKlasa mk2 = new MojaKlasa();
        MojaKlasa mk3 = new MojaKlasa();
        mkr = mk2;
    }

    GC.Collect();

    MojaKlasa.FunkcjaPrzekazujaca(ref mk1);
    Console.WriteLine(mkr.a);
}
```

root references

Korzenie aplikacji to referencje do:

- obiektów globalnych (w innych językach niż C#)
- obiektów statycznych (lub pól statycznych)
- obiektów lokalnych w kodzie bazowym apl.,
- parametrów przesłanych do metody,
- obiektów oczekujących na finalizację oraz powiązania rejestrów CPU odwołujących się do obiektu

Graf obiektów podczas odświeżania



Każdy obiekt występuje tylko raz w grafie aby nie było cykli, czyli sytuacji, w której dwa obiekty wskazują na siebie wzajemnie, ale nie są już dostępne.

KOD ZARZĄDZANY

Zarządzanie czasem życia obiektów

- [C#] net 2.0 – destruktor, net 3.0+ - finalizator `~Object();`

Usuwanie referencji – np. przypisanie null

- finalizator – dla obiektów niezarządzalnych – 'free()',

`public class Object`

```
{
    ...
    protected virtual void Finalize() {}
    Finalizator krytyczny – dziedziczący po CriticalFinalizerObject,
```

- obiekty jednorazowe: `IDisposable / using`

```
public interface IDisposable
{
    void Dispose(); // poinformowanie, CLR, że obiekt jest już do usunięcia
}
```

- obiekty o słabych referencjach – krótkie i długie

(silne referencje to standardowo tworzone obiekty)

`WeakReference<>, Lazy<>` (leniwe tworzenie instancji obiektów)

KOD ZARZĄDZANY

Destruktor / Finalizator

```
public class Object
{
    ...
    protected virtual void Finalize()
    {
    }
}
```

- finalizator mimo funkcji finalize jest definiowany jako destruktory klasy;
- ma on zastosowanie dla klas (alokowanych na stercie), struktury nie są typami referencyjnymi, zatem nie mogą posiadać finalizatorów,
- GC posiada osobną kolejkę finalizacji – obsługa takich obiektów jest dodatkowym obciążeniem,
- kiedy finalizator zostanie uruchomiony – nie wiadomo,
- finalizacja obiektu może być anulowana - GC.SuppressFinalize() (jeśli np. jest użyte Dispose())

KOD ZARZĄDZANY

Destruktor / Finalizator

```
class MojaKlasa
{
    public int a = 0;
    int[] duza = new int[100000]; // obciążenie

    public MojaKlasa(int w)
    {
        a=w;
        duza[0] = 0; // wymuszenie utworzenia tablicy
        Console.WriteLine(" Konstruktor obiektu nr: " + a
            + " generacja: " + GC.GetGeneration(this)
            + " cykli: " + GC.CollectionCount(2));
    }

    ~MojaKlasa() // destruktory / finalizatory
    {
        Console.WriteLine(" Destruktor obiektu nr: " + a
            + " generacja: " + GC.GetGeneration(this)
            + " cykli: " + GC.CollectionCount(2));
    }
}
```

KOD ZARZĄDZANY

```
MojaKlasa mk1 = new MojaKlasa(1);
{
    MojaKlasa mk2 = new MojaKlasa(2);
}
Console.WriteLine("obiekty utworzone");

{
    MojaKlasa mk3 = null;
    for (int i = 10; i < 15; i++)
        mk3 = new MojaKlasa(i);
}
Console.WriteLine("grupa obiektów utworzona");

Console.WriteLine("wywołanie GC Collect");
GC.Collect();
Console.WriteLine("po wywołaniu GC Collect");

GC.WaitForPendingFinalizers();

Console.WriteLine("po wywołaniu GC WaitForPendingFinalizers");

GC.WaitForFullGCComplete();

Console.WriteLine("po wywołaniu GC WaitForFullGCComplete");

//Console.WriteLine("Wartosc: " + mk1.a); //
```

KOD ZARZĄDZANY

Destruktor / Finalizator

```
Konstruktor obiektu nr: 1 generacja: 0 cykli: 0
Konstruktor obiektu nr: 2 generacja: 0 cykli: 0
obiekty utworzone
Konstruktor obiektu nr: 10 generacja: 0 cykli: 0
Konstruktor obiektu nr: 11 generacja: 0 cykli: 0
Konstruktor obiektu nr: 12 generacja: 0 cykli: 0
Konstruktor obiektu nr: 13 generacja: 0 cykli: 0
Konstruktor obiektu nr: 14 generacja: 0 cykli: 0
grupa obiektów utworzona
wywołanie GC Collect
po wywołaniu GC Collect
Destruktor obiektu nr: 14 generacja: 1 cykli: 1
Destruktor obiektu nr: 1 generacja: 1 cykli: 1
Destruktor obiektu nr: 13 generacja: 1 cykli: 1
Destruktor obiektu nr: 12 generacja: 1 cykli: 1
Destruktor obiektu nr: 11 generacja: 1 cykli: 1
Destruktor obiektu nr: 10 generacja: 1 cykli: 1
Destruktor obiektu nr: 2 generacja: 1 cykli: 1
po wywołaniu GC WaitForPendingFinalizers
po wywołaniu GC WaitForFullGCComplete
```

5 obiektów bez późnego użycia ob 1

KOD ZARZĄDZANY

```

Konstruktor obiektu nr: 1 generacja: 0 cykli: 0
Konstruktor obiektu nr: 2 generacja: 0 cykli: 0
obiekty utworzone
Konstruktor obiektu nr: 10 generacja: 0 cykli: 0
Konstruktor obiektu nr: 11 generacja: 0 cykli: 0
Konstruktor obiektu nr: 12 generacja: 0 cykli: 0
Konstruktor obiektu nr: 13 generacja: 0 cykli: 0
Konstruktor obiektu nr: 14 generacja: 0 cykli: 0
grupa obiektów utworzona
wywołanie GC Collect
po wywołaniu GC Collect
Destruktor obiektu nr: 14 generacja: 1 cykli: 1
Destruktor obiektu nr: 2 generacja: 1 cykli: 1
Destruktor obiektu nr: 13 generacja: 1 cykli: 1
Destruktor obiektu nr: 12 generacja: 1 cykli: 1
Destruktor obiektu nr: 11 generacja: 1 cykli: 1
Destruktor obiektu nr: 10 generacja: 1 cykli: 1
po wywołaniu GC WaitForPendingFinalizers
po wywołaniu GC WaitForFullGCCollect
Wartosc: 1
Destruktor obiektu nr: 1 generacja: 1 cykli: 1

```

5 obiektów z późnym użyciem ob 1

KOD ZARZĄDZANY

15 obiektów ...

```

Konstruktor obiektu nr: 1 generacja: 0 cykli: 0
Konstruktor obiektu nr: 2 generacja: 0 cykli: 0
obiekty utworzone
Konstruktor obiektu nr: 10 generacja: 0 cykli: 0
Konstruktor obiektu nr: 11 generacja: 0 cykli: 0
Konstruktor obiektu nr: 12 generacja: 0 cykli: 0
Konstruktor obiektu nr: 13 generacja: 0 cykli: 0
Konstruktor obiektu nr: 14 generacja: 0 cykli: 0
Konstruktor obiektu nr: 15 generacja: 0 cykli: 0
Konstruktor obiektu nr: 16 generacja: 1 cykli: 1
Konstruktor obiektu nr: 17 generacja: 0 cykli: 1
Konstruktor obiektu nr: 18 generacja: 0 cykli: 1
Konstruktor obiektu nr: 19 generacja: 0 cykli: 1
Konstruktor obiektu nr: 20 generacja: 0 cykli: 1
Konstruktor obiektu nr: 21 generacja: 0 cykli: 1
Konstruktor obiektu nr: 22 generacja: 0 cykli: 1
Konstruktor obiektu nr: 23 generacja: 0 cykli: 1
Destruktor obiektu nr: 15 generacja: 1 cykli: 1
Destruktor obiektu nr: 11 generacja: 2 cykli: 2
Destruktor obiektu nr: 10 generacja: 2 cykli: 2
Destruktor obiektu nr: 2 generacja: 2 cykli: 2
Konstruktor obiektu nr: 24 generacja: 1 cykli: 2
grupa obiektów utworzona
wywołanie GC Collect
Destruktor obiektu nr: 14 generacja: 2 cykli: 2
Destruktor obiektu nr: 13 generacja: 2 cykli: 3
Destruktor obiektu nr: 12 generacja: 2 cykli: 3

```

KOD ZARZĄDZANY

Interfejs IDisposable

```

public interface IDisposable
{
    void Dispose();
}

```

- Dispose() – implementacja funkcji ma sens w przypadku zwalniania zasobów zewnętrznych,
- do wywołania ręcznie lub w danym zakresie (using)

```

class KlasaDysp : IDisposable
{
    public int a = 0;

    public KlasaDysp(int b)
    {
        a = b;
    }

    public void Dispose()
    {
        Console.WriteLine("Wywołanie Dispose dla " + a);
    }
}

```

KOD ZARZĄDZANY

Interfejs IDisposable / using

```

KlasaDysp kd1 = new KlasaDysp(1);
KlasaDysp kd2 = new KlasaDysp(2);

kd2.Dispose();

using (KlasaDysp kd3 = new KlasaDysp(3))
{
}

```

```

Wywołanie Dispose dla 2
Wywołanie Dispose dla 3

```

KOD ZARZĄDZANY

Interfejs IDisposable / using

```
// https://msdn.microsoft.com/pl-pl/library/yh598w02.aspx

using (Font font1 = new Font("Courier", 12.0f))
{
    byte charset = font1.GdiCharSet;
}

// jest rownowazne:
// utworzenie obiektu przed using - wywołanie po - exc.

{
    Font font1 = new Font("Courier", 12.0f);
    try
    {
        byte charset = font1.GdiCharSet;
    }
    finally
    {
        if (font1 != null)
            ((IDisposable)font1).Dispose();
    }
}
```

KOD ZARZĄDZANY

```
// standardowe i wlasciwe tworzenie pliku (klasa Stream dziedziczy IDisposable)
FileStream fs1 = new FileStream("test.txt", FileMode.OpenOrCreate,
FileAccess.Write);
fs1.WriteByte(65); //A
fs1.Close();

// zamkniecie poprzez jawne wywołanie Dispose - przyklad zasobu niezarazanego
FileStream fs2 = new FileStream("test.txt", FileMode.Append, FileAccess.Write);
fs2.WriteByte(66); //B
fs2.Dispose();

// zamkniecie po zakonczeniu bloku using() - przyklad zasobu niezarazanego
using (FileStream fs3 = new FileStream("test.txt", FileMode.Append, FileAccess.Write))
{
    fs3.WriteByte(67); //C
}

// test dostepu
{
    FileStream fs4 = new FileStream("test.txt", FileMode.Append, FileAccess.Write);
    fs4.WriteByte(68); //D
}

// test dostepu - zamkniecie fs4 odbywa sie w nieokreslonym czasie,
// otwarcie fs5 wyrzuca wyjatke ..
FileStream fs5 = new FileStream("test.txt", FileMode.Append, FileAccess.Write);
fs5.WriteByte(69); //E
fs5.Close();
```

MONO

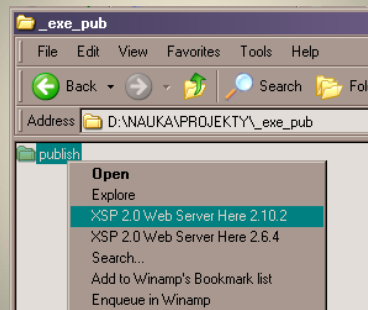
```
Mono-2.10.2 Command Prompt
Mono version 2.10.2 Build 5
Prepending 'C:\PROGRAM~1\MONO-2~1.2\bin' to PATH
C:\WINNT\system32>d:

D:\>cd kasuj\AppTest\bin\Release
D:\KASUJ\AppTest\bin\Release>AppTest.exe
Wplata 1000, UAI:230

D:\KASUJ\AppTest\bin\Release>mono AppTest.exe
Wplata 1000, UAI:230

D:\KASUJ\AppTest\bin\Release>_
```

MONO



MONO

```

c:\WINNT\system32\cmd.exe
xsp2
Listening on address: 0.0.0.0
Root directory: D:\NAUKA\PROJEKTY\exe_pub\publish
Listening on port: 8080 (non-secure)
Hit Return to stop the server.
_

```

Literatura

1. Piórkowski, A., Żupnik, M.: Loop optimization in managed code environments with expressions evaluated only once. *TASK Quarterly*, Vol. 14, No 4, 2010, s. 397-404.
2. Kras, S., Piórkowski, A.: Optymalizacja kodu przy wykorzystaniu algorytmu SSA w środowiskach maszyn wirtualnych. *Studia Informatica*, vol. 37, No 1, 2016, s. 97-111.
3. Olshevskyy Y., Piórkowski A.: Oszacowanie wydajności optymalizacji przy wykorzystaniu liniowej reprezentacji pośredniej kodu. *Studia Informatica*, vol. 39, No 1, 2018, s. 113--125.
4. ...
5. Piórkowski, A., Szemla, P.: Client-Side Processing Environment Based on Component Platforms and Web Browsers. *Computer Networks 2013, CCIS*, vol. 370, Springer, pp. 21-30, 2013
6. Kowal, A., Piórkowski, A., Danek, T., Pięta, A.: Analysis of selected component technologies efficiency for parallel and distributed seismic wave field modeling. In: *Innovations and Advances in Computer Sciences and Engineering*, Springer, pp. 359-362, 2010
7. Piórkowski, A., Pięta A., Kowal A., Danek T.: The Performance of Geothermal Field Modeling in Distributed Component Environment. In: *Innovations in Computing Sciences and Software Engineering*. Springer, pp. 279-283, 2010