SPRÁVA PAMĚTI A ZDROJŮ V .NET FRAMEWORKU

Aleš Keprt
Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého, Olomouc
Ales.Keprt@upol.cz, aley@keprt.cz

ABSTRAKT:
Známé kliše říká, že .NET Framework má automatickou správu paměti, učebnice Basicu či C# toto klišé dále velmi prohlubují. Někteří programátoři se nechají opít rohlíkem a v domnění, že „o vše je postaráno“, vytvářejí nekvalitní a někdy i nefunkční kód. Navíc různé jazyky se chovají jinak, přestože běží společně ve stejném prostředí .NET. Cílem tohoto příspěvku je popsat správu paměti technicky přesně, popsat rozdíly v životním cyklu objektů s automatickou správou paměti mezi jazyky C# a C++/CLI, a to včetně patternů (vzorů) správy zdrojů, což je téma úzce související právě s životními cykly objektů.

KLÍČOVÁ SLOVA:
garbage collector, řízená halda, generace haldy, sémantika destrukce objektů

1 Pojem správy paměti

Správa operační paměti je velmi důležitým úkolem každého operačního systému. Zahrnuje velké množství systémových algoritmů, které jsou před programátory aplikačního softwaru skryty (viz např. [3]). Systém Windows nabízí také poměrně hodně funkcí týkajících se správy paměti ve svém API, většina programovacích jazyků (i ty starší jako např. jazyk C) však toto složité místo zakrývá a zjednodušuje svou vlastní správou paměti. Kromě assembleru, který sám nedělá vůbec nic, prakticky každý programovací jazyk nabízí nějakou svou správu paměti a programátoři při práci používají téměř výhradně jen prostředky svého jazyka a přímo na operační systém se neobracejí.

2 Automatická správa paměti v .NETu

Základní nutnost správy paměti, je být schopen přidělovat, evidovat a uvolňovat paměťové bloky. V případě .NETu hovoříme o tzv. „automatické správě paměti“, kde se uživatelské procesy starají jen o alokaci (řeknou systému, kolik paměti potřebují), zatímco evidenci a uvolňování řeší sám systém. Tento způsob správy paměti je velmi odlišný od toho, jaké činnosti s pamětí provádí systém Windows. [5,7]

Automatická správa paměti v .NETu má dva hlavní přínosy:

- Snižuje chybovost (čili zvyšuje spolehlivost) kódu
- Odstraňuje fragmentaci paměti

Často omílané zjednodušení kódu je jen vedlejším efektem automatické správy, zejména pro začátečníky je však velkým přínosem. (Bylo by ale chybou dávat přednost C# před C++ jen proto, že člověk nechápe správu paměti a chce ji nechat na automatice systému. Taková představa je lichá a vede jen k dalším chybám.)

Paměť se v .NETu dělí v zásadě na tři části:

- Zásobník – zde jsou lokální hodnotové proměnné
- Malá řízená halda – zde jsou malé řízené objekty, do velikosti cca 80KB
- Velká řízená halda – zde jsou velké řízené objekty

2.1 Organizace malé řízené haldy

Malá halda je místo, kde se obvykle nachází většina objektů. Dělí se na tři generace, které číslujeme 0–1–2. Nově alokované objekty jdou vždy do generace 0, jejíž velikost je obvykle do 16MB. Při zaplnění paměti vyhrazené pro kteroukoliv generaci se spouští garbage collector (doslova „sběrač smetí“, dále jen kolektor) a ten ji sloučí (neboli provede úklid, sloučení = kolekce, anglicky collect):

1. Jsou nalezeny dožité objekty a jsou zrušeny.
2. Poslední generace je defragmentována, čili živé objekty jsou přeskládány na začátek její paměti. U ostatních generací se živé objekty přesunou o generaci výše.
3. Pokud předchozí bod způsobí zaplnění vyšší generace, i ta je defragmentována.

Jelikož všechny nové objekty jsou umisťovány do generace 0, tato musí být nejčastěji slučována. Přínos generací je právě v tom, že při slučování není třeba uklízet všechny objekty, ale jen ty, které jsou v zaplněné generaci. Šetříme tedy čas a o další generace se nestaráme, dokud se také nezaplní.

Systém si u každé generace pamatuje pouze její maximální možnou velikost a ukazatel na místo, kam přijde nový objekt – je to vždy přesně za koncem dosud posledního objektu této generace. Každý objekt přitom zabere na haldě přesně tolik bajtů, kolik je jeho velikost (a nic navíc). Ve výsledku je tedy tento způsob paměti na evidenci méně náročný, než systém používaný v prostředích bez automatické správy paměti.

Zajímavou vlastností tohoto alokačního algoritmu je i to, že dodržuje princip časoprostorové lokality: Společně alokované a společně používané objekty jsou v paměti vždy vedle sebe, a to dokonce i po provedení kolekce. Generační systém navíc nejčastěji uklízí generaci 0, takže zohledňuje i to, že většina objektů se obvykle ruší brzy po založení. Naopak objekty, které první úklid přežijí a dostanou se do další generace, ve které neprobíhá úklid tak často, obvykle žijí ještě déle. (Když už se objekt dožil další generace, lze předpokládat, že asi bude žít dlouho. A to je v souladu s tím, že generace starších objektů se nečistí tak často, protože tam pravděpodobně ani žádné dožité objekty nebudou. Tím se činnost kolektoru zrychluje.)

2.2 Pinning

Při spolupráci s neřízeným kódem může být důležité mít možnost zakázat některým objektům přesun v paměti během kolekce. Tato možnost tu je a nazývá se pinning (odvozeno od anglického slova „pin“ = připíchnout špendlíkem). Pinning je nezbytný například tam, kde funkce z knihovny BCL volají Windows API a předávají nějaká data odkazem. Kdyby kolektor přesunul objekty během toho, co běží neřízený kód ve Windows API, program by se zhroutil. Pinning vytváří de facto pointer na řízený objekt, kterému říkáme pin (špendlík). Dokud pin nezrušíme, nebude se příslušný objekt v paměti přesouvat.

2.3 Hledání dožitých objektů

Důležitou otázkou je, jak vlastně kolektor pozná či najde dožité objekty, když nevedeme jejich seznam, ani nijak zvlášť neevidujeme místa paměti, která jsou obsazená či volná. Základem této operace jsou tzv. aplikační kořeny (application roots) – místa, ke kterým se lze přímo dostat z daného místa aplikace, kde se spouští GC. Živé objekty jsou pak přesně ty, které jsou kořeny nebo na ně z kořenů vede přímý či nepřímý odkaz. GC tedy při hledání dožitých objektů prochází objekty skrze strom jejich odkazů a všechny objekty, které nenavštíví, jsou dožité.

Důležitý rozdíl oproti systémům používajícím počítání referencí je tam, kde několik objektů ukazuje na sebe navzájem. Příkladem může být třeba spojový seznam, kde sousední prvky na sebe navzájem ukazují, takže počítadlo referencí nikdy není nulové, i když celou kolekci třeba už nikdo nepoužívá. V .NETu takovou dožitou kolekci zjistí kolektor velmi jednoduše,

protože pokud ji nikdo nepoužívá, nevede na ni žádný odkaz z aplikačních kořenů, takže je možno celou kolekci zrušit.

Aplikační kořeny jsou:

- Všechny globální a statické proměnné
- Všechny lokální proměnné na celém zásobníku
- Argumenty předané do volání metod v celém zásobníku volání
- Všechny registry procesoru odkazující na objekty
- Objekty čekající na finalizaci

2.4 Správa velké haldy

Velká řízená halda nemá generace, jinak funguje stejně jako malá. Smysl její existence je v tom, že přesouvat velké kusy paměti zabere více času, takže úklid této hlady provádí kolektor méně často.

2.5 Explicitní spuštění kolektoru

Kolektor a automatická správa paměti jsou v .NETu zavedeny proto, aby nám zjednodušily naši práci. Nemá proto smysl se nějak starat o ně, když oni se mají starat o nás. V některých velmi specifických případech však může být žádoucí, abychom sami určovali, kdy se má kolektor spouštět.
Ke kolektoru se dostaneme pomocí statické třídy System.GC. Ta nabízí několik metod, z nichž nejdůležitější jsou dvě: Voláním GC.Collect() spustíme kolektor. Nepovinně lze uvést číslo generace, u které má kolekce skončit; standardně se uklízejí všechny generace (odpovídá hodnotě parametru rovné 2). Za toto volání je vhodné přidat ještě volání metody GC.WaitForPendingFinalizers(), která počká na zpracování finalizerů – ukončovacích metod, které se vyskytují u některých tříd. Finalizery si podrobněji vysvětlíme v následující sekci.

Příklad použití: Kolektor takto explicitně spouštějí například počítačové hry, které potřebují kreslit plynulé video (či grafiku) bez cukání. Jelikož běh kolektoru by program na nějakou dobu zastavil a to by mohlo mít špatný dopad na plynulost videa, hra raději pravidelně kolektor sama volá ve vhodných okamžicích.

2.6 Tři verze kolektoru

Distribuce .NET Frameworku ve skutečnosti obsahuje ne jednu, ale hned tři různé implementace kolektoru. Podívejme se nyní na to, jak se mezi sebou liší a jak systém vybírá, kterou z nich na konkrétním počítači použije.

1. Workstation – jednoprocesorová verze
   Tato verze se používá na obyčejných jednoprocesorových počítačích. Kolektor při úklidu zastaví (suspend) všechna řízená vlákna, aby nemohla pracovat s objekty, které právě uklízí. Výhodou je malá režie, protože tento kolektor nepotřebuje řešit synchronizaci. Nevýhodou je, že u realtimových aplikací dochází k viditelnému „cukání“ v okamžicích, kdy kolektor pracuje.
2. Workstation – víceprocesorová verze
   Na víceprocesorových (či vícejádrových) počítačích je generace 0 rozdělena na více částí, tzv. arén, a při alokaci paměti pak každé vlákno alokuje objekty v jiné aréně. Smyslem a výhodou tohoto řešení je, že je možno provádět více alokací současně různými vlákny bez potřeby zamykat haldu. Jelikož na víceprocesorovém počítači mohou teoreticky další vlákna během úklidu haldy vykonávat jiné výpočty, algoritmus úklidu je optimalizován tak, aby ostatní vlákna byla zastavována (suspend) na co nejkratší nutnou dobu.
3. Serverová verze

Tato verze se používá na serverech. Rovněž předpokládá víceprocesorový počítač a rozděluje haldu na samostatné sekce pro jednotlivé procesory. Úklid pak běží paralelně na všech procesorech současně, každý uklízí svou část haldy. Toto řešení je výhodné pro serverové aplikace.

3 Životní cyklus objektu v jazyce C#

3.1 Obyčejné referenční třídy

Věnujme se nyní instancím referenčních tříd (tedy ne hodnotovým instancím, ani proměnným). Životní cyklus objektů je dán architekturou .NETu, v jednotlivých programovacích jazycích se však může v jistých detailech lišit. Např. v jazyce C++ mohou vznikat nové instance při volání funkcí, bez explicitního vytvoření objektu programátorem. Jazyk C# naštěstí používá mnohem jednodušší model než C++: Objekt vznikne jedině použitím operátoru new a je „živý“ tak dlouho, dokud je používán. O odstranění objektu se postará systém automaticky pomocí kolektoru.
Na rozdíl od některých jiných prostředí, např. COM, v .NETu se u objektů nepočítají reference. Na rozdíl od neřízených jazyků, kde se buď sleduje počet referencí, nebo o odstranění objektu požádá sám programátor (např. v C++ operátorem delete), v .NETu nevíme, kdy přesně objekt zanikne. Proto taky není zvykem definovat nějaký kód, který by měl být vykonán při rušení objektu (C++ má pro tento účel destruktory).

3.2 Třídy používající neřízené zdroje

Specifická situace je u tříd pracujících se systémovými zdroji (či jakýmikoliv neřízenými prostředky), které je třeba explicitně uvolnit. Třídy, které se zdroji pracují, by měly implementovat rozhraní IDisposable, které předepisuje jedinou operaci Dispose() a tuto je třeba pak explicitně volat při skončení práce s objektem. (Tato metoda uvolní zdroje a samotný objekt existuje dál až do úklidu paměti kolektorem.)
V .NETu lze u třídy definovat i tzv. finalizer, což je metoda, kterou kolektor zavolá při uklízení objektu. (Je zajímavé, že syntaxe definice finalizeru se v jednotlivých programovacích jazycích dosti liší.) Finalizer zhoršuje efektivitu úklidu, takže jej definujeme pouze u těch tříd, které implementují IDisposable. Správný vzor destrukce instancí tříd používajících neřízené zdroje je tento:

1. Definujeme finalizer, v C# se finalizer definuje jako metoda bez parametrů pojmenovaná stejně jako konstruktor s přidáním vlnovky na začátek, tedy např. ~Třída(). Ve finalizeru korektně ukončíme neřízené zdroje.
2. Implementujeme IDisposable. Přidáme proměnnou IsDisposed typu bool, kterou použijeme ke sledování, zda bylo Dispose() již zavoláno.
3. Metodu Dispose() implementujeme takto: Ukončíme všechny řízené zdroje, tj. voláme Dispose() a přiřadíme null do všech referencí, které jsou IDisposable. Potom ukončíme i neřízené zdroje a nastavíme proměnnou IsDisposed na true.
4. Do všech veřejných součástí třídy včetně metody Dispose() přidáme na začátek test proměnné IsDisposed. Je-li true, vyvoláme výjimku System.ObjectDisposedException (nelze používat zrušený objekt).

Zdrojový kód vypadá takto:

```
class Třída : IDisposable {
  bool disposed;

  public bool IsDisposed {
    get { return disposed; }
```

```
  }

  void Úklid(bool disposing) {
    if(!disposed) {
      disposed = true;
      if(disposing) {
        ...úklid řízených zdrojů...
      }
      ...úklid neřízených zdrojů...
    }
  }

  public void Dispose() {
    if(IsDisposed) throw new System.ObjectDisposedException();
    Úklid(true);
    GC.SuppressFinalize(this);
  }

  ~Třída() {
    Úklid(false);
  }
}
```

Tento vzor se týká jazyka C# a některých dalších, ne však všech. Například C++/CLI používá jinou sémantiku destrukce, budeme ji diskutovat později.
Řada programátorů přešla k jazyku C# z C++ a zřejmě proto se finalizerům v C# často ne zcela přesně říká destruktory. Jak si ukážeme dále v textu, při srovnání s C++ jsou zde ve skutečnosti jisté odlišnosti.

3.3 Vliv finalizeru na život a resurekce (oživování mrtvých) objektů

V předchozí sekci jsme mlčky přešli volání GC.SuppressFinalize(this) v metodě Dispose(), nyní si jej vysvětlíme.
Kolektor při úklidu dožitých objektů u každého z nich kontroluje, zda implementuje finalizer. Pokud ano, tak je kolekce provedena dvoufázově. Nejprve místo zrušení objektu je tento jen označen k finalizaci a je nechán živý. Toto „označení“ znamená, že je objekt připojen do zvláštního seznamu objektů čekajících na finalizaci. Finalizace je pak provedena na pozadí zvláštním finalizačním vláknem až po skončení aktuálního úklidu a objekt zůstává v paměti dokonce až do dalšího úklidu (toto v dané situaci už přímo plyne z logiky věci). Každý finalizovatelný objekt žije (tak trochu zbytečně) o jeden úklid déle, než by bylo třeba jen kvůli tomu, že je finalizovatelný. Proto při volání Dispose(), kdy jsou všechny neřízené prostředky uvolněny a následné volání finalizeru tedy již nemá smysl, voláme GC.SuppressFinalize(this), čímž kolektoru oznámíme, že na tomto objektu již finalizaci provádět nemá a má jej hned napoprvé opravdu zrušit.
Tento vzor dvoufázové destrukce ukazuje ještě jednu zvláštní možnost: Pokud bychom to z nějakého důvodu chtěli, ve finalizeru můžeme objekt oživit (resurektovat) – jednoduše jej připojíme zpět k aplikačním kořenům (obvykle se to dělá přiřazením reference do nějaké statické proměnné či kolekce).

3.4 Třída agregující objekty používající neřízené zdroje

Pokud naše třída sama s neřízenými zdroji nepracuje a pouze obsahuje objekty jiných tříd, které jsou IDisposable, stačí implementovat jednodušší vzor: Naše třída musí být také IDisposable a v metodě Dispose() zavoláme Dispose() u všech agregovaných objektů. Finalizer ale nepotřebujeme. Pokud uživatel řádně zavolá naše Dispose(), jsou zavolány i Dispose() vnitřních objektů a jejich neřízené zdroje jsou správně uvolněny. Kdyby uživatel Dispose() na našem objektu zavolat zapomněl, finalizery vnitřních objektů se zavolají tak jako tak při jejich destrukci. Je tedy vidět, že finalizery se v praxi používají mnohem méně často, než IDisposable.

3.5 Specifika C++/CLI

Jazyk C++/CLI poskytuje bohatší možnosti co do správy paměti. Například umožňuje používání hodnotových instancí referenčních typů, kdy objekty definujeme tak jako v klasickém C++ jako lokální hodnotové proměnné a překladač sám zajistí dodržení referenční sémantiky.
C++/CLI umožňuje používat i neřízené třídy, jejichž instance vytváří na zvláštní neřízené haldě; odpovídá to chování klasického C++, takže C++/CLI má vlastně automatickou i neautomatickou správu paměti dohromady. Neřízené typy jsou ukončovány stejně jako v C++ operátorem delete nebo delete[]. U řízených typů se sémantika C++/CLI liší jak od klasického C++, tak od C#.
Deklarujeme-li klasický „vlnovkový“ destruktor v řízené třídě, tento je nativně přeložen do metody Dispose(), přitom je třída automaticky označena jako IDisposable. Tento destruktor můžeme explicitně volat pomocí klasické konstrukce ~Třída(). Dealokační scénář by měl odpovídat doporučením CLI, čili je třeba deklarovat také finalizer (což odpovídá výše diskutovanému vlnovkovému destruktoru jazyka C#). Finalizer je deklarován podobně jako destruktor, ale s uvedením vykřičníku místo vlnovky před názvem typu.

Následuje ukázka doporučeného řešení:

```
ref class FinClass {
  //destruktor - ukončuje řízené zdroje a volá finalizer
  ~FinClass() {
    ...uvolnění řízených zdrojů...
    this->!FinClass();
  }

  //finalizer - ukončuje neřízené zdroje
  !FinClass() {
    ...uvolnění neřízených zdrojů...
  }
};
```

Poznámka: Všimněte si, že v C++/CLI je tento kód mnohem jednodušší, než v C#.

Na konci destruktoru tedy voláme vlastní finalizer (Pozor! Ze syntaktických důvodů je nutno jej volat odkazem přes this.), překladač sám doplní kód zajišťující, aby se po zavolání destruktoru finalizer již nevolal (toto ošetření jsme v C# dělali ručně). Na tomto místě je vidět, že C++/CLI je novější než C# a jeho autoři se poučili z chyb v návrhu jazyka C#. Koncept destrukce objektů v C++/CLI je totiž jednoznačně lepší a jako takový by se jistě hodil i do příští verze jazyka C#. Základní koncepční rozdíl lze popsat takto: C# vznikl v době kdy mezi odborníky převládalo přesvědčení, že správný princip zní: „Kolektor místo destruktoru.“ O několik let novější jazyk C++/CLI již funguje na principu: „Kolektor a destruktor.“ a praxe ukazuje, že tento model správy paměti je lepší.

S modelem destrukce souvisí i výše zmíněná možnost používat hodnotové proměnné řízených referenčních typů, které můžeme vytvořit jak v rámci metody, tak v rámci jiné třídy. Příklad následuje.

```
ref class A { ... };
ref class B {
 ...
 A a; //lokální instance typu A
};

void Metoda() {
  A a; //lokální instance typu A
  B b; //lokální instance typu B
}
```

C++/CLI u těchto proměnných zajišťuje automatické volání destruktorů v okamžiku ukončení nadřazeného bloku kódu či objektu. I tento prvek jde za hranice C#, C++/CLI je zřejmě dokonce prvním jazykem umožňujícím tento pohodlný způsob vytváření instancí a řízené destrukce objektů referenčních typů. V jazyce C# lze toto pouze částečně nahradit blokovým příkazem using. Pro více informací o C++/CLI a jeho paměťovém modelu viz [2,6].

4 Další témata

4.1 Chování systému při nedostatku paměti

Současné počítače mají dost paměti na to, aby většina programů nemusela nikdy nedostatek paměti potkat a tudíž ani řešit. Přesto se však můžeme podívat, co se v systému při nedostatku paměti děje.

Nedostatek paměti může nastat ve dvou případech: Buď je plná halda, nebo je plný zásobník. V případě nedostatku místa pro vytvoření nového objektu na haldě se aktivuje kolektor a ten získá další paměť úklidem. Pokud ani po úklidu není paměti dost, je vyhozena výjimka System.OutOfMemoryException. Tuto výjimku lze běžným způsobem zachytit pomocí try–catch bloku a ošetřit. Během hledání příslušného catch bloku se mohou spouštět mezilehlé finally bloky a ty mohou uvolňovat objekty. Je tedy možné, že v místě zachycení výjimky bude původně chybějící paměť již k dispozici (ačkoliv z toho obvykle nelze nic vytěžit, protože došlo mezitím ke ztrátě jiných objektů). Pravidla .NETu také vyžadují (a to je důležité!), aby se při uvolňování objektů nic nového nealokovalo. Tzn. finalizer a Dispose() nesmějí nic alokovat, takže při úklidu paměti nemůže dojít k zacyklení neustálým vyhazováním výjimky OutOfMemoryException.

Druhá možnost zaplnění paměti se týká zásobníku. Připomeňme, že u programového zásobníku se při kompilaci klasickým způsobem (tj. stejně jako u nativního kódu) nastavují hodnoty committed a reserved, tedy jeho počáteční a maximální velikost. Každé vlákno má nejprve zásobník o velikosti committed a při jeho zaplnění se tento automaticky zvětšuje až na velikost reserved. Při maximálním zaplnění a dalším nedostatku je vyhozena výjimka System.StackOverflowException. Alokace místa na zásobníku se přitom provádí jen na začátku metody – tehdy se vytvoří místo pro všechny lokální proměnné definované v této metodě bez ohledu na to, jaký přesně je jejich rozsah platnosti (ten totiž může být menší, než na celou metodu). Výjimka StackOverflowException znamená, že program nemůže dál pokračovat. Tuto výjimku nelze zachytit v try–catch bloku, program je totiž okamžitě ukončen pomocí System.Environment.FailFast(). Běhové prostředí pouze umožňuje nastavit, aby systém při StackOverflowException ukončil jen aplikační doménu, kde chyba vznikla, a zbytek procesu zůstal běžet. Toto je tedy jediná alternativa k okamžitému ukončení celého procesu, navíc není k dispozici na Windows 9x. Více viz [1,4].

4.2 Paměťová brána

Čistější způsob, jak se vypořádat s možným nedostatkem paměti, nabízí třída System.Runtime.MemoryFailPoint. Vytvořením objektu této třídy vzniká tzv. paměťová brána (memory gate), velikost požadované paměti v megabajtech zadáme jako parametr konstruktoru. Vytvořením brány systém nezaručuje dostatek paměti dlouhodobě, ale v okamžiku jejího vytvoření. Pokud však všechen kód alokující paměť (a to především velké bloky) používá brány, tak díky tomu, že brány jsou IDisposable, systém zohlední celkovou kapacitu potřebnou všemi otevřenými branami. Zavoláním Dispose() se brána uzavře a dáme tím signál, že danou paměť již nepotřebujeme rezervovat. Vzor použití brány vypadá takto:

```
using (MemoryFailPoint gate = new MemoryFailPoint(kolik)) { ... }
```

Při volání konstruktoru se zjistí, zda je paměti dostatek. Při nedostatku se postupuje takto:

1. Provede se kolekce všech částí řízené haldy.
2. Pokud stále není dostatek paměti, systém se pokusí zvětšit swap file ve Windows.
3. Pokud to stále nestačí, je vyhozena výjimka InsufficientMemoryException. (Je to tedy jiná výjimka, než OutOfMemoryException vyhozená při skutečném nedostatku paměti v okamžiku vytváření objektů.) Program může tuto výjimku zachytit, je to signál, že blok kódu, který měl pracovat v rámci brány, ani nezačal. To je často daleko bezpečnější, než složitější operaci začít a teprve b jejím průběhu zjistit nedostatek paměti – nemusíme totiž řešit úklid a zotavení systému z nekonzistentního stavu.

5 Shrnutí

V tomto příspěvku jsme se seznámili se správou paměti a zdrojů a zejména pak nahlédli pod pokličku automatické správy paměti v .NETu. V první části jsme se seznámili s tím, jak správa objektů funguje a především jsme se zabývali (garbage) kolektorem, který se stará o úklid dožitých objektů z paměti a slučování objektů živých. Ve druhé části kapitoly jsme diskutovali třídy používající neřízené zdroje (a to přímo, či nepřímo přes agregaci jiných objektů). V závěru jsme ještě nakoukli k jazyku C++/CLI, který přestože je syntakticky podobný, používá zcela jinou sémantiku destrukce objektů a souvisejících věcí.

LITERATURA

1. Joe Duffy. *Professional .NET Framework 2.0.* Wrox Press, **2006**. ISBN 0-7645-7135-4, ISBN-13: 978-0-7645-7135-0.
2. Aleš Keprt. Kombinace C++ a .NET – jak a proč. Ve sborníku konference: *Objekty 2006.* Česká zemědělská univerzita, Praha, **2006**, pp. 193–208. ISBN 80-213-1568-7.
3. Aleš Keprt. *Operační systémy.* Univerzita Palackého, **2007**. Text pro distanční vzdělávání.
4. Visual Studio 2005/2008 – nápověda MSDN. Na webu: http://msdn.microsoft.com/.
5. Jeffrey Richter. Garbage Collection – Part 2: Automatic Memory Management in the Microsoft .NET Framework. V časopise: *MSDN Magazine.* Dec **2000**. ISSN 1528-4859.
6. Nishant Sivakumar. *C++/CLI in Action.* Manning, **2006**. 416 pp., ISBN: 1-932394-81-8.
7. Andrew Troelsen. *Pro C# 2008 and the .NET 3.5 Platform*, 4.vydání. Apress, **2007**. 1370pp., ISBN 1-59059-884-9, 978-1-59059-884-9.

ANNOTATION

This paper describes memory and resource management and especially the automatic memory management of .NET Framework. The first part presents how the management of object works in .NET and what and how does its garbage collector, which is responsible for cleaning the memory of dead objects and merging the live ones. The second part discuses C# classes using unmanaged resources (either directly, or indirectly via aggregation of other objects). The third part aims at C++/CLI language, which (although syntactically very similar to C#) uses very different object destruction semantics.