



Zaawansowane zarządzanie pamięcią w .NET

Lepszy kod, wydajność
i skalowalność

—

Konrad Kokosa

Apress®

Konrad Kokosa

Zaawansowane zarządzanie pamięcią w .NET

**Lepszy kod, wydajność
i skalowalność**

przekład: Jakub Niedźwiedź

APN Promise, Warszawa 2020

Zaawansowane zarządzanie pamięcią w .NET. Lepszy kod, wydajność i skalowalność

First published in English under the title

Pro .NET Memory Management; For Better Code, Performance, and Scalability

by Konrad Kokosa, edition: 1

Copyright © Konrad Kokosa, 2018

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature.

APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from publisher.

Polish language edition published by APN PROMISE S.A., Copyright © 2020

Autoryzowany przekład z wydania w języku angielskim, zatytułowanego: Pro .NET Memory Management; For Better Code, Performance, and Scalability, by Konrad Kokosa, edition: 1, opublikowanego przez APress Media, LLC, oddział Springer Nature.

Wszystkie prawa zastrzeżone. Żadna część niniejszej książki nie może być powielana ani rozpowszechniana w jakiegokolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny), włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów bez pisemnej zgody wydawcy.

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa

tel. +48 22 35 51 600, fax +48 22 35 51 699

e-mail: mspress@promise.pl

Książka ta przedstawia poglądy i opinie autorów. Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń, chyba że zostanie jednoznacznie stwierdzone, że jest inaczej. Ewentualne podobieństwo do jakiegokolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

Wszystkie znaki towarowe występujące w książce mogą być własnością ich odnośnych właścicieli.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji.

Jednakże nikomu nie udziela się rękojmi ani gwarancji.

APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiegokolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-413-4 (druk), 978-83-7541-412-7 (ebook)

Przekład: Jakub Niedźwiedź

Redakcja merytoryczna: Konrad Kokosa

Korekta: Ewa Świędrowska

Skład i łamanie: MAWart Marek Włodarz

*Ukochanej żonie – Justynie, bez której nic naprawdę cennego
nie zdarzyłoby się w moim życiu.*

Spis treści

O autorze	xiii
Podziękowania	xv
Przedmowa	xvii
Wprowadzenie	xix
Rozdział 1. Podstawowe pojęcia	1
Terminy związane z pamięcią	3
Alokacja statyczna	10
Maszyna rejestrowa	11
Stos	12
Maszyna stosowa	19
Wskaźnik	23
Serta	25
Ręczne zarządzanie pamięcią	28
Automatyczne zarządzanie pamięcią	34
Alokator, mutator i kolektor	36
Zliczanie referencji	42
Kolektor śledzący	48
Faza oznaczania (Mark)	49
Faza zbierania (Collect)	54
Nieco historii	58
Podsumowanie	61
Zasada 1 – Ucz się	61
Rozdział 2. Niskopoziomowe zarządzanie pamięcią	63
Sprzęt	64
Pamięć	71
Procesor (CPU)	73
System operacyjny	94
Pamięć wirtualna	94
Duże strony	99
Fragmentacja pamięci wirtualnej	100

Spis treści

Ogólny układ pamięci	101
Zarządzanie pamięcią w Windows	103
Układ pamięci systemu Windows	109
Zarządzanie pamięcią w systemie Linux	112
Układ pamięci w systemie Linux	114
Wpływ systemu operacyjnego	116
NUMA i grupy procesorów	117
Podsumowanie	119
Zasada 2 – Należy unikać losowego dostępu, a promować dostęp sekwencyjny	119
Zasada 3 – Poprawiaj przestrzenną i czasową lokalność danych	120
Zasada 4 – Używaj bardziej zaawansowanych możliwości	121
Rozdział 3. Pomiar pamięci	123
Mierzyć wcześniej	125
Obciążenie i inwazyjność	126
Próbkowanie i śledzenie	126
Drzewo wywołań	127
Grafy obiektów	128
Statystyki	130
Opóźnienie a przepustowość	134
Zrzuty pamięci, śledzenie, debugowanie na żywo	136
Środowisko systemu Windows	137
Przegląd	137
VMMMap	138
Liczniki wydajności	139
Śledzenie zdarzeń dla systemu Windows	146
Zestaw narzędzi Windows Performance Toolkit	160
PerfView	173
ProcDump, DebugDiag	184
WinDbg	185
Deasembler i dekompile	188
BenchmarkDotNet	189
Narzędzia komercyjne	191
Środowisko Linux	202
Przegląd	202
Perfcollect	204
Trace Compass	206
Zrzuty pamięci	217
Podsumowanie	219
Zasada 5 – Mierz odświeżanie pamięci wcześniej	221
Rozdział 4. Podstawy .NET	223

Wersje .NET	224
Wewnętrzne mechanizmy .NET	227
Dogłębna analiza przykładowego programu	231
Podzespoły i domeny aplikacji	238
Usuwalne podzespoły	240
Regiony pamięci procesu	241
Scenariusz 4-1. Jak dużo pamięci zajmuje mój program?	247
Scenariusz 4-2. Zużycie pamięci przez nasz program stale rośnie	249
Scenariusz 4-3. Zużycie pamięci przez nasz program stale rośnie	252
Scenariusz 4-4. Zużycie pamięci przez nasz program stale rośnie	255
System typów	259
Kategorie typów	259
Przechowywanie typów	262
Typy wartościowe	263
Typy referencyjne	273
Łańcuchy znaków	281
Internowanie łańcuchów znaków	288
Scenariusz 4-5. Zużycie pamięci przez nasz program jest zbyt duże	295
Opakowywanie i rozpakowywanie	298
Przekazywanie przez referencję	304
Przekazywanie wystąpienia typu wartościowego przez referencję	304
Przekazywanie wystąpienia typu referencyjnego przez referencję	306
Lokalność danych dla typów	307
Dane statyczne	311
Pola statyczne	311
Wewnętrzne działanie danych statycznych	312
Podsumowanie	318
Struktury	318
Klasy	319
Rozdział 5. Partycjonowanie pamięci	323
Strategie partycjonowania	324
Partycjonowanie według rozmiaru	326
Serta małych obiektów	328
Serta dużych obiektów	328
Partycjonowanie ze względu na czas życia	333
Scenariusz 5-1. Czy mój program jest zdrowy? Rozmiary generacji w czasie	339
Pamiętane zbiory	344
Tabele kart	350
Pakiety kart	357
Partycjonowanie fizyczne	360
Scenariusz 5-2. Wyciek pamięci w nopCommerce?	367

Spis treści

Scenariusz 5-3. Marnowanie miejsca w stercie dużych obiektów?	378
Anatomia segmentów i sterty	380
Ponowne wykorzystanie segmentów	383
Podsumowanie	387
Zasada 11 – Monitoruj rozmiary generacji	387
Zasada 12 – Unikaj niepotrzebnych referencji na stercie	388
Zasada 13 – Monitoruj użycie segmentów	389
Rozdział 6. Alokacja pamięci	391
Wprowadzenie do alokacji	392
Alokacja z przesuwaniem wskaźnika	393
Alokacja oparta na liście wolnych obszarów	401
Tworzenie nowego obiektu	406
Alokacja na stercie małych obiektów	408
Alokacja na stercie dużych obiektów	412
Równoważenie stert	416
Wyjątek OutOfMemoryException	420
Scenariusz 6-1. Brak pamięci	421
Alokacje na stosie	424
Unikanie alokacji	426
Jawne alokacje typów referencyjnych	428
Ukryte alokacje	458
Różne ukryte alokacje wewnątrz bibliotek	468
Scenariusz 6-2. Badanie alokacji	474
Scenariusz 6-3. Azure Functions	478
Podsumowanie	479
Zasada 14 – Unikaj alokacji na stercie na ścieżkach kodu krytycznych ze względu na wydajność ..	480
Zasada 15 – Unikaj nadmiernych alokacji na stercie LOH	480
Zasada 16 – Preferuj alokacje na stosie, gdy jest to właściwe	481
Rozdział 7. Wprowadzenie do odśmiecania pamięci	483
Widok wysokopoziomowy	484
Proces odśmiecania pamięci na przykładzie	485
Kroki procesu odśmiecania pamięci	493
Scenariusz 7-1. Analizowanie wykorzystania odśmiecania pamięci	494
Profilowanie odśmiecania pamięci	499
Dane dostrajające wydajność odśmiecania pamięci	501
Dane statyczne	501
Dane dynamiczne	504
Scenariusz 7-2. Zrozumienie budżetu alokacji	508
Wyzwalacze odśmiecania	519
Wyzwalanie przez alokację	520

Wyzwalanie jawne	521
Scenariusz 7-3. Analizowanie jawnych wywołań odśmiecania pamięci	526
Wyzwalacz systemowy przy niskim poziomie pamięci	533
Różne wyzwalacze wewnętrzne	534
Wstrzymywanie silnika wykonawczego	535
Scenariusz 7-4. Analizowanie czasów wstrzymania	538
Generacja do skazania	540
Scenariusz 7-5. Analiza skazanych generacji	544
Podsumowanie	546
Rozdział 8. Odśmiecanie pamięci – faza oznaczania	547
Przechodzenie przez graf obiektów i oznaczanie	547
Korzenie w zmiennych lokalnych	549
Przechowywanie zmiennych lokalnych	550
Korzenie na stosie	551
Zakres leksykalny	552
Żywe korzenie na stosie a zakres leksykalny	553
Żywe korzenie na stosie z gorliwym odśmiecaniem korzeni (eager root collection)	555
Informacje GC	563
Przypięte zmienne lokalne	569
Skanowanie korzeni na stosie	572
Korzenie finalizacji	573
Wewnętrzne korzenie GC	574
Korzenie uchwytów GC	575
Obsługa wycieków pamięci	584
Scenariusz 8-1. Wyciek pamięci w nopCommerce?	586
Scenariusz 8-2. Identyfikowanie najpopularniejszych korzeni	590
Podsumowanie	593
Rozdział 9. Odśmiecanie pamięci – faza planowania	595
Sterta małych obiektów	596
Zaślepki (plugs) i luki (gaps)	596
Scenariusz 9-1. Zrzut pamięci z nieprawidłowymi strukturami	602
Tabela klocków (brick table)	604
Przypinanie	606
Scenariusz 9-2. Badanie przypinania	612
Granice generacji	618
Degradowanie	619
Sterta dużych obiektów	624
Zaślepki i luki	624
Decydowanie o kompaktowaniu	627
Podsumowanie	629

Rozdział 10. Odśmiecianie pamięci – zmiatanie i kompaktowanie	631
Faza zmiatania	631
Sterta małych obiektów	632
Sterta dużych obiektów	633
Faza kompaktowania	634
Sterta małych obiektów	634
Sterta dużych obiektów	640
Scenariusz 10-1 Fragmentacja sterty dużych obiektów	640
Podsumowanie	651
Zasada 17 – Obserwuj wstrzymania środowiska uruchomieniowego	652
Zasada 18 – Unikaj „kryzysu wieku średniego”	653
Zasada 19 – Unikaj fragmentacji w starej generacji i na stercie LOH	654
Zasada 20 – Unikaj jawnego odśmieciania pamięci	655
Zasada 21 – Unikaj wycieków pamięci	655
Zasada 22 – Unikaj przypinania	656
Rozdział 11. Odmiany odśmieciania pamięci	659
Przegląd trybów	659
Tryb stacji roboczej a tryb serwera	659
Tryb niewspółbieżny a współbieżny	662
Konfiguracja trybów	663
Platforma .NET Framework	664
Platforma .NET Core	665
Przerwa na odśmiecianie pamięci i obciążenie procesora	667
Opisy trybów	670
Niewspółbieżny tryb stacji roboczej (Workstation Non-Concurrent)	670
Współbieżny tryb stacji roboczej (Workstation Concurrent, przed wersją 4.0)	672
Tryb stacji roboczej w tle (Background Workstation)	674
Niewspółbieżny tryb serwera (Server Non-Concurrent)	685
Tryb serwera w tle (Background Server)	687
Tryby opóźnień (latency modes)	689
Tryb wsadowy	690
Tryb interakcyjny	690
Niskie opóźnienie	691
Trwałe niskie opóźnienie	692
Region bez odśmieciania pamięci	694
Cele optymalizacji opóźnień	697
Wybieranie odmiany odśmieciania pamięci	698
Scenariusz 11-1. Sprawdzanie ustawień odśmieciania pamięci	700
Scenariusz 11-2. Testowanie różnych trybów odśmieciania pamięci	703
Zasada 23 – Świadomie wybieraj tryb odśmieciania pamięci	712
Zasada 24 – Pamiętaj o trybach opóźnień	713

Rozdział 12. Czas życia obiektów	715
Cykl życia obiektów i zasobów	716
Finalizacja	718
Wprowadzenie	718
Problem z gorliwym odśmiecaniem korzeni	724
Finalizatory krytyczne	728
Wewnętrzne działanie finalizacji	729
Scenariusz 12-1. Wyciek pamięci przy finalizacji	739
Wskrzeszanie	747
Obiekty wykorzystujące wzorzec Disposable	751
Bezpieczne uchwytty (safe handles)	759
Słabe referencje	766
Buforowanie (caching)	772
Wzorzec słabych zdarzeń	775
Scenariusz 12-2. Wyciek pamięci z powodu zdarzeń	783
Podsumowanie	786
Zasada 25 – Unikaj finalizatorów	787
Zasada 26 – Preferuj jawne czyszczenie	788
Rozdział 13. Różnorodne zagadnienia	791
Uchwytty zależne	792
Pamięć lokalna wątku	799
Pola statyczne wątku	800
Sloty na dane wątku	804
Wewnętrzne działanie pamięci lokalnej wątku	805
Scenariusze użycia	814
Wskaźniki zarządzane	815
Zmienne lokalne ref	817
Zwracane wartości ref	818
Zmienne ref tylko do odczytu i parametry in	820
Wewnętrzne działanie typów ref	826
Wskaźniki zarządzane w C# – zmienne ref	841
Więcej na temat struktur	849
Struktury tylko do odczytu	850
Struktury ref (typy w rodzaju byref)	852
Bufory o stałych rozmiarach	855
Układ obiektów/struktur	860
Ograniczenie unmanaged	873
Typy kopiowalne (blittable)	879
Podsumowanie	882
Rozdział 14. Techniki zaawansowane	883

Spis treści

Span<T> i Memory<T>	883
Span<T>	884
Memory<T>	903
IMemoryOwner<T>	907
Wewnętrzne działanie Memory<T>	913
Wskazówki dotyczące Span<T> i Memory<T>	916
Typ Unsafe	916
Wewnętrzne działanie Unsafe	923
Projektowanie zorientowane na dane	924
Projektowanie taktyczne	926
Projektowanie strategiczne	930
Więcej na temat przyszłości...	943
Typy referencyjne dopuszczające wartość null	943
Potoki	951
Podsumowanie	959
Rozdział 15. Programowe interfejsy API	961
Interfejs API dla odśmiecania pamięci	962
Dane i statystyki dotyczące odśmiecania pamięci	962
Powiadomienia ze strony odśmiecania pamięci	973
Kontrolowanie obciążenia pamięci niezarządzanej	976
Jawne odśmiecanie	976
Regiony bez odśmiecania pamięci	977
Zarządzanie finalizacją	977
Użycie pamięci	977
Wewnętrzne wywołania w klasie GC	979
Hosting środowiska uruchomieniowego CLR	980
ClrMD	992
Biblioteka TraceEvent	1000
Niestandardowe odśmiecanie pamięci	1003
Podsumowanie	1008
Indeks	1011

0 autorze

Konrad Kokosa jest doświadczonym projektantem oprogramowania i programistą szczególnie interesującym się technologiami Microsoft, ale przyglądającym się z zaciekawieniem również wielu innym. Programuje od kilkunastu lat, rozwiązując problemy wydajnościowe i zagadki architektoniczne w świecie .NET oraz projektując i przyspieszając aplikacje .NET. Jest niezależnym konsultantem, prowadzi blog <http://tooslowexception.com>, aktywnie uczestniczy w konferencjach i spotkaniach członków społeczności oraz jest fanem Twittera (@konradkokosa). Jego pasją jest też prowadzenie szkoleń w obszarze .NET, zwłaszcza związanych z wydajnością aplikacji, dobrymi praktykami kodowania oraz diagnostyką. Posiada tytuł Microsoft MVP w kategorii Visual Studio and Development Tools. Jest współzałożycielem Dotnetos.org – inicjatywy trzech fanów .NET, mającej na celu organizowanie pokazów i konferencji dotyczących wydajności w .NET.

0 recenzentach technicznych

Damien Foggon jest programistą, autorem i recenzentem technicznym w zakresie najnowszych technologii, który przyczynił się do wydania ponad 50 książek dotyczących .NET, C#, Visual Basic i ASP.NET. Jest współzałożycielem grupy użytkowników NEBytes z Newcastle (w sieci pod adresem <http://www.nebytes.net>), posiada wiele certyfikatów MCPD w zakresie .NET 2.0 i kolejnych wersji, a w sieci można go znaleźć pod adresem <http://blog.fasm.co.uk>.

Maoni Stephens jest architektką i główną programistką w zakresie .NET GC w firmie Microsoft. Jej blog można znaleźć pod adresem <https://blogs.msdn.microsoft.com/maoni/>.

Podziękowania

Przede wszystkim bardzo chciałbym podziękować swojej żonie. Bez jej wsparcia ta książka nigdy by nie powstała. Zaczynając pracę nad tą książką, nie wyobrażałem sobie, jak wiele czasu, który moglibyśmy spędzić razem, będę musiał poświęcić, aby ją napisać. Dziękuję Ci za cierpliwość, wsparcie i otuchę, jakimi obdarowałaś mnie w tym czasie!

Po drugie, chciałbym podziękować Maoni Stephens za wyczerpujące, dokładne i bezcenne uwagi podczas recenzowania pierwszych wersji tej książki. Bez cienia wątpliwości mogę powiedzieć, że dzięki niej ta książka jest lepsza. Fakt, że główna programistka .NET GC pomagała mi w pisaniu tej książki, jest dla mnie nagrodą samą w sobie! Wiele podziękowań kieruję też do innych członków zespołu .NET, którzy pomagali mi w zweryfikowaniu niektórych części tej książki przy wielkiej pomocy ze strony Maoni (w kolejności poświęconych nakładów pracy): Stephen Toub, Jared Parsons, Lee Culver, Josh Free i Omar Tawfik. Chciałbym też podziękować Markowi Probstowi z Xamarin za przejrzenie uwag dotyczących środowiska uruchomieniowego Mono. Specjalne podziękowania należą się Patrickowi Dussud, „ojcu .NET GC” za czas poświęcony na recenzowanie historii powstania CLR.

Po trzecie, chciałbym podziękować Damienowi Foggonowi, recenzentowi technicznemu z wydawnictwa Apress, który poświęcił wiele pracy na skrupulatną weryfikację wszystkich rozdziałów. Jego doświadczenie w wydawaniu i pisaniu książek było bezcenne i sprawiło, że ta książka stała się bardziej zrozumiała i spójna. Nieraz byłem zaskoczony dokładnością uwag i sugestii Damiena.

Chciałbym podziękować wszystkim pracownikom Apress, bez których ta książka nie zostałaby wydana. Specjalne podziękowania kieruję do Laury Berendson (redaktor prowadzący), Nancy Chen (redaktor koordynujący) i Joan Murray (starszy redaktor) za wsparcie i cierpliwość przy ciągłym przedłużaniu terminów. W pewnym momencie data dostarczenia ostatecznej wersji stała się tematem tabu w naszych rozmowach! Chciałbym też podziękować Gwenan Spearing, z którą zacząłem pracę nad tą książką, ale nie zdążyłem jej ukończyć, zanim odeszła z zespołu Apress.

Chciałbym podziękować świetnej społeczności .NET w Polsce i na całym świecie za inspiracje czerpane z wielu prezentacji, artykułów i wpisów w sieciach

społecznościowych, za całe poparcie i zachęty oraz za niekończące się pytania „Jak tam prace nad książką?”. Takie podziękowania kieruję szczególnie do następujących osób (kolejność alfabetyczna): Maciej Aniserowicz, Arkadiusz Benedykt, Sebastian Gębski, Michał Grzegorzewski, Jakub Gutkowski, Paweł Klimczyk, Szymon Kulec, Paweł Łukasik, Alicja Musiał, Łukasz Olbromski, Łukasz Pyrzyk, Bartek Sokół, Sebastian Solnica, Paweł Sroczyński, Jarek Stadnicki, Piotr Stapp, Michał Śliwoń, Szymon Warda i Artur Wincenciak, a także do wszystkich posiadaczy tytułu MVP (zwłaszcza z zakresu Azure) oraz do wielu innych; przepraszam szczerze tych, których pominąłem; wielkie dzięki wszystkim, którzy uważają, że zasłużyli na takie podziękowania. Wymienienie tutaj wszystkich po prostu nie jest możliwe. Dziękuję za inspirację i otuchę.

Chciałbym podziękować wszystkim doświadczonym autorom, którzy znaleźli czas na udzielenie mi rad dotyczących pisania książek, szczególnie Tedowi Newardowi (<http://blogs.tedneward.com/>) i Jonowi Skeetowi (<https://codeblog.jonskeet.uk>) – choć założę się, że nie pamiętają tych rozmów! Andrzej Krzywda (<http://andrzejonsoftware.blogspot.com>) i Gynvael Coldwind (<https://gynvael.coldwind.pl>) również udzielili mi wielu cennych rad dotyczących pisania i wydawania książek.

Następnie chciałbym wymienić wszystkich twórców świetnych narzędzi i bibliotek, z których korzystałem podczas pisania tej książki: Andrey Shchekin, twórca SharpLab (<https://sharplab.io>); Andrey Akinshin, twórca BenchmarkDotNet (<https://benchmarkdotnet.org>) oraz Adam Sitnik, główny wspierający ten projekt; Sergey Teplyakov, twórca ObjectLayoutInspector (<https://github.com/SergeyTeplyakov/ObjectLayoutInspector>); 0xd4d, anonimowy twórca dnSpy (<https://github.com/0xd4d/dnSpy>); Sasha Goldshtein, twórca wielu przydatnych narzędzi pomocniczych (<https://github.com/goldshtn>); wszyscy twórcy takich świetnych narzędzi, jak PerfView i WinDbg (oraz wszystkich rozszerzeń związanych z .NET).

Chciałbym też podziękować swojemu wcześniejszemu pracodawcy, Bankowi Millennium za pomoc i wsparcie przy rozpoczęciu prac nad tą książką. Nasze drogi się rozeszły, ale zawsze będę pamiętać, że właśnie tam zaczęła się moja przygoda z pisaniem, prowadzeniem bloga i prezentacjami na konferencjach. Wiele podziękowań kieruję też zbiorczo do swoich byłych współpracowników za mnóstwo motywacji i zachęt poprzez dopytywanie się o postępy prac nad książką.

Chciałbym też podziękować wszystkim użytkownikom serwisu Twitter, którzy odpowiadali na moje ankiety i pytania związane z książką, dając mi wskazówki, co jest, a co nie jest ciekawe, przydatne i cenne z punktu widzenia użytkowników .NET.

Na koniec chciałbym zbiorczo podziękować całej mojej rodzinie i wszystkim przyjaciołom, którym brakowało mnie w czasie, gdy pracowałem nad tą książką.

Przedmowa

Gdy dołączyłam do zespołu Common Language Runtime (środowisko uruchomieniowe dla .NET) ponad 10 lat temu, nie miałam pojęcia, że element zwany odśmiecaniem pamięci (Garbage Collector) stanie się tym, czemu będę poświęcać większość swojego czasu w dalszym zawodowym życiu. Wśród kilku pierwszych osób, z którymi współpracowałam w tym zespole, był Patrick Dussud, który był zarówno architektem, jak i programistą CLR GC od jego powstania. Po kilku miesiącach przekazał mi pałeczkę i stałam się następną osobą poświęcającą swój czas w zespole CLR głównie na rozwijanie GC.

Tak zaczęła się moja przygoda z GC. Wkrótce odkryłam, jak fascynujący jest świat odśmiecania pamięci – byłam zdumiona złożonymi i rozległymi wyzwaniami związanymi z GC i bardzo podobało mi się znajdowanie dla nich sprawnych rozwiązań. W miarę jak środowisko CLR było używane w coraz większej liczbie scenariuszy przez coraz więcej użytkowników, a pamięć stanowiła jeden z najważniejszych aspektów wydajnościowych, pojawiały się nowe wyzwania w obszarze zarządzania pamięcią. Gdy zaczynałam, rzadko można było spotkać stertę GC o rozmiarze 200MB; obecnie sterta o wielkości 20GB nie jest wyjątkiem. Kilka z najbardziej obciążonych procesów na świecie jest obecnie obsługiwanych przez CLR. Sposoby lepszego obsługiwanie pamięci w ich przypadku stanowią fascynujący problem.

W roku 2015 otworzyliśmy kod źródłowy CoreCLR. Gdy to zostało ogłoszone, członkowie społeczności pytali, czy źródła GC będą dostępne w repozytorium CoreCLR – co nie było takie oczywiste, gdyż nasza implementacja GC zawierała wiele innowacyjnych mechanizmów i zasad. Odpowiedź była twierdząca, a był to ten sam kod GC, który wykorzystywaliśmy w podstawowym środowisku CLR. To przyciągnęło wielu ciekawskich. Rok później byłam zachwycona, gdy dowiedziałam się, że jeden z naszych klientów zamierza napisać książkę ściśle dotyczącą naszej implementacji GC. Gdy pracownik z naszego polskiego biura zapytał mnie, czy mogłabym zrecenzować książkę Konrada, oczywiście się zgodziłam.

Gdy otrzymałam rozdziały książki od Konrada, stało się dla mnie jasne, że pilnie przestudiował on nasz kod GC. Byłam pod wrażeniem szczegółowości omawianych zagadnień. Można oczywiście samemu zbudować kod CoreCLR i przechodzić przez

niego krok po kroku. Książka ta z pewnością znacznie to wszystkim ułatwi. Ponieważ ważną część czytelników tej książki stanowią użytkownicy GC, Konrad zamieścił mnóstwo materiałów pozwalających lepiej zrozumieć zachowanie GC oraz wiele wzorców programowania pomagających wydajniej korzystać z GC. Na początku książki można też znaleźć podstawowe informacje na temat pamięci, a pod koniec książki omówienie wykorzystania pamięci w różnych bibliotekach. Uważam, że w książce panuje idealna równowaga pomiędzy wprowadzeniem do GC, omówieniem wewnętrznych mechanizmów i przykładami zastosowań.

Książka ta jest dla każdego, kto korzysta z .NET i komu zależy na wydajnym działaniu pamięci oraz dla każdego, kto po prostu jest ciekaw działania .NET GC i chce zrozumieć dokładnie działanie tej implementacji. Mam nadzieję, że czytanie tej książki sprawi wszystkim tyle radości, co mnie jej recenzowanie.

Maoni Stephens

Lipiec 2018

Wprowadzenie

W informatyce zawsze mieliśmy do czynienia z pamięcią – od kart perforowanych poprzez taśmy magnetyczne aż do dzisiejszych, skomplikowanych układów DRAM. Pamięć zawsze będzie obecna, w przyszłości być może w formie jakichś układów holograficznych lub jeszcze bardziej zadziwiających technologii, których jeszcze nie jesteśmy sobie w stanie wyobrazić. Oczywiście pamięć istnieje nie bez powodów. Dobrze wiadomo, że programy komputerowe to połączenie algorytmów i struktur danych. Bardzo lubię to określenie. Chyba każdy co najmniej raz słyszał o książce *Algorytmy + Struktury danych = Programy* napisanej przez Niklausa Wirtha (Prentice Hall, 1976), gdzie określenie to powstało.

Od samego początku inżynierii oprogramowania zarządzanie pamięcią było zagadnieniem znanym ze swojego znaczenia. Od pierwszych urządzeń komputerowych inżynierowie musieli myśleć o przechowywaniu algorytmów (kodu programu) i struktur danych (danych programu). Zawsze było ważne, jak i gdzie te dane są ładowane i przechowywane na później.

W tym aspekcie inżynieria oprogramowania i zarządzanie pamięcią zawsze były nieodłącznie powiązane. Sądzę też, że tak będzie zawsze. Pamięć jest ograniczonym zasobem i zawsze takim będzie. Dlatego do pewnego stopnia pamięć zawsze będzie zaprzątać umysły przyszłych programistów. Jeśli pewien zasób jest ograniczony, zawsze może pojawić się jakiś błąd lub nieprawidłowy sposób użycia, który doprowadzi do wyczerpania tego zasobu. Pamięć nie jest tutaj wyjątkiem.

To powiedziawszy, z pewnością jest jedna rzecz dotycząca zarządzania pamięcią, która się stale zmienia – ilość. Pierwsi programiści albo inżynierowie oprogramowania musieli zwracać uwagę na każdy pojedynczy bit w swoich programach. Później mieli do dyspozycji kilobajty pamięci. Z każdą dekadą liczby te rosły i dzisiaj żyjemy w czasach gigabajtów, gdy terabajty i petabajty czekają już grzecznie na progu na swoją kolej. W miarę jak rozmiar pamięci rośnie, maleją też czasy dostępu, co umożliwia przetwarzanie wszystkich tych danych w zadowalającym czasie. Choć jednak możemy stwierdzić, że pamięć jest szybka, proste algorytmy zarządzania pamięcią, próbujące przetwarzać gigabajty danych bez żadnych optymalizacji i bardziej skomplikowanego dostrajania, nie są wystarczające. Dzieje się tak głównie dlatego, że czasy dostępu do pamięci zmniejszają się wolniej niż poprawia się moc obliczeniowa procesorów, które

z niej korzystają. Trzeba zwracać szczególną uwagę, aby nie wprowadzać wąskich gardeł do zarządzania pamięcią, które ograniczałyby moc obecnych procesorów.

To sprawia, że zarządzanie pamięcią nie tylko jest bardzo istotne, ale też stanowi niezwykle fascynującą część informatyki. Jeszcze lepiej jest dzięki automatycznemu zarządzaniu pamięcią. Nie wystarczy tylko powiedzieć „zwolnijmy nieużywane obiekty”. Co, jak i kiedy – te proste aspekty zarządzania pamięcią sprawiają, że jest to stale trwający proces poprawiania starych i wymyślania nowych algorytmów. Niezliczone prace naukowe i doktoraty zajmują się sposobami automatycznego zarządzania pamięcią w możliwie optymalny sposób. Wydarzenia, takie jak ISMM (International Symposium on Memory Management – międzynarodowe sympozjum dotyczące zarządzania pamięcią) pokazują co roku, jak wiele dzieje się w tej dziedzinie w zakresie odświeczania pamięci, alokacji dynamicznej oraz interakcji z środowiskami uruchomieniowymi, kompilatorami i systemami operacyjnymi. Później badania akademickie zmieniają się w komercyjne i otwarte produkty, z których korzystamy w codziennej pracy.

.NET jest idealnym przykładem środowiska zarządzanego, gdzie cała ta złożoność jest ukryta pod spodem i jest dostępna dla programistów w postaci przyjemnej, gotowej do użycia platformy. W istocie możemy korzystać z niej, nie mając żadnej świadomości wewnętrznej złożoności, co jest świetnym osiągnięciem .NET. Jednakże im bardziej nasz program będzie wrażliwy na problemy wydajnościowe, tym mniej możliwe będzie obejście się bez wiedzy na temat wewnętrznego sposobu działania pamięci. Osobiście uważam też, że dobrze jest wiedzieć, jak działają rzeczy, z których codziennie korzystamy!

Napisałem tę książkę w taki sposób, jak chciałbym ją przeczytać wiele lat temu – gdy zaczynałem swoją przygodę w dziedzinie wydajności .NET i diagnostyki. Książka ta nie zaczyna się więc od typowego wprowadzenia do pojęć sterty i stosu oraz opisu wielu generacji. Zaczynam natomiast od najważniejszych ogólnych podstaw zarządzania pamięcią. Innymi słowy, próbowałem napisać tę książkę w sposób pozwalający dobrze poczuć ten bardzo ciekawy temat, a nie tylko przedstawić mechanizm odświeczania pamięci w .NET i sposoby jego działania. Odpowiedzi na pytania co, jak, a przede wszystkim – dlaczego – powinny pomóc czytelnikom w zrozumieniu wewnętrznego działania zarządzania pamięcią w .NET. Wszystkie przyszłe lektury dotyczące tego tematu powinny być bardziej zrozumiałe dla czytelników po przeczytaniu tej książki. Próbuję przekazać też wiedzę nieco bardziej ogólną niż tylko związaną z .NET, zwłaszcza w pierwszych dwóch rozdziałach. Prowadzi to do głębszego zrozumienia tematu, który może mieć często zastosowanie również w innych zadaniach inżynierii oprogramowania (dzięki zrozumieniu algorytmów, struktur danych i po prostu dobrych zasad inżynierskich).

Chciałem napisać tę książkę w sposób przyjazny dla każdego programisty .NET. Bez względu na stopień zaawansowania, każdy powinien tutaj znaleźć coś ciekawego dla siebie. Choć zaczniemy od podstaw, młodzi programiści szybko będą mieli okazję zagłębić

się w wewnętrzne szczegóły platformy .NET. Zaawansowanych programistów bardziej zaciekawia szczegóły implementacyjne. Przede wszystkim niezależnie od doświadczenia każdy powinien być w stanie skorzystać z przedstawionych praktycznych przykładów kodu i diagnostyki problemów.

Wiedza uzyskana z tej książki powinna pomóc czytelnikom w pisaniu lepszego kodu – bardziej wydajnego i świadomego zarządzania pamięcią, z wykorzystaniem odpowiednich funkcji bez obawy przed ich niezrozumieniem. Będzie to też prowadzić do lepszej wydajności i skalowalności tworzonych aplikacji – im bardziej zorientowany na pamięć będzie pisany kod, tym mniej będzie podatny na wąskie gardła i nieoptymalne korzystanie z zasobów. Mam nadzieję, że po przeczytaniu tej książki każdy uzna podtytuł „Lepszy kod, wydajność i skalowalność” za uzasadniony.

Mam też nadzieję, że wszystko to sprawi, iż ta książka będzie bardziej ogólna niż prosty opis obecnego stanu platformy .NET i jej wewnętrznych mechanizmów. Niezależnie od tego, jak będą ewoluować przyszłe wersje platformy .NET, wierzę, że większość wiedzy wyniesionej z tej książki pozostanie przydatna przez długi czas. Nawet jeśli pewne szczegóły implementacyjne się zmieniają, czytelnik będzie je w stanie łatwo zrozumieć, dzięki wiedzy uzyskanej z tej książki. Ogólne zasady po prostu nie będą się zmieniać tak szybko. Życzę wszystkim przyjemnej podróży przez ogromny i ciekawy temat automatycznego zarządzania pamięcią!

Chciałbym też podkreślić kilka kwestii, które nie są szczególnie obecne w tej książce. Temat zarządzania pamięcią, choć wydaje się na pierwszy rzut oka bardzo wąski i specjalistyczny, jest zaskakująco szeroki. Choć poruszam mnóstwo zagadnień, to czasami nie są one przedstawiane tak szczegółowo, jakbym chciał (ze względu na brak miejsca). Nawet przy takich ograniczeniach, książka ta ma ponad 1000 stron! Do pominiętych tematów należą na przykład wyczerpujące odwołania do innych środowisk zarządzanych (takich jak Java, Python czy Ruby). Przepraszam też fanów F# za tak nieliczne odwołania do tego języka. Po prostu nie starczyło stron na porządny opis tych zagadnień, a nie chciałem poruszać pewnych tematów bez wyczerpującego ich omówienia. Chciałbym też móc poświęcić więcej uwagi środowisku Linux, ale jest to zagadnienie tak świeże i bez wielu dostępnych narzędzi, że poruszam je tylko ogólnie w rozdziale 3 (i całkowicie pomijam świat macOS z tych samych powodów). Oczywiście pominąłem też dużą część innych zagadnień wydajnościowych w .NET niezwiązanych bezpośrednio z pamięcią, jak wielowątkowość.

Choć starałem się jak najlepiej przedstawić praktyczne zastosowanie omawianych zagadnień i technik, to nie zawsze jest to możliwe bez zrobienia tego w całkiem wyczerpujący sposób. Praktycznych zastosowań jest po prostu zbyt dużo. Raczej oczekuję od czytelnika, że będzie czytał uważnie, poświęci czas na przemyślenie danego zagadnienia i zastosuje zdobytą wiedzę w swojej codziennej pracy. Wystarczy zrozumieć, jak coś działa, aby być w stanie z tego korzystać!

Odnosi się to w szczególności do tzw. scenariuszy. Warto zauważyć, że wszystkie scenariusze zawarte w tej książce mają charakter poglądowy. Ich kod został ograniczony do niezbędnego minimum, aby łatwiej pokazać główną przyczynę pojedynczego problemu. W rzeczywistości może istnieć wiele różnych powodów zaobserwowanego błędnego działania programu (tak jak na wiele sposobów można zaobserwować wycieki pamięci). Scenariusze zostały przygotowane w sposób pomagający zilustrować takie problemy na pojedynczym przykładzie, ponieważ oczywiście nie da się zawrzeć wszystkich możliwych przyczyn danego problemu w jednej książce. Co więcej, w rzeczywistych scenariuszach sytuacja będzie zagmatwana przez szum powodowany dodatkowymi danymi oraz wybór ślepych ścieżek badania problemu. Często nie istnieje tylko jeden sposób na rozwiązanie przedstawionych problemów, ale wiele sposobów na znalezienie podstawowej przyczyny podczas analizy problemów. Takie rozwiązywanie problemów staje się mieszkanką zadania czysto inżynierskiego z odrobiną sztuki wspartej własną intuicją. Warto też zauważyć, że scenariusze odwołują się do siebie nawzajem, aby nie powtarzać za każdym razem tych samych kroków, rysunków i opisów.

W szczególności powstrzymywałem się w tej książce od przywoływania różnych przypadków i źródeł problemów specyficznych dla konkretnych technologii. Są one po prostu... zbyt specyficzne dla poszczególnych technologii. Gdybym pisał tę książkę 10 lat temu, prawdopodobnie musiałbym wymienić różne scenariusze typowe dla wycieków pamięci w ASP.NET WebForms i WinForms. Kilka lat temu? ASP.NET MVC, WPF, WCF, WF,... Obecnie? ASP.NET Core, EF Core, Azure Functions i co jeszcze? Mam nadzieję, że widać, o co chodzi. Taka wiedza zbyt szybko staje się przestarzała. Książka naszpikowana przykładami wycieków pamięci w WCF mało kogo by dzisiaj interesowała. Jestem wielkim zwolennikiem powiedzenia: „Daj człowiekowi rybę; nakarmisz go na jeden dzień. Naucz człowieka łowić ryby; nakarmisz go na całe życie”. Tak więc cała wiedza zawarta w tej książce, wszystkie scenariusze, mają uczyć, jak „łować ryby”. Wszystkie problemy, niezależnie od bazowej, specyficznej technologii, można diagnozować w taki sam sposób, jeśli zastosuje się odpowiednią wiedzę i podejście.

Wszystko to sprawia też, że czytanie tej książki jest dość wymagające, gdyż czasami jest pełna szczegółów i przekazuje dość przytłaczającą ilość informacji. Pomimo tego wszystkiego zachęcam do dogłębnego i powolnego czytania oraz opierania się pokusie pobieżnego przeglądania niektórych fragmentów. Na przykład, aby w pełni czerpać korzyści z tej książki, należy dokładnie zapoznać się z pokazanym kodem i przedstawionymi rysunkami (nie wystarczy tylko rzucić na nie okiem i uznać je za oczywiste).

Żyjemy w świetnych czasach środowiska uruchomieniowego CoreCLR z otwartym kodem źródłowym. Daje nam to dużo większe możliwości zrozumienia działania środowiska CLR. Nie ma tajemnic i zgadywania. Wszystko dostępne jest w kodzie, który można przeczytać i zrozumieć. Moje badania działania środowiska uruchomieniowego

są więc w znacznym stopniu oparte na kodzie GC w CoreCLR (kod ten jest też wykorzystywany przez platformę .NET Framework). Niezliczone dni i tygodnie spędziłem na analizowaniu tej ogromnej ilości dobrej inżynierskiej roboty. Myślę, że jest to świetny kod i sądzę, że wiele osób również chciałoby przebadać słynny plik `gc.cpp`, składający się z dziesiątek tysięcy wierszy kodu. Jest to jednak związane z bardzo stromą krzywą uczenia się. Żeby w tym pomóc, często pozostawiam wskazówki, od czego zacząć badanie kodu CoreCLR w odniesieniu do opisywanych tematów. Zachęcam gorąco do jeszcze głębszego zapoznania się z sugerowanymi przeze mnie fragmentami pliku `gc.cpp`!

Po przeczytaniu tej książki, czytelnik powinien być w stanie:

- Pisać wydajny kod uwzględniający zarządzanie pamięcią w .NET. Choć przedstawione przykłady są w języku C#, to myślę, że nabyta wiedza i poznany zestaw narzędzi pozwolą zastosować je też w językach F# i VB.NET.
- Diagnozować typowe problemy związane z zarządzaniem pamięcią w .NET. Ponieważ większość technik opiera się na danych ETW/LLTng i rozszerzeniu SOS, można je stosować w systemach Windows i Linux (bardziej zaawansowane narzędzia dostępne są w systemie Windows).
- Zrozumieć, jak działa CLR w obszarze zarządzania pamięcią. Poświęciłem sporo uwagi na wyjaśnienie nie tylko tego, jak pewne rzeczy działają, ale też, dlaczego.
- Czytać z pełnym zrozumieniem zgłoszenia wielu ciekawych problemów dotyczących C# i środowiska uruchomieniowego CLR w serwisie GitHub, a nawet samemu brać udział w dyskusji.
- Czytać kod GC w CoreCLR (zwłaszcza w pliku `gc.cpp`) ze zrozumieniem pozwalającym na dalsze badania i dociekania.
- Czytać z pełnym zrozumieniem informacje na temat odśmiecania pamięci i zarządzania pamięcią w różnych środowiskach, takich jak Java, Python lub Go.

Jeśli chodzi o samą zawartość tej książki, to przedstawia się ona następująco. Rozdział 1 jest bardzo ogólnym, teoretycznym wprowadzeniem do zarządzania pamięcią prawie bez jakichkolwiek szczególnych odwołań do .NET. Rozdział 2 podobnie jest ogólnym wprowadzeniem do zarządzania pamięcią na poziomie sprzętu i systemu operacyjnego. Oba rozdziały można traktować jako ważne, ale opcjonalne wprowadzenie. Stanowią one przydatne, szersze spojrzenie na to zagadnienie, które będzie przydatne w pozostałych częściach książki. Choć oczywiście usilnie zachęcam do ich przeczytania, to można je pominąć, jeśli ktoś się spieszy lub jest zainteresowany jedynie najbardziej praktycznymi tematami związanymi z .NET. Nawet zaawansowanych czytelników, którzy sądzą, że dobrze znają zagadnienia z tych dwóch pierwszych rozdziałów, zachęcam do ich przeczytania. Próbowałem w nich zawrzeć nie tylko oczywiste informacje.

Rozdział 3 jest poświęcony wyłącznie pomiarom i różnym narzędziom (niektóre z nich są bardzo często wykorzystywane w dalszych częściach książki). Jest to lektura zawierająca głównie listę narzędzi i opis ich używania. Jeśli kogoś ciekawi głównie teoretyczna część tej książki, może jedynie pobieżnie przejrzeć ten rozdział. Z drugiej strony, jeśli ktoś planuje intensywnie wykorzystywać wiedzę z tej książki w diagnostyce problemów, to prawdopodobnie będzie często wracać do tego rozdziału.

Rozdział 4 jest pierwszym, w którym zaczynamy intensywnie omawiać .NET, ale nadal w dość ogólny sposób, co pozwoli nam zrozumieć pewne istotne wewnętrzne mechanizmy, takie jak system typów .NET (w tym porównanie typów wartościowych i typów referencyjnych, internowanie łańcuchów znaków albo dane statyczne. Jeśli ktoś ma naprawdę mało czasu, może zacząć czytanie od tego rozdziału. Rozdział 5 opisuje pierwsze zagadnienie naprawdę związane z pamięcią – jak pamięć jest zorganizowana w aplikacjach .NET, wprowadzając pojęcie sterty małych obiektów i sterty dużych obiektów, a także segmentów. Rozdział 6 zagłębia się dalej w wewnętrzne mechanizmy związane z pamięcią i jest poświęcony alokowaniu pamięci. Zaskakujące, że całkiem duży rozdział może być poświęcony tak prostemu teoretycznie zagadnieniu. Ważną i dużą częścią tego rozdziału są opisy różnych źródeł alokacji w kontekście ich unikania.

Rozdziały od 7 do 10 są najważniejszymi częściami książki opisującymi, jak działa odśmiecanie pamięci w .NET i zawierają praktyczne przykłady i uwarunkowania wynikające z tej wiedzy. Aby nie przytłoczyć czytelnika nadmiarem informacji naraz, rozdziały te opisują najprostszą odmianę odśmiecania pamięci – tak zwane odśmiecanie niewspółbieżne w trybie stacji roboczej. Z kolei rozdział 11 jest poświęcony opisowi wszystkich pozostałych odmian z wyczerpującymi rozważaniami nad wyborem odpowiedniej z nich. Rozdział 12 kończy część książki poświęconą odśmiecaniu pamięci, opisując trzy ważne mechanizmy: finalizację, obiekty sprzątające po sobie i słabe odwołania.

Trzy ostatnie rozdziały stanowią „zaawansowaną” część książki w tym sensie, że wyjaśniają, jak wszystko działa poza podstawową częścią zarządzania pamięcią w .NET. Rozdział 13 wyjaśnia na przykład zagadnienie wskaźników zarządzanych i zajmuje się głębiej strukturami (w tym niedawno dodanymi strukturami typu ref). Rozdział 14 poświęca wiele uwagi typom i technikom zyskującym ostatnio coraz więcej popularności, takim jak typy `Span<T>` i `Memory<T>`. Część tego rozdziału jest poświęcona niezbyt dobrze znanemu tematowi projektowania zorientowanego na dane, a kilka słów poświęcono też nadchodzącym funkcjom języka C# (takim jak typy referencyjne dopuszczające wartość null i potoki). Rozdział 15 (ostatni) opisuje różne sposoby sterowania odśmiecaniem pamięci i jego monitorowania z poziomu kodu, w tym interfejs API klasy GC, hosting środowiska CLR oraz bibliotekę ClrMD.

Większość programów z tej książki jest dostępna w repozytorium GitHub pod adresem <https://github.com/Apress/pro-.net-memory>. Jest ono zorganizowane według rozdziałów, a większość z nich zawiera dwa rozwiązania: jedno dla przeprowadzanych

xxiv

testów i jedno dla pozostałych programów. Warto zauważyć, że choć dołączone projekty zawierają listingi programów z książki, to zwykle jest w nich więcej kodu. Jeśli ktoś chce wykorzystać określony program z książki (lub z nim poeksperymentować), najłatwiej po prostu wyszukać numer danego listingu i go wypróbować. Zachęcam też do przejrzenia całych projektów dla określonych tematów, aby lepiej je zrozumieć.

Chciałbym też tu wspomnieć o kilku ważnych konwencjach stosowanych w tej książce. Najważniejszą jest rozróżnienie pomiędzy dwoma głównymi pojęciami wykorzystywanymi w całej książce:

- Odśmianie pamięci (GC – garbage collection) – ogólnie rozumiany proces odzyskiwania niepotrzebnej już pamięci.
- Mechanizm odśmiania pamięci (the GC – the garbage collector) – konkretny mechanizm realizujący odśmianie pamięci, zwłaszcza w kontekście mechanizmu odśmiania pamięci w .NET.

Ta książka stanowi zamkniętą całość i nie odwołuje się do wielu innych materiałów lub książek. Oczywiście jest wiele innych publikacji, z których można czerpać podobną wiedzę i wiele razy musiałbym się odwoływać do różnych źródeł. Zamiast tego pozwolę sobie na wymienienie wybranej przeze mnie listy proponowanych książek i artykułów, stanowiących uzupełniające źródło wiedzy:

- Książka *Pro .NET Performance* – Sasha Goldshtein, Dima Zurbalev i Ido Flatow (Apress, 2012).
- Książka *CLR via C#* – Jeffrey Richter (Microsoft Press, 2012).
- *Writing High-Performance .NET Code* – Ben Watson (Ben Watson, 2014).
- *Advanced .NET Debugging* – Mario Hewardt (Addison-Wesley Professional, 2009).
- *.NET IL Assembler* – Serge Lidin (Microsoft Press, 2012)
- *Shared Source CLI Essentials* – David Stutz (O'Reilly Media, 2003).
- Dokumentacja otwartego kodu środowiska uruchomieniowego .NET: „Book Of The Runtime”, opracowywana równolegle z samym środowiskiem uruchomieniowym, dostępna pod adresem <https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md>.

Wiele cennych informacji można też znaleźć w różnych blogach i artykułach dostępnych w Internecie. Zamiast wymieniać je na tych stronach, wskażę na świetne repozytorium <https://github.com/adamsitnik/awesome-dot-net-performance> utrzymywane przez Adama Sitnika.

Rozdział 1

Podstawowe pojęcia

Zacznijmy od prostego, ale bardzo ważnego pytania. Kiedy powinniśmy przejmować się zarządzaniem pamięcią w .NET, skoro jest w całości zautomatyzowane? Czy w ogóle powinniśmy sobie zawracać tym głowę? Jak można by oczekiwać po tym, że napisałem tę książkę – gorąco zachęcam do pamiętania o pamięci w każdej sytuacji programistycznej. Jest to po prostu kwestia naszego profesjonalizmu. Konsekwencja tego, jak wykonujemy swoją pracę. Czy staramy się wykonywać ją jak najlepiej, czy po prostu ją wykonujemy? Jeśli dbamy o jakość swojej pracy, nie powinno wystarczać nam, aby efekty naszej pracy po prostu działały. Powinniśmy się też przejmować tym, jak działają? Czy tworzone oprogramowanie jest optymalne z punktu widzenia wykorzystania procesora i pamięci? Czy jest łatwe w utrzymaniu i testowaniu, otwarte na rozszerzenia, ale zamknięte na modyfikacje? Czy nasz kod spełnia zasady SOLID? Wierzę, że wszystkie te pytania odróżniają początkujących programistów od bardziej zaawansowanych i doświadczonych. Ci pierwsi są głównie zainteresowani wykonaniem zadania i nie przejmują się zbyt wiele powyższymi pozafunkcyjnymi aspektami swojej pracy. Ci drudzy są na tyle doświadczeni, że biorą też pod uwagę jakość swojej pracy. Sądzę, że każdy chciałby do tego dążyć. Nie jest to oczywiście sprawa trywialna. Pisanie eleganckiego kodu, pozbawionego błędów i spełniającego wszystkie możliwe wymagania pozafunkcyjne jest naprawdę trudne.

Czy jednak takie dążenie do doskonałości ma być jedynym wymogiem zdobywania większej wiedzy na temat zarządzania pamięcią w .NET? Uszkodzenia pamięci przejawiające się jako wyjątki `AccessViolationException` są niezwykle rzadkie¹. Może się też wydawać, że podobnie jest z niekontrolowanym wzrostem wykorzystania pamięci. Czy mamy się więc czym przejmować? Dzięki wyrafinowanej implementacji środowiska

¹ Wyjątek `AccessViolationException` albo inne uszkodzenia sterty mogą być często wzbudzone przez automatyczne zarządzanie pamięcią, nie dlatego, że stanowi ono ich przyczynę, ale ponieważ jest najcięższym składnikiem środowiska związanym z pamięcią. Dlatego ma największą możliwość ujawnienia wszelkich niespójnych stanów pamięci.

uruchomieniowego .NET na szczęście nie musimy poświęcać wiele uwagi aspektom pamięciowym. Z drugiej jednak strony, gdy zajmujemy się analizowaniem problemów wydajnościowych dużych aplikacji opartych na .NET, problemy z wykorzystaniem pamięci zawsze znajdują się wysoko na liście. Czy z długoterminowego punktu widzenia problemem jest, gdy mamy wyciek pamięci po wielu dniach ciągłego działania aplikacji? W Internecie można znaleźć zabawny mem dotyczący nienaprawionego wycieku pamięci w oprogramowaniu pewnego pocisku raketowego, ponieważ pamięć była wystarczająca do czasu, aż pocisk osiągnął swój cel. Czy nasz system jest takim jednorazowym pociskiem? Czy zdajemy sobie sprawę z tego, czy zautomatyzowane zarządzanie pamięcią wprowadza duże obciążenie dla naszej aplikacji, czy nie? Może moglibyśmy wykorzystywać tylko dwa serwery zamiast dziesięciu? Co więcej nawet w czasach bezserwerowego przetwarzania w chmurze nie obojędźmy się bez pamięci. Jednym z przykładów mogą być funkcje Azure Functions, za które opłaty wyliczane są w oparciu o jednostkę nazywaną „gigabajtosekundami” (GB-s). Są one obliczane poprzez pomnożenie średniego rozmiaru zajętej pamięci w gigabajtach przez czas w sekundach potrzebny do wykonania danej funkcji. Zużycie pamięci przekłada się więc bezpośrednio na wydawane pieniądze.

W każdym razie zaczynamy sobie zdawać sprawę, że nie mamy pojęcia, gdzie szukać prawdziwej przyczyny problemów i przydatnych pomiarów. W tym miejscu zaczynamy zdawać sobie sprawę, że warto zrozumieć wewnętrzne mechanizmy naszych aplikacji i bazowego środowiska uruchomieniowego.

Aby dogłębnie zrozumieć zarządzanie pamięcią w .NET, najlepiej zacząć od samego początku. Niezależnie od tego, czy ktoś jest nowicjuszem, czy zaawansowanym programistą. Zapraszam do wspólnego przejścia przez teoretyczne wprowadzenie zawarte w tym rozdziale. Da to pewien poziom wiedzy i zrozumienia podstawowych pojęć, na którym będzie bazować pozostała część książki. Aby nie była to tylko nudna teoria, czasami będę odwoływać się do konkretnych technologii. Przyjrzymy się też nieco historii tworzenia oprogramowania. Wiąże się to dobrze z rozwojem pojęć dotyczących zarządzania pamięcią. Zauważymy też dość ciekawe fakty, które, mam nadzieję, okażą się też interesujące dla czytelnika. Poznawanie historii danego zagadnienia jest zawsze najlepszym sposobem na uzyskanie szerszej perspektywy.

Nie ma się jednak czego obawiać. Nie jest to książka historyczna. Nie będę opisywał biografii wszystkich inżynierów zajmujących się rozwijaniem algorytmów odświeżania pamięci od 1950 roku. Nie będzie też potrzebna wiedza o zamierzchłej historii. Mam jednak nadal nadzieję, że dla wszystkich ciekawe będzie poznanie, jak to zagadnienie się rozwijało i gdzie jest obecnie na linii czasu. Pozwoli nam to też na porównanie podejścia zastosowanego w .NET z wieloma innymi językami i środowiskami uruchomieniowymi.

Terminy związane z pamięcią

Zanim zaczniemy, dobrze będzie się przyjrzeć kilku bardzo ważnym definicjom, bez których trudno sobie wyobrazić omawianie tematu pamięci:

- *Bit* – jest to najmniejsza jednostka informacji wykorzystywana w informatyce. Reprezentuje dwa możliwe stany, odpowiadające zwykle wartościom liczbowym 1 i 0 lub wartościom logicznym prawda i fałsz. W rozdziale 2 krótko wspomnimy, w jaki sposób nowoczesne komputery przechowują pojedyncze bity. W przypadku większej wartości liczbowej potrzebna jest kombinacja wielu bitów do zakodowania jej w postaci liczby binarnej, jak wyjaśniono dalej. Przy określaniu rozmiaru danych bity oznaczane są małą literą b.
- *Liczba binarna* – całkowita wartość liczbową reprezentowaną przez sekwencję bitów. Każdy kolejny bit określa składową sumę danej wartości, będącą kolejną potęgą liczby 2. Na przykład, aby przedstawić liczbę 5, wykorzystujemy trzy kolejne bity o wartościach 1, 0 i 1, ponieważ $1 \times 1 + 0 \times 2 + 1 \times 4$ równa się 5. Liczba binarna n-bitowa może reprezentować maksymalną wartość $2^n - 1$. Zwykle dodatkowy bit przeznaczony jest na reprezentowanie znaku wartości, aby kodować zarówno liczby dodatnie, jak i ujemne. Istnieją też inne, bardziej złożone sposoby kodowania wartości liczbowych w postaci binarnej, zwłaszcza dla liczb zmiennoprzecinkowych.
- *Kod binarny* – zamiast wartości liczbowych, sekwencja bitów może reprezentować określony zestaw innych danych – takich jak znaki tekstu. Każda sekwencja bitów jest przypisana do określonych danych. Najbardziej podstawowym i przez wiele lat najpopularniejszym kodem był kod ASCII, który wykorzystuje 7-bitowe sekwencje do reprezentowania tekstu i innych znaków. Istnieją też inne ważne kody binarne takie jak *opcodes* (kody operacji), które kodują instrukcje nakazujące komputerowi, co ma robić.
- *Bajt* – historycznie była to sekwencja bitów potrzebna do zakodowania pojedynczego znaku tekstu przy użyciu określonego kodu binarnego. Najbardziej typowym rozmiarem bajtu jest bajt 8-bitowy, choć zależy to od architektury komputera i może się różnić pomiędzy różnymi systemami. Ze względu na tę niejednoznaczność istnieje bardziej precyzyjny termin *oktet*, który oznacza jednostkę danych dokładnie 8-bitowej długości. W każdym razie standardem de facto jest rozumienie bajtu jako wartości o długości 8 bitów i w takiej formie termin ten stał się niekwestionowanym standardem definiowania rozmiarów danych. Obecnie raczej nie spotyka się innych architektur niż standardowa architektura z 8-bitowymi bajtami. Przy określaniu rozmiaru danych bajty oznaczane są wielką literą B.

Określając rozmiar danych, korzystamy z typowych przedrostków oznaczających rząd wielkości. Jest to powodem stałego zamieszania i nieporozumień, które warto w tym momencie wyjaśnić. Powszechnie używane terminy, takie jak kilo, mega i giga oznaczają kolejne potęgi tysiąca. Jeden *kilo* oznacza 1000 (i oznaczamy to w skrócie małą literą k), jeden *mega* oznacza milion (wielka litera M), itd. Z drugiej strony czasami popularne jest podejście wyrażające rzędy wielkości jako kolejne potęgi 1024. W takich przypadkach używamy jako przedrostka *kibi*, co oznacza 1024 (oznaczane w skrócie jako Ki), jeden *mebi* to $1024 \cdot 1024$ (oznaczane w skrócie jako Mi), jeden *gibi* (Gi) to $1024 \cdot 1024 \cdot 1024$, itd. Wprowadza to powszechnie występującą niejednoznaczność. Gdy ktoś mówi o 1 „gigabajcie”, w zależności od kontekstu możemy sądzić, że chodzi o miliard bajtów (1 GB) albo o 1024^3 bajtów (1 GiB). W praktyce niewiele osób przejmuje się dokładnym stosowaniem tych przedrostków. Bardzo powszechne obecnie jest określanie rozmiarów modułów pamięciowych w komputerach jako gigabajty (GB), gdy w rzeczywistości chodzi o gibibajty (GiB) albo odwrotnie w przypadku pojemności dysków twardych. Nawet norma JEDEC Standard 100B.01 „Terms, Definitions, and Letter Symbols for Microcomputers, Microprocessors, and Memory Integrated Circuits” (Terminy, definicje i symbole literowe dla mikrokomputerów, mikroprocesorów i zintegrowanych układów pamięciowych) odwołuje się do powszechnego użycia liter K, M i G do oznaczania potęg 1024. W takich sytuacjach musimy odwoływać się do zdrowego rozsądku, aby odpowiednio interpretować te prefiksy w danym kontekście.

Obecnie jesteśmy bardzo przyzwyczajeni do terminów, takich jak pamięć RAM albo pamięć trwała, które są instalowane w naszych komputerach. Nawet inteligentne zegarki są obecnie wyposażane w 8 GiB pamięci RAM. Łatwo można zapomnieć, że pierwsze komputery nie były wyposażone w takie luksusy. Można by powiedzieć, że w nic nie były wyposażone. Przyjrzenie się krótkiej historii rozwoju komputerów pozwoli nam inaczej spojrzeć na samą pamięć. Zaczniemy od początku.

Powinniśmy mieć na uwadze, że można się spierać, które urządzenie należałoby nazywać „całkiem pierwszym komputerem”. Podobnie bardzo trudno jest wskazać jedyne „wynalazcę komputera”. Jest to kwestia samej definicji, czym tak naprawdę jest „komputer”. Zamiast więc zaczynać niekończące się dyskusje, co było pierwsze i kto był pierwszy, przyjrzyjmy się kilku najstarszym maszynom i temu, co oferowały programistom, choć samo słowo *programista* miało się pojawić wiele lat później. Na początku byli oni zwani *koderami* albo *operatorami*.

Należy podkreślić, że maszyny, które można zdefiniować jako pierwsze komputery, nie były w pełni elektroniczne, ale elektromechaniczne. Z tego powodu były bardzo powolne i pomimo ogromnych rozmiarów oferowały bardzo niewiele. Pierwszy z tych programowalnych elektromechanicznych komputerów został zaprojektowany w Niemczech przez Konrada Zuse i nazywał się komputerem Z3. Ważył jedną tonę! Jedna

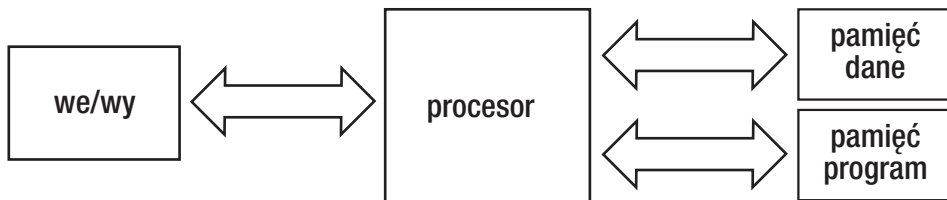
operacja dodawania zabierała około sekundy, a pojedyncze mnożenie zabierało 3 sekundy! Zbudowany był z 2000 elektromechanicznych przekaźników i oferował jednostkę arytmetyczną potrafiącą wykonywać jedynie operacje dodawania, odejmowania, mnożenia, dzielenia i wyciągania pierwiastka kwadratowego. Jednostki arytmetycznie zawierały też dwa 22-bitowe rejestry pamięciowe wykorzystywane w obliczeniach. Komputer ten oferował też 64 komórki pamięci ogólnego przeznaczenia, z których każda miała 22 bity długości. Obecnie moglibyśmy powiedzieć, że oferował 176 bajtów wewnętrznej pamięci do przechowywania danych!

Dane były wprowadzane poprzez specjalną klawiaturę, a program był wczytywany podczas wykonywania obliczeń z perforowanej taśmy celuloidowej. Możliwość przechowywania programu w wewnętrznej pamięci komputera miała zostać zaimplementowana kilka lat później i wkrótce do tego wrócimy, choć Zuse był w pełni świadomy tego pomysłu. W kontekście niniejszej książki ważniejsze jest pytanie o dostęp do pamięci Z3. Programując Z3 mieliśmy do dyspozycji tylko dziewięć instrukcji! Jedna z nich pozwalała nam załadować wartość jednej z 64 komórek pamięci do rejestru pamięciowego jednostki arytmetycznej. Inna zapisywała wartość z powrotem w komórce pamięci. I to wszystko, jeśli chodzi o „zarządzanie pamięcią” w tym pierwszym komputerze. Choć Z3 wyprzedzał swój czas pod wieloma względami, to z powodów politycznych i wybuchu II wojny światowej jego wpływ na rozwój komputerów był pomijalny. Firma Zuse rozwijała swoją linię komputerów przez wiele lat po wojnie, a ostatnia wersja komputera Z22 została zbudowana w roku 1955.

Podczas wojny i wkrótce po niej głównymi centrami rozwoju informatyki były Stany Zjednoczone i Zjednoczone Królestwo. Jednym z pierwszych komputerów zbudowanych w Stanach Zjednoczonych był Harvard Mark I opracowany przez IBM we współpracy z Uniwersytetem Harvarda i znany też jako ASCC (Automatic Sequence Controlled Calculator – kalkulator sterowany automatycznymi sekwencjami). Również był urządzeniem elektromechanicznym, jak wspomniany wcześniej Z3. Miał ogromne rozmiary, mierzył 2,4 m wysokości, 16 m długości i niemal metr głębokości. Ważył przy tym 5 ton! Jest nazywany największą maszyną obliczeniową w historii. Budowano go przez kilka lat, a pierwsze programy uruchomiono na nim pod koniec II wojny światowej w roku 1944. Służył marynarce wojennej, ale też Johnowi von Neumannowi podczas jego prac nad pierwszą bombą atomową w ramach projektu Manhattan. Mimo takich rozmiarów oferował jedynie 72 jednostki pamięciowe dla 23-cyfrowych liczb ze znakiem. Taka jednostka zwana była *akumulatorem* – i była dedykowanym niewielkim fragmentem pamięci, gdzie przechowywane były pośrednie wyniki operacji arytmetycznych i logicznych. Tłumacząc to na dzisiejsze miary, moglibyśmy powiedzieć, że ta 5-tonowa maszyna zapewniała dostęp do 72 komórek pamięci, z których każda miała 78 bitów długości (potrzebujemy 78 bitów do reprezentowania dużych liczb 23-cyfrowych); tak

więc oferowała pamięć o rozmiarze 702 bajtów! Programy były wtedy de facto ciągami obliczeń matematycznych działających na tych 72 komórkach pamięci. Były to języki programowania pierwszej generacji (określane skrótowo 1GL) albo języki maszynowe, gdzie programy były przechowywane na taśmie perforowanej, która była wprowadzana do maszyny na żądanie lub obsługiwano je przy pomocy przełączników na przednim panelu. Mógł wykonywać tylko trzy operacje dodawania lub odejmowania na sekundę. Pojedyncze mnożenie zabierało 20 sekund, a obliczenie funkcji $\sin(x)$ zabierało minutę! Tak jak w przypadku Z3, zarządzanie pamięcią w tej maszynie w ogóle nie istniało – można było tylko odczytywać lub zapisywać wartość w jednej z wyżej wspomnianych komórek pamięci.

Dla nas interesujący będzie fakt, że z tego komputera wywodzi się termin *architektura harwardzka* (zobacz rysunek 1-1). Zgodnie z tą architekturą kod programu i przechowywane dane są od siebie fizycznie oddzielone. Dane są przetwarzane przez jakiś rodzaj urządzenia elektronicznego lub elektromechanicznego (takiego jak centralna jednostka przetwarzająca – CPU, czyli procesor). Takie urządzenie często odpowiada też za sterowanie urządzeniami wejścia/wyjścia, takimi jak czytniki kart perforowanych, klawiatury lub urządzenia ekranowe. Choć komputery Z3 albo Mark I wykorzystywały tę architekturę ze względu na jej prostotę, to nie została ona całkiem zapomniana. Jak zobaczymy w rozdziale 2, jest ona wykorzystywana obecnie w niemal każdym komputerze w formie określonej jako *zmodyfikowana architektura harwardzka*. Możemy nawet zobaczyć jej wpływ na programy, które codziennie piszemy.



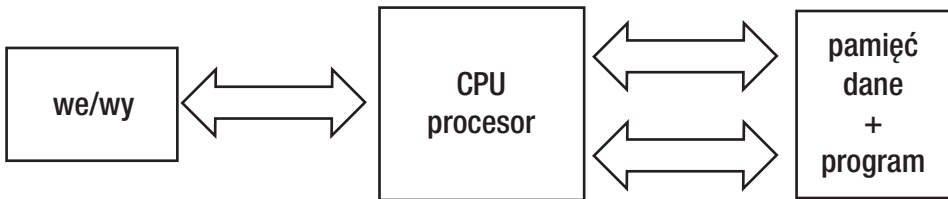
Rysunek 1-1 Schemat architektury harwardzkiej

Bardziej znany komputer ENIAC ukończony w 1946 roku był już urządzeniem elektronicznym opartym na lampach próżniowych. Oferował tysiące razy szybsze wykonywanie operacji matematycznych niż Mark I. Jednakże pod względem pamięci nadal wyglądał mało atrakcyjnie. Oferował jedynie 20 10-cyfrowych (ze znakiem) akumulatorów i nie posiadał pamięci wewnętrznej do przechowywania programów. Ze względu na II wojnę światową priorytetem było jak najszybsze budowanie maszyn obliczeniowych dla celów militarnych, a nie budowanie czegoś bardziej złożonego.

Naukowcy, tacy jak Konrad Zuse, Alan Turing i John von Neumann, zajmowali się jednak pomysłami wykorzystania wewnętrznej pamięci komputera do przechowywania

programu razem z jego danymi. Pozwoliłoby to na znacznie łatwiejsze programowanie (a w szczególności przeprogramowywanie) niż kodowanie poprzez karty perforowane lub mechaniczne przełączniki. John von Neumann napisał w 1945 roku bardzo wpływową pracę zatytułowaną „First Draft of a Report on the EDVAC” (Pierwszy szkic raportu dotyczącego maszyny EDVAC), w której opisuje architekturę zwaną *architekturą von Neumanna*. Warto podkreślić, że nie były to wyłącznie pomysły von Neumanna, gdyż inspirował się on też pracami innych naukowców z tego okresu.

Architektura von Neumanna pokazana na rysunku 1-2 jest uproszczoną architekturą harwardzką, w której jest pojedyncza jednostka pamięciowa do przechowywania zarówno danych, jak i programu. Z pewnością przypomina to nam obecne komputery i nie bez powodu. Z bardzo ogólnego punktu widzenia dokładnie w taki sposób są nadal konstruowane nowoczesne komputery, w których architektura von Neumanna i architektura harwardzka spotykają się w postaci zmodyfikowanej architektury harwardzkiej.



Rysunek 1-2 *Schemat architektury von Neumanna*

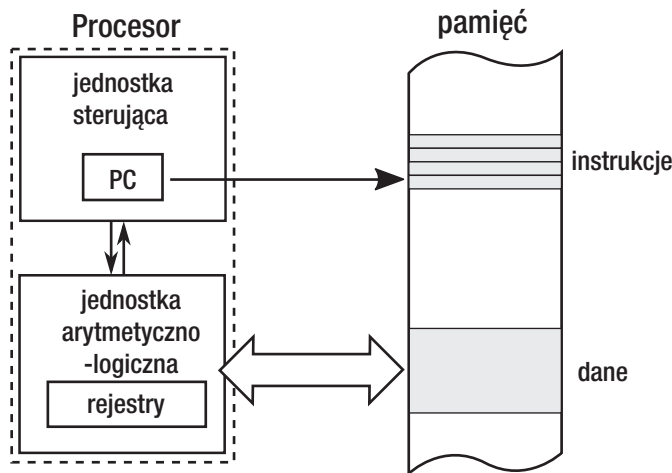
Maszyna Manchester Small-Scale Experimental Machine (SSEM) o przydomku „Baby”, zbudowana w 1948 roku oraz powstały w 1949 roku w Cambridge komputer EDSAC były pierwszymi komputerami na świecie, które przechowywały instrukcje programu oraz dane w tym samym obszarze, a więc wcielały w życie architekturę von Neumanna. Komputer „Baby” był dużo bardziej nowoczesny i innowacyjny, ponieważ był pierwszym komputerem wykorzystującym nowy rodzaj pamięci – lampy Williamsa oparte na lampach kineskopowych (CRT). Lampy Williamsa mogą być uważane za pierwszą pamięć o dostępie swobodnym (RAM). Pamięć komputera SSEM składała się z 32 komórek, z których każda miała 32 bity długości. Możemy więc powiedzieć, że pierwszy komputer z pamięcią RAM miał jej całe 128 bajtów! Przeszliśmy więc drogę od 128 bajtów w 1949 roku do typowych 16 gibibajtów w 2018 roku. Lampy Williamsa stały się standardem na przełomie lat 40. i 50., gdy powstało wiele innych komputerów.

Prowadzi nas to do idealnego momentu historycznego, w którym możemy wyjaśnić wszystkie podstawowe pojęcia architektury komputera. Wszystko zestawiono poniżej i pokazano na rysunku 1-3:

- *Pamięć* – jest odpowiedzialna za przechowywanie danych i samego programu. Sposób implementacji pamięci ewoluował znacząco w czasie, poczynając od wspomnianych wcześniej kart perforowanych, przez pamięci magnetyczne i lampy kineskopowe, aż do obecnie wykorzystywanych tranzystorów. Pamięć można podzielić na dwie główne podkategorie:
 - ◆ *Pamięć o dostępie swobodnym (RAM – Random Access Memory)* – pozwala nam odczytywać dane z takim samym czasem dostępu, niezależnie od tego, do którego regionu pamięci sięgamy. W praktyce (jak zobaczymy w rozdziale 2) nowoczesna pamięć spełnia ten warunek tylko w przybliżeniu ze względów technologicznych.
 - ◆ *Pamięć o dostępie niejednorodnym (non-uniform access memory)* – przeciwieństwo RAM, czas wymagany na dostęp do pamięci zależy od położenia danej komórki na fizycznym nośniku. Dotyczy to oczywiście kart perforowanych, taśm magnetycznych, klasycznych dysków twardych, płyt CD i DVD, itd., gdzie nośnik pamięciowy musi być odpowiednio ustawiony (na przykład obrócony) przed uzyskaniem dostępu do danego fragmentu pamięci.
- *Adres* – reprezentuje określone położenie w całym obszarze pamięci. Jest zwykle wyrażany w formie bajtów, gdyż pojedynczy bajt jest najmniejszym możliwym do zaadresowania elementem na wielu platformach.
- *Jednostka arytmetyczno-logiczna (ALU – arithmetic and logic unit)* – odpowiada za wykonywanie operacji, takich jak dodawanie i odejmowanie. Jest to serce komputera, w którym odbywa się większość pracy. Obecnie komputery zawierają więcej niż jedną jednostkę arytmetyczno-logiczną, co pozwala na przeprowadzanie obliczeń równoległych.
- *Jednostka sterująca* – dekoduje instrukcje programu (*opcodes* – kody operacji) wczytywane z pamięci. W oparciu o wewnętrzny opis instrukcji wie, którą operację arytmetyczną lub logiczną należy wykonać i na których danych.
- *Rejestr* – miejsce w pamięci szybko dostępne dla jednostki arytmetyczno-logicznej i/lub jednostki sterującej (do których możemy odwoływać się zbiorczo jako do *jednostek wykonawczych*) i zwykle w niej zawarte. Wspomniane wcześniej akumulatory są specjalnymi, uproszczonymi typami rejestrów. Rejestry są niezwykle szybkie, jeśli chodzi o czas dostępu i właściwie żadne miejsce przechowywania danych nie jest bliższe jednostkom wykonawczym niż one.
- *Słowo maszynowe* – podstawowa jednostka danych o określonym rozmiarze, wykorzystywana przez dany projekt komputera. Widać jej odzwierciedlenie w wielu obszarach projektowych, takich jak rozmiar większości rejestrów, maksymalny adres albo największy blok danych przekazywany w pojedynczej operacji.

Najczęściej jest wyrażane w liczbie bitów (mówimy wtedy o *rozmiarze* lub *długości słowa maszynowego*). Większość dzisiejszych komputerów jest 32-bitowa lub 64-bitowa, więc w tym przypadku długość słowa maszynowego wynosi odpowiednio 32 bity lub 64 bity, rejestry są 32-bitowe lub 64-bitowe, itd.

Architektura von Neumanna ucieleśniona w maszynach SSEM lub EDSAC prowadzi do określenia *komputery z przechowywanymi programami*, co obecnie jest oczywiste, ale nie było takie w początkach ery komputerów. W takim założeniu kod programu, który ma być wykonywany, jest przechowywany w pamięci, więc może być dostępny, tak jak normalne dane – włączając w to takie przydatne operacje, jak modyfikowanie go i nadpisywanie nowym kodem programu.



Rysunek 1-3 Schemat komputera z przechowywanym programem – pamięć + wskaźnik instrukcji

Jednostka sterująca przechowuje dodatkowy rejestr zwany *wskaźnikiem instrukcji* (IP – *instruction pointer*) albo *licznikiem programu* (PC – *program counter*), który wskazuje na aktualnie wykonywaną instrukcję. Normalne wykonywanie programu polega na zwiększaniu adresu przechowywanego w liczniku programu, aby przechodzić do kolejnych instrukcji. Operacje, takie jak pętle lub skoki, wymagają po prostu zmiany wartości wskaźnika instrukcji na inny adres, określając, gdzie chcemy przenieść wykonywanie programu.

Pierwsze komputery były programowane przy użyciu kodu binarnego, który bezpośrednio określał wykonywane instrukcje. Jednakże wraz ze zwiększającą się złożonością programów to rozwiązanie stawało się coraz bardziej uciążliwe. Nowy język programowania (określany jako druga generacja języków programowania – 2GL) został

zaprojektowany, tak aby opisywać kod w bardziej przystępny sposób przy pomocy tak zwanego *kodu asemblera* (*assembly code*). Jest to tekstowy i bardzo zwarty opis poszczególnych instrukcji wykonywanych przez procesor. Było to znacznie wygodniejsze od bezpośredniego kodowania binarnego. Później zaprojektowano języki jeszcze wyższego poziomu (3GL), takie jak powszechnie znane języki C, C++ lub Pascal.

Co ciekawe, wszystkie te języki muszą być przekształcone z formy tekstowej na binarną, a następnie umieszczone w pamięci komputera. Proces takiej transformacji jest nazywany *kompilacją*, a narzędzie, które ten proces przeprowadza nazywane jest *kompilatorem*. W przypadku kodu asemblera proces ten nazywamy *asemblacją* wykonywaną przez narzędzie zwane *assemblerem*. Końcowym wynikiem jest program w formie kodu binarnego, który można później wykonywać – ma on postać sekwencji kodów operacji oraz ich argumentów (operandów).

Wyposażeni w podstawową wiedzę możemy teraz zacząć swoją podróż po temacie zarządzania pamięcią.

Alokacja statyczna

Większość wczesnych języków programowania pozwalała jedynie na *statyczną alokację pamięci* – ilość i dokładne położenie pamięci musiały być znane podczas kompilacji, jeszcze przed wykonaniem programu. Przy stałych i predefiniowanych rozmiarach zarządzanie pamięcią było trywialne. Wszystkie „starodawne” języki programowania, począwszy od kodu maszynowego czy kodu asemblera, aż po pierwsze wersje języków FORTRAN i ALGOL, miały takie ograniczone możliwości. Mają też one jednak wiele minusów. Statyczne alokacje pamięci mogą łatwo prowadzić do niewydajnego wykorzystania pamięci – jeśli nie wiemy z góry, ile danych będzie przetwarzane, jak możemy ustalić, ile pamięci powinniśmy przydzielić? To sprawia, że programy są ograniczone i mało elastyczne. Ogólnie rzecz biorąc, taki program powinien zostać ponownie skompilowany, aby mógł przetwarzać większe ilości danych.

W najwcześniejszych komputerach wszystkie alokacje były statyczne, ponieważ wykorzystane komórki pamięci (akumulatory, rejestry albo komórki pamięci RAM) były ustalane podczas kodowania programu. Zdefiniowane „zmienne” istniały więc przez cały czas życia programu. Obecnie nadal wykorzystujemy alokację statyczną w takim sensie, gdy tworzymy globalne zmienne statyczne (i tym podobne), przechowywane w specjalnym segmencie danych programu. W późniejszych rozdziałach zobaczymy, gdzie są one przechowywane w przypadku programów .NET.

Maszyna rejestrowa

Jak dotąd widzieliśmy przykłady maszyn, które wykorzystywały rejestry (lub akumulatory w konkretnym przypadku) przy wykonywaniu operacji na jednostkach arytmetyczno-logicznych (ALU). Maszyna opierająca się na takich założeniach jest nazywana *maszyną rejestrową*. Podczas wykonywania programów na takim komputerze w istocie wykonujemy obliczenia na rejestrach. Jeśli chcemy wykonać operację dodawania, dzielenia lub jakąś inną, musimy załadować właściwe dane z pamięci do odpowiednich rejestrów. Następnie wywołujemy konkretne instrukcje, aby przeprowadzić na nich odpowiednie operacje oraz inną instrukcję, aby przepisać wynik z jednego z rejestrów do pamięci.

Założmy, że chcemy napisać program, który oblicza wyrażenie $s = x + (2 * y) + z$ w komputerze z dwoma rejestrami o nazwach A i B. Założmy też, że s, x, y oraz z są adresami w pamięci, pod którymi przechowywane są pewne wartości. Zakładamy też, że mamy jakiś niskopoziomowy kod pseudo-asmblera z instrukcjami typu Load (załaduj), Add (dodaj) i Multiply (pomnóż). Taką teoretyczną maszynę można zaprogramować przy pomocy następującego prostego programu (zobacz listing 1-1).

Listing 1-1 Pseudokod przykładowego programu wykonującego obliczenie

$s = x + (2 * y) + z$ na prostej maszynie rejestrowej z dwoma rejestrami. Komentarze pokazują stan rejestru po wykonaniu każdej instrukcji.

```
Load      A, y      // A = y
Multiply  A, 2      // A = A * 2 = 2 * y
Load      B, x      // B = x
Add       A, B      // A = A + B = x + 2 * y
Load      B, z      // B = z
Add       A, B      // A = A + B = x + 2 * y + z
Store     s, A      // s = A
```

Jeśli ten kod przypomina komuś x86 lub inny znany kod asmblera – nie jest to przypadek. Wynika to z faktu, że prawie każdy nowoczesny komputer jest pewnego rodzaju złożoną maszyną rejestrową. Wszystkie procesory firm Intel i AMD, które wykorzystujemy w swoich komputerach, działają w taki sposób. Przy pisaniu kodu asmblera bazującego na x86/x64 operujemy na rejestrach ogólnego zastosowania, takich jak eax, ebx, ecx, itd. Oczywiście istnieje dużo więcej instrukcji, dodatkowe, wyspecjalizowane rejestry, itd. Ale ogólna zasada jest taka sama.

Uwaga Czy można by sobie wyobrazić maszynę z zestawem instrukcji, które pozwalają nam przeprowadzać operacje bezpośrednio w pamięci, bez konieczności ładowania danych do rejestrów? Kod w naszym języku pseudo asemblera mógłby wyglądać na bardziej zwarty i byłby zbliżony do języków wysokiego poziomu, ponieważ nie byłoby dodatkowych instrukcji ładujących dane z pamięci do rejestrów i zapisujących wartości z rejestrów w pamięci:

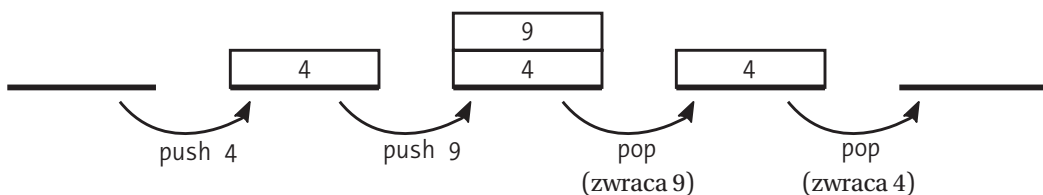
```
Multiply    s, y, 2    // s = 2 * y
Add         s, x       // s = s + x = 2 * y + x
Add         s, z       // s = s + z = 2 * y + x + z
```

Tak, były takie maszyny, np. IBM System/360, ale obecnie nie kojarzę żadnego powszechnie wykorzystywanego komputera tego rodzaju.

Stos

Koncepcyjnie stos jest strukturą danych, którą można po prostu opisać jako listę LIFO – „last in, first out” (ostatnie na wejściu, pierwsze na wyjściu). Pozwala na dwie główne operacje: odłożenie jakichś danych na wierzch stosu („push”) i zdjęcie jakichś danych z wierzchu stosu („pop”), co zilustrowano na rysunku 1-4.

Stos od samego początku stał się nieodłącznie związany z programowaniem komputerów, głównie ze względu na pojęcie podprogramu. Obecnie .NET intensywnie wykorzystuje pojęcia „stosu wywołań” (call stack) i „stosu” (stack), więc przyjrzyjmy się, od czego to wszystko się zaczęło. Pierwotne znaczenie stosu jako struktury danych nadal obowiązuje (na przykład mamy w .NET dostępny typ kolekcji `Stack<T>`), ale zobaczmy, jak to znaczenie ewoluowało w odniesieniu do organizacji pamięci komputera.



Rysunek 1-4 Operacje pop i push na stosie. Jest to tylko rysunek koncepcyjny, niepowiązany z żadnym konkretnym modelem pamięci czy implementacją.

Najwcześniejsze komputery, o których mówiliśmy wcześniej, pozwalały jedynie na sekwencyjne wykonywanie programu, wczytując każdą instrukcję jedną po drugiej z karty perforowanej lub taśmy. Ale pomyśl, aby pisać niektóre części programów

(*podprogramy*), tak aby można było z nich korzystać z różnych punktów całego programu, był bardzo kuszący. Możliwość wywoływania różnych części programu wymagała oczywiście, aby kod można było adresować, gdyż w jakiś sposób musimy wskazywać, jaką inną część programu chcemy wywołać. Pierwotne podejście było stosowane przez słyną Grace Hopper w systemie A-0 – zwanym pierwszym kompilatorem. Zakodowała ona zestaw różnych programów na taśmie, nadając każdemu kolejną liczbę, umożliwiającą komputerowi jego znalezienie. Właściwy „program” składał się z sekwencji liczb (indeksów podprogramów) i odpowiadających parametrów. Choć system ten faktycznie wywoływał podprogramy, był oczywiście w dużym stopniu ograniczony. Program mógł wywoływać podprogramy tylko kolejno po sobie i nie były możliwe zagnieżdżone wywołania.

Zagnieżdżone wywołania wymagają nieco bardziej złożonego podejścia, ponieważ komputery muszą jakoś pamiętać, gdzie mają kontynuować wykonywanie programu (gdzie wrócić) po zakończeniu wykonywania określonego podprogramu. Przechowywanie adresu powrotu w jednym z akumulatorów było pierwszym podejściem wynalezionym przez Davida Wheelera na maszynie EDSAC (metoda ta jest nazywana „*skokiem Wheelera*”). W jego uproszczonym podejściu nie były możliwe wywołania *rekurencyjne*, co oznacza wywoływanie podprogramu z poziomu jego samego.

Pierwsza wzmianka o pojęciu stosu w takim znaczeniu, w jakim znamy je dzisiaj w kontekście architektury komputerów, pojawiła się prawdopodobnie w raporcie Alana Turinga opisującym ACE (Automatic Computing Engine – automatyczny silnik obliczeniowy) z wczesnych lat 40. XX wieku. Opisywał on koncepcję maszyny typu von Neumanna, która była faktycznie komputerem z programami przechowywanymi w pamięci. Poza wieloma innymi szczegółami implementacyjnymi opisywał on dwie instrukcje – BURY (zakop) i UNBURY (odkop) – działające na głównej pamięci i akumulatorach:

- Przy wywoływaniu podprogramu (BURY) adres aktualnie wykonywanej instrukcji zwiększony o 1 (żeby wskazywać na następną instrukcję, do której należy wrócić z podprogramu) był zapisywany w pamięci. Inna, tymczasowo zachowywana wartość, służąca jako wskaźnik stosu, była zwiększana o 1.
- Przy powrocie z podprogramu (UNBURY) podejmowane było działanie odwrotne.

Stanowiło to pierwszą implementację stosu w sensie zorganizowanego w formie LIFO miejsca na adresy powrotów z podprogramów. Jest to rozwiązanie, które nadal jest używane w nowoczesnych komputerach i choć oczywiście znacząco ewoluowało od tego czasu, to podstawy są nadal takie same.

Stos jest bardzo ważnym aspektem zarządzania pamięcią, ponieważ przy programowaniu w .NET wiele naszych danych może być tam umieszczanych. Przyjrzymy się bliżej działaniu

stosu i jego wykorzystaniu w wywołaniach funkcji. Wykorzystamy przykładowy program z listingu 1-2 napisany w pseudokodzie podobnym do języka C, w którym wywołujemy dwie funkcje – najpierw `main` wywołuje funkcję `fun1` (przekazując dwa argumenty `a` i `b`), która ma dwie zmienne lokalne `x` i `y`. Następnie funkcja `fun1` wywołuje w pewnym momencie funkcję `fun2` (przekazując pojedynczy argument `n`), która ma pojedynczą zmienną lokalną `z`.

Listing 1-2 *Pseudokod programu wywołującego funkcję wewnątrz innej funkcji*

```
void main()
{
    ...
    fun1(2, 3);
    ...
}

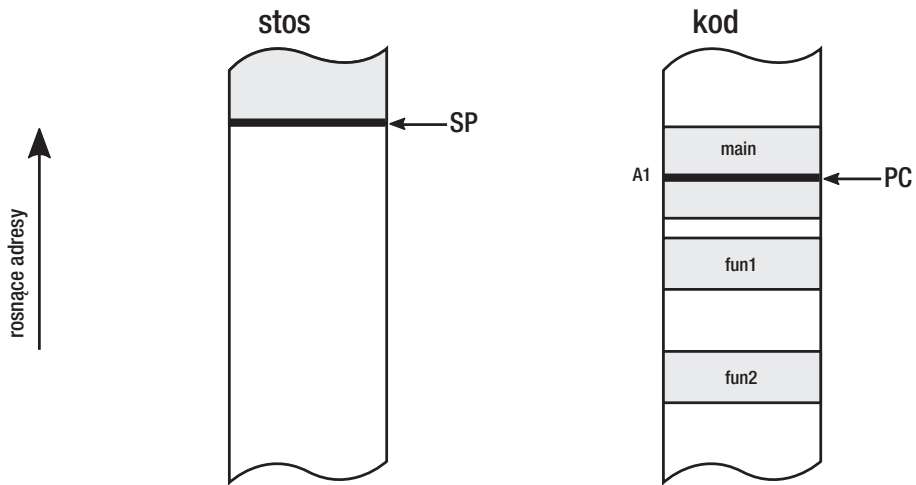
int fun1(int a, int b)
{
    int x, y;
    ...
    fun2(a+b);
}

int fun2(int n)
{
    int z;
    ...
}
```

Najpierw wyobraźmy sobie ciągły obszar pamięci przeznaczony na obsługę stosu, narysowany w taki sposób, że adresy kolejnych komórek pamięci rosną zgodnie ze strzałką w górę (lewa część na rysunku 1-5a) oraz drugi region pamięci, gdzie znajduje się nasz kod (prawa część rysunku 1-5a), zorganizowany w taki sam sposób. Ponieważ kod funkcji nie musi leżeć obok siebie, bloki kodu dla funkcji `main`, `fun1` i `fun2` zostały narysowane oddzielone od siebie. Wykonywanie programu z listingu 1-2 można opisać w następujących krokach:

1. Tuż przed wywołaniem `fun1` wewnątrz funkcji `main` (zobacz rysunek 1-5a). Ponieważ program już działa, jakiś region na stosie został już utworzony (szara część regionu stosu na rysunku 1-5a). Wskaźnik stosu (SP) przechowuje adres wskazujący na aktualną granicę stosu. Licznik programu (PC) wskazuje na jakieś miejsce

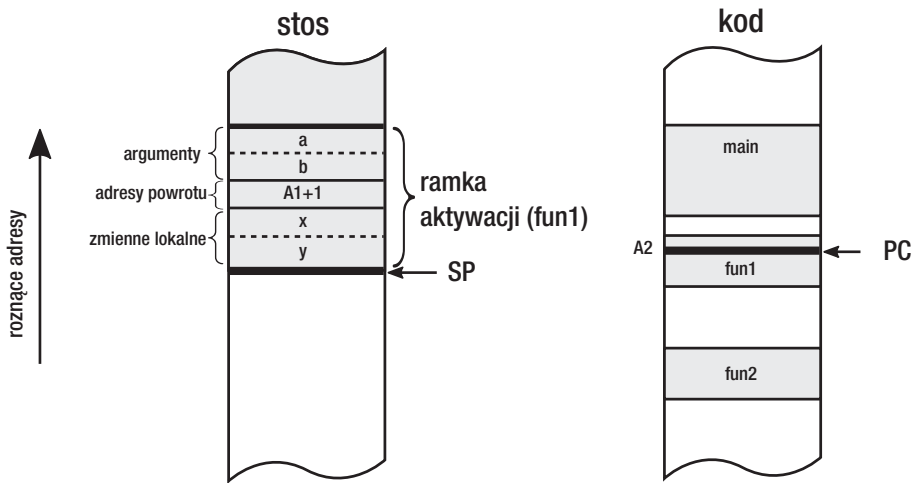
wewnątrz funkcji `main` (oznaczyliśmy to jako adres `A1`) tuż przed instrukcją wywołującą funkcję `fun1`.



Rysunek 1-5a Regiony pamięci dla stosu i dla kodu – w momencie tuż przed wywołaniem funkcji `fun1` z listingu 1-2

2. Po wywołaniu funkcji `fun1` wewnątrz `main` (zobacz rysunek 1-5b). Gdy funkcja jest wywoływana, stos jest rozszerzany przez przesunięcie wskaźnika `SP`, tak aby stos zawierał potrzebne informacje. Ten dodatkowy obszar obejmuje:
 - ◆ Argumenty – wszystkie argumenty funkcji mogą być zapisane na stosie. W naszym przykładzie zostały tu zapisane argumenty `a` oraz `b`.
 - ◆ Adres powrotu – aby mieć możliwość dalszego wykonywania funkcji `main` po wykonaniu wywołania funkcji `fun1`, adres następnej instrukcji tuż po wywołaniu funkcji jest zapisywany na stosie. W naszym przypadku oznaczamy go jako adres `A1+1` (wskazujący na następną instrukcję po instrukcji pod adresem `A1`).
 - ◆ Zmienne lokalne – miejsce na wszystkie zmienne lokalne, które mogą być również zapisane na stosie. W naszym przykładzie zostały tu zapisane zmienne `x` oraz `y`.

Taka struktura umieszczana na stosie, gdy wywoływany jest podprogram, jest nazywana *ramką aktywacji* (*activation frame*). W typowej implementacji wskaźnik stosu jest zmniejszany o odpowiednie przesunięcie, aby wskazywał na miejsce, gdzie może zacząć się nowa ramka aktywacji. Dlatego często mówi się, że stos rośnie w dół.



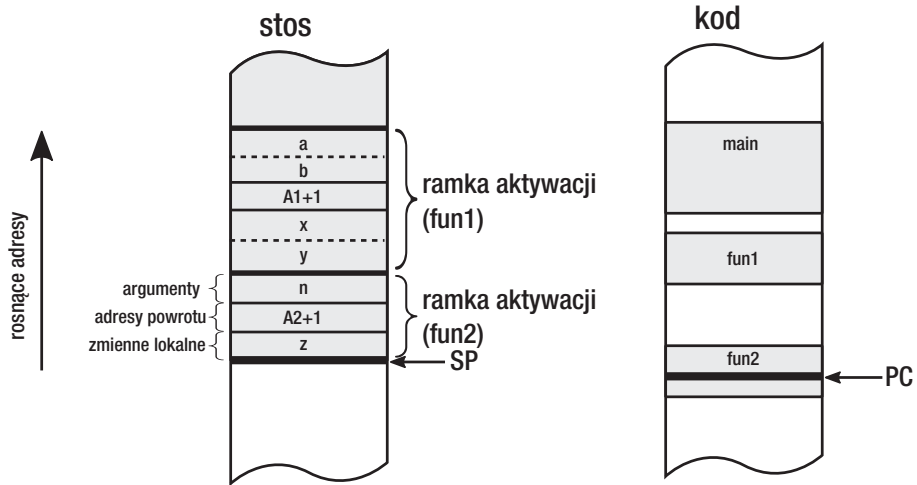
Rysunek 1-5b Regiony pamięci dla stosu i dla kodu – w chwili po wywołaniu funkcji fun1 z listingu 1-2

- Po wywołaniu funkcji fun2 wewnątrz funkcji fun1 (zobacz rysunek 1-5c). Powtarza się ten sam wzorec tworzenia nowej ramki aktywacji. Tym razem zawiera ona region pamięci dla argumentu n, adresu powrotu A2+1 oraz zmiennej lokalnej z.

Ramka aktywacji jest też nazywana bardziej ogólnie *ramką stosu* (*stack frame*), co oznacza dowolne ustrukturyzowane dane zapisywane na stosie w określonym celu.

Jak widać, kolejne zagnieżdżone wywołania podprogramów po prostu powtarzają ten wzorec, dodając pojedynczą ramkę aktywacji dla każdego wywołania. Im bardziej zagnieżdżone wywołania podprogramów, tym więcej ramek aktywacji będzie na stosie. Oczywiście sprawia to, że nieskończone zagnieżdżone wywołania podprogramów nie są możliwe, gdyż wymagałyby pamięci na nieskończoną liczbę ramek aktywacji². To jest właśnie przypadek wyjątku `StackOverflowException`, z którym pewnie każdy programista kiedyś się spotkał. Oznacza on, iż wywołano tak dużo zagnieżdżonych podprogramów, że limit pamięci przeznaczony na stos się wyczerpał.

² Jest jeden ciekawy wyjątek zwany rekurencją ogonową (prawostronną), którego tutaj nie opisujemy dla zachowania zwięzłości.



Rysunek 1-5c Regiony pamięci dla stosu i dla kodu – w chwili po wywołaniu funkcji *fun2* z funkcji *fun1*

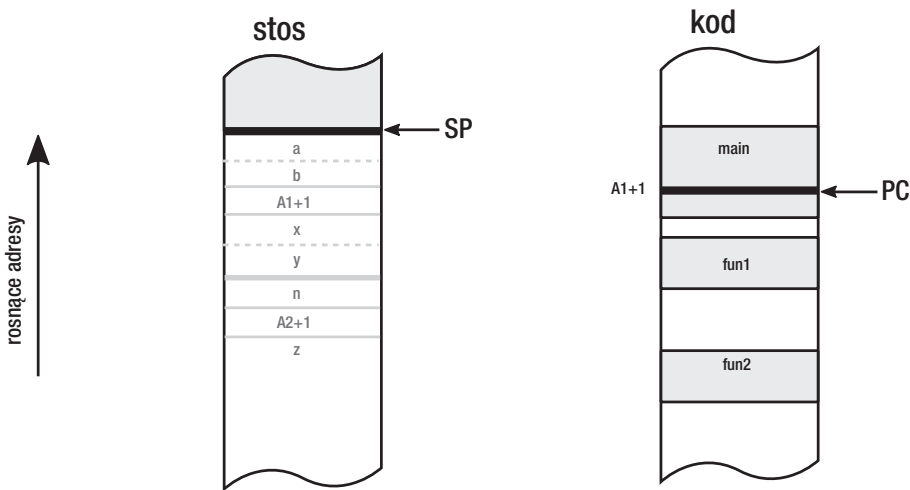
Jak widać, kolejne zagnieżdżone wywołania podprogramów po prostu powtarzają ten wzorzec, dodając pojedynczą ramkę aktywacji dla każdego wywołania. Im bardziej zagnieżdżone wywołania podprogramów, tym więcej ramek aktywacji będzie na stosie. Oczywiście sprawia to, że nieskończone zagnieżdżone wywołania podprogramów nie są możliwe, gdyż wymagałyby pamięci na nieskończoną liczbę ramek aktywacji³. To jest właśnie przypadek wyjątku `StackOverflowException`, z którym pewnie każdy programista kiedyś się spotkał. Oznacza on, iż wywołano tak dużo zagnieżdżonych podprogramów, że limit pamięci przeznaczonej na stos się wyczerpał.

Trzeba mieć na uwadze, że mechanizm przedstawiony tutaj jest tylko przykładowy i bardzo uogólniony. Faktyczne implementacje mogą się różnić pomiędzy różnymi architekturami i systemami operacyjnymi. W późniejszych rozdziałach przyjrzymy się dokładniej, jak ramki aktywacji i stos są wykorzystywane przez .NET.

Gdy wykonywanie podprogramu się kończy, jego ramka aktywacji jest odrzucana przez zwiększenie wskaźnika stosu o rozmiar aktualnej ramki aktywacji, natomiast zapisany adres powrotu jest wykorzystywany do ustawienia licznika programu (PC), żeby wykonywanie programu było kontynuowane w funkcji wywołującej. Innymi słowy, to, co było wewnątrz ramki stosu (zmienne lokalne, parametry), nie jest już potrzebne, więc zwiększenie wskaźnika stosu wystarczy do „zwolnienia” pamięci wykorzystywanej

³ Jest jeden ciekawy wyjątek zwany rekurencją ogonową (prawostronną), którego tutaj nie opisujemy dla zachowania zwięzłości.

przez tę ramkę. Dane te zostaną po prostu nadpisane przy następnym użyciu stosu (zobacz rysunek 1-6).



Rysunek 1-6 Regiony pamięci dla stosu i dla kodu – po powrocie z funkcji `fun1` obie wcześniejsze ramki aktywacji są odrzucane

Jeśli chodzi o implementację, to zarówno wskaźnik SP, jak i licznik PC są zwykle przechowywane w dedykowanych rejestrach. W tym momencie sam rozmiar tego adresu, określone obszary pamięci i rejestry nie są szczególnie istotne.

Stos w nowoczesnych komputerach jest obsługiwany zarówno przez sprzęt (poprzez zapewnianie dedykowanych rejestrów dla wskaźników stosu) oraz przez oprogramowanie (zapewnianą przez system operacyjny abstrakcję wątku i część pamięci wydzieloną jako stos).

Warto zauważyć, że można sobie wyobrazić wiele różnych implementacji stosu z punktu widzenia architektury sprzętowej. Stos może być przechowywany w dedykowanym bloku pamięci wewnątrz CPU albo na dedykowanym układzie scalonym. Może też wykorzystywać ogólną pamięć komputera. Tak jest dokładnie w przypadku najnowszych architektur, gdzie stos jest po prostu regionem o określonym rozmiarze w pamięci procesu. Mogą też istnieć implementacje z architekturą o wielu stosach. W takim przykładowym przypadku stos dla adresów powrotów z podprogramów mógłby być oddzielony od stosu z danymi – parametrami i zmiennymi lokalnymi. Może to być korzystne z powodów wydajnościowych, ponieważ pozwala na jednoczesny dostęp do dwóch oddzielnych stosów. Pozwala to na dodatkowe dostrajanie przetwarzania potokowego przez procesor i innych mechanizmów niskiego poziomu. W każdym razie w obecnych komputerach osobistych stos jest po prostu częścią głównej pamięci.

FORTTRAN może być uważany za pierwszy szeroko używany wysokopoziomowy język programowania ogólnego przeznaczenia. W roku 1954, gdy był definiowany, możliwa była tylko alokacja statyczna. Wszystkie tablice musiały mieć rozmiary definiowane w czasie kompilacji a wszystkie alokacje były oparte na stosie. ALGOL był kolejnym bardzo ważnym językiem, który w większym lub mniejszym stopniu stanowił bezpośrednią inspirację dla mnóstwa innych języków (takich jak C/C++, Pascal, Basic, a poprzez języki Simula i Smalltalk – wszystkie nowoczesne języki zorientowane obiektowo, takie jak Python czy Ruby). ALGOL 60 miał jedynie alokacje na stosie – wraz z tablicami dynamicznymi (o rozmiarach określanych przez zmienne). Alan Perlis, wybitny członek zespołu, który stworzył język ALGOL, powiedział:

Algol 60 nie byłby w stanie odpowiednio działać bez pojęcia stosów. Choć mieliśmy stosy już wcześniej, dopiero w języku Algol 60 stosy zajęły centralne miejsce w projektowaniu procesorów.

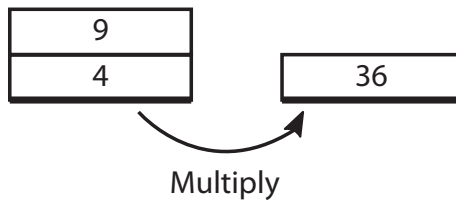
Podczas gdy rodzina języków ALGOL i FORTRAN była wykorzystywana głównie przez naukowców, to pojawił się inny nurt rozwijający języki programowania zorientowane biznesowo, poczynawszy od „A-0”, FLOW-MATIC, poprzez COMTRANS do szerzej znanego języka COBOL (Common Business Language). We wszystkich brakowało bezpośredniego zarządzania pamięcią i operowały głównie na prymitywnych typach danych, takich jak liczby i łańcuchy znaków.

Maszyna stosowa

Zanim przejdziemy do innych pojęć związanych z pamięcią, pozostanmy przez chwilę w kontekście związanym ze stosem i omówmy tak zwane *maszyny stosowe*. W przeciwieństwie do maszyny rejestrowej, w maszynie stosowej wszystkie instrukcje są wykonywane na dedykowanym *stosie wyrażen* (*expression stack*) albo *stosie obliczeń* (*evaluation stack*). Trzeba mieć na uwadze, że ten stos nie musi być tym samym stosem, o którym mówiliśmy wcześniej. Taka maszyna mogłaby więc mieć zarówno dodatkowy „*stos wyrażen*”, jak i stos ogólnego zastosowania. Poza stosem może w ogóle nie być rejestrów. W takiej maszynie instrukcje domyślnie pobierają argumenty z wierzchu stosu wyrażen – tyle, ile ich potrzebują. Wyniki również są zapisywane na wierzchu stosu. W takich przypadkach mamy do czynienia z *czystymi maszynami stosowymi* w przeciwieństwie do implementacji, w których operacje mogą mieć dostęp nie tylko do wartości z wierzchu stosu, ale mogą też sięgać głębiej.

Jak dokładnie wygląda operacja na stosie wyrażen? Na przykład hipotetyczna instrukcja Multiply (pomnóż), wykonywana bez żadnych argumentów, pobierze dwie wartości

z wierzchu stosu wyrażeń, przemnoży je przez siebie i odłoży wynik na wierzchu stosu wyrażeń (zobacz rysunek 1-7).



Rysunek 1-7 Hipotetyczna instrukcja *Multiply* na maszynie stosowej – ściąga dwa elementy ze stosu i odkłada na stos wynik ich mnożenia przez siebie

Wróćmy do przykładowego wyrażenia $s = x + (2 * y) + z$ znanego już z omówienia maszyny rejestrowej i przepiszmy go w stylu maszyny stosowej (zobacz listing 1-3).

Listing 1-3 Pseudokod prostej maszyny stosowej wykonującej obliczenie $s = x + (2 * y) + z$. Komentarze pokazują stan stosu wyrażeń.

```
// pusty stos
Push 2           // [2] - pojedynczy element na stosie - wartość 2
Push y           // [2][y] - dwa elementy na stosie - wartość 2 oraz y
Multiply         // [2*y]
Push x           // [2*y][x]
Add              // [2*y+x]
Push z           // [2*y+x][z]
Add              // [2*y+x+z]
Pop 1            // [] (z efektem ubocznym zapisania wartości pod 1)
```

Ten pomysł prowadzi do bardzo przejrzystego i zrozumiałego kodu. Główne zalety można opisać następująco:

- Nie ma problemu z tym, jak i gdzie zapisywać wartości tymczasowe – czy miałyby to być rejestry, stos, czy pamięć główna. Konceptyjnie jest to łatwiejsze niż próba optymalnego zarządzania wszystkimi tymi miejscami docelowymi. Upraszcza to implementację.
- Kody operacji mogą być krótsze w sensie wymaganej pamięci, gdyż istnieje wiele instrukcji bez operandów lub z pojedynczymi operandami. Pozwala to na wydajne binarne kodowanie instrukcji i daje bardziej zagęszczony kod binarny. Nawet

mimo tego, że liczba instrukcji może być większa niż w podejściu opartym na rejestrach ze względu na większą liczbę operacji load/store, to nadal jest to korzystne.

Było to ważną zaletą we wczesnych czasach komputerów, gdy pamięć była bardzo droga, a jej ilość była ograniczona. Może to być też korzystne obecnie w przypadku kodu pobieranego na smartfony lub w aplikacjach WWW. Gęste kodowanie binarne instrukcji oznacza też lepsze wykorzystanie pamięci podręcznej procesora.

Pomimo swoich zalet, założenia maszyny stosowej były rzadko implementowane w samym sprzęcie. Jednym z ważnych wyjątków były maszyny firmy Burroughs, takie jak B5000, które zawierały sprzętową implementację stosu. Obecnie chyba nie ma szeroko wykorzystywanej maszyny, która mogłaby być określona jako maszyna stosowa. Pewnym wyjątkiem jest jednostka zmiennoprzecinkowa x87 (wewnątrz procesorów kompatybilnych z x86), która była zaprojektowana jako maszyna stosowa, a ze względu na kompatybilność wsteczną nadal jest programowana w ten sposób nawet dzisiaj.

Dlaczego więc w ogóle wspominamy tego rodzaju maszyny? Ponieważ taka architektura jest świetnym sposobem projektowania niezależnych od platformy maszyn wirtualnych albo silników wykonawczych. Wirtualna maszyna Java firmy Sun oraz środowisko uruchomieniowe .NET są świetnymi przykładami maszyn stosowych. Są one wykonywane na dobrze znanych maszynach rejestrowych z architekturą x86 lub ARM, ale nie zmienia to faktu, że działają zgodnie z logiką maszyn stosowych. Zobaczmy to wyraźnie, gdy będziemy opisywać język pośredni (IL) w .NET w rozdziale 4. Dlaczego środowisko uruchomieniowe .NET i JVM (Java Virtual Machine) zostały zaprojektowane w ten sposób? Jak zawsze mamy tu do czynienia z pewną mieszanką powodów inżynierskich i historycznych. Kod maszyny stosowej jest kodem wyższego poziomu i lepiej odpowiada faktycznemu bazowemu sprzętowi. Środowisko uruchomieniowe .NET albo maszyna JVM mogłyby zostać napisane jako maszyny rejestrowe, ale wtedy, ile rejestrów byłoby potrzebnych? Ponieważ są tylko wirtualne, najlepszą odpowiedzią byłoby – nieskończona liczba rejestrów. Potrzebny byłby sposób ich obsługi i wielokrotnego wykorzystywania. Jak wyglądałaby optymalna, abstrakcyjna maszyna oparta na rejestrach?

Jeśli odłożymy takie problemy, pozwalając czemuś innemu (środowisku Java lub .NET w tym przypadku) dokonywać optymalizacji dla konkretnych platform, przetłumaczy to mechanizmy oparte na rejestrach lub oparte na stosie na określoną architekturę opartą na rejestrach. Maszyny oparte na stosie są jednak koncepcyjnie prostsze. Wirtualna maszyna stosowa (taka, która nie jest wykonywana przez prawdziwą, sprzętową maszynę stosową) może zapewniać dobrą niezależność od platformy, dając przy tym wysoce wydajny kod. Łącząc to ze wspomnianym lepszym zagęszczeniem kodu, otrzymujemy dobry wybór dla platformy, która ma być uruchamiana na szerokim zakresie urządzeń. Prawdopodobnie taki był powód, dla którego firma Sun zdecydowała się wybrać tę ścieżkę przy implementowaniu środowiska Java dla różnych, niewielkich

urządzeń. Firma Microsoft przy projektowaniu .NET również poszła tą drogą. Koncepcja maszyny stosowej jest po prostu elegancka, prosta i zwyczajnie działa. To sprawia, że implementowanie maszyny wirtualnej staje się przyjemniejszym zadaniem inżynierskim!

Z drugiej strony projekty maszyn wirtualnych opartych na rejestrach są bardziej zbliżone do projektów rzeczywistego sprzętu, na którym będą uruchamiane. Jest to bardzo pomocne w zakresie możliwych optymalizacji. Zwolennicy tego podejścia mówią, że pozwala ono osiągnąć dużo lepszą wydajność zwłaszcza w interpretowanych środowiskach uruchomieniowych. Interpreter ma znacznie mniej czasu na wykonanie zaawansowanych optymalizacji, im bardziej więc interpretowany kod będzie podobny do kodu maszynowego, tym lepiej. Dodatkowo operowanie na najczęściej używanym zestawie rejestrów zapewnia świetne wykorzystanie pamięci podręcznej⁴.

Jak zawsze przy podejmowaniu decyzji trzeba poczynić pewne kompromisy. Dyskusja pomiędzy zwolennikami obu podejść jest długa i nierozstrzygnięta. W każdym razie faktem jest, że obecnie silnik wykonawczy .NET jest zaimplementowany jako maszyna stosowa, choć nie jest to całkiem czysta maszyna stosowa – zauważymy to w rozdziale 4. Zobaczmy też, jak stos obliczeń jest mapowany na bazowy sprzęt składający się z rejestrów i pamięci.

Uwaga Czy wszystkie maszyny wirtualne i silniki wykonawcze są maszynami stosowymi? Absolutnie nie! Jednym z wyjątków jest Dalvik, maszyna wirtualna w systemie Android firmy Google do wersji 4.4, która była implementacją JVM opartą na rejestrach. Był to interpreter pośredniego „kodu bajtowego Dalvik”. Ale później kompilacja JIT (Just in Time) została wprowadzona w następcy maszyny Dalvik – środowisku Android Runtime (ART). Do innych przykładów należą BEAM – maszyna wirtualna dla Erlang/Elixir, Chakra – silnik wykonawczy JavaScript w przeglądarce IE9, Parrot (maszyna wirtualna dla Perl 6) oraz Lua VM (maszyna wirtualna dla Lua). Nie można więc powiedzieć, że tego rodzaju maszyna nie jest popularna.

⁴ Uwaga: W rozdziale 2 przyjrzymy się znaczeniu wzorców dostępu do pamięci w kontekście użycia pamięci podręcznej.

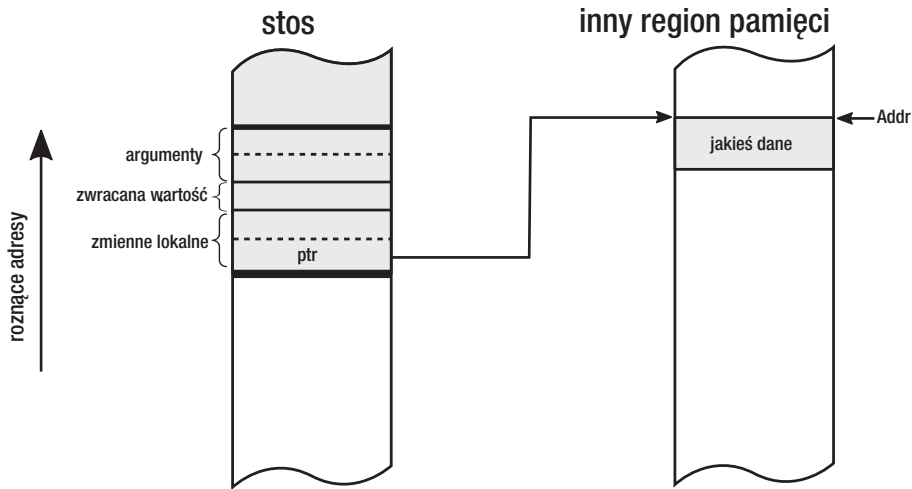
Wskaźnik

Jak dotąd wprowadziliśmy tylko dwa pojęcia pamięciowe: alokację statyczną i alokację na stosie (jako część ramki stosu). Pojęcie *wskaźnika* (*pointer*) jest bardzo ogólne i pojawia się od samego początku ery komputerowej – jak w przypadku wcześniej przedstawionego pojęcia wskaźnika instrukcji (licznika programu) albo wskaźnika stosu. Określone rejestry przeznaczone do adresowania pamięci, takie jak *rejestry indeksowe* (*index registers*) również mogą być traktowane jako wskaźniki⁵.

Język PL/I został zaproponowany przez IBM ok. roku 1965 i w zamierzeniu miał mieć ogólne zastosowanie zarówno w świecie nauki, jak i biznesu. Choć ten cel nie został całkiem osiągnięty, to stanowi on ważny element historii informatyki, ponieważ był to pierwszy język, który wprowadził pojęcie wskaźników i alokacji pamięci. Harold Lawson, który brał udział w rozwoju języka PL/I, otrzymał w 2000 roku nagrodę od IEEE „za wymyślenie zmiennej wskaźnikowej i wprowadzenie tego pojęcia do PL/I, zapewniając w ten sposób po raz pierwszy możliwość elastycznej obsługi list łączonych w wysokopoziomowym języku ogólnego przeznaczenia”. Dokładnie taka była potrzeba, która stała za wynalazkiem wskaźnika – zapewnienie możliwości przetwarzania list i działań na innych mniej lub bardziej złożonych strukturach danych. Pojęcie wskaźnika zostało następnie zastosowane podczas opracowywania języka C, który ewoluował z języka B (i jego poprzedników BCPL oraz CPL). Dopiero wersja języka FORTRAN 90 (następca FORTRAN 77) zdefiniowana w 1991 wprowadzała dynamiczną alokację pamięci (poprzez procedury *allocate/deallocate*), atrybut *POINTER*, przypisywanie wskaźników oraz instrukcję *NULLIFY*.

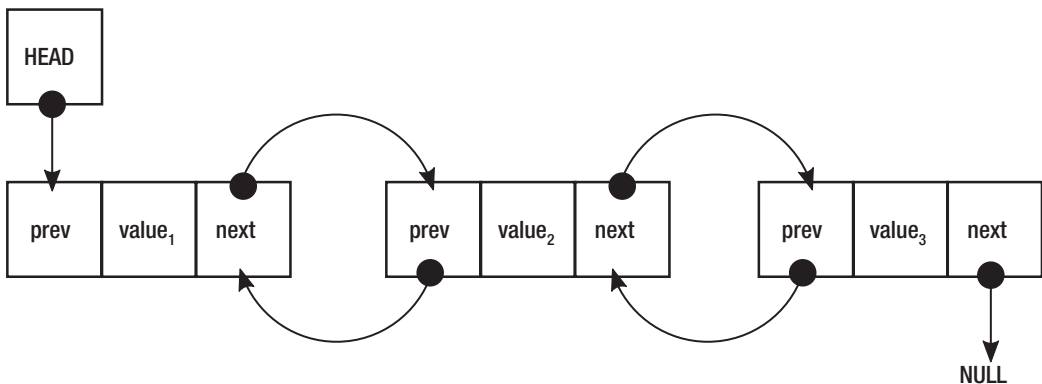
Wskaźnik jest zmienną, w której przechowujemy adres jakiegoś położenia w pamięci. Pozwala nam to odwoływać się do innych miejsc w pamięci poprzez ich adres. Rozmiar wskaźnika jest związany z długością słowa, o której wspominaliśmy wcześniej, co wynika z architektury danego komputera. Obecnie więc zwykle mamy do czynienia z 32-bitowymi lub 64-bitowymi wskaźnikami. Ponieważ sam wskaźnik zajmuje niewielki obszar w pamięci, możemy go umieszczać na stosie (na przykład jako zmienną lokalną lub argument funkcji) albo w rejestrze procesora. Rysunek 1-8 pokazuje typową sytuację, gdzie jedna ze zmiennych lokalnych (zapisana wewnątrz ramki aktywacji funkcji) jest wskaźnikiem do innego regionu w pamięci pod adresem *Addr*.

5 W kontekście adresowania pamięci ważnym ulepszeniem był rejestr indeksowy wprowadzony w maszynę Manchester Mark 1, następcy komputera „Baby”. Rejestr indeksowy pozwalał nam na bezpośrednie odwoływanie się do pamięci, dodając jego wartość do innego rejestru. Dzięki temu mniej instrukcji było wymaganych do wykonywania operacji na ciągłych regionach pamięci, takich jak tablice.



Rysunek 1-8 *Zmienna lokalna w funkcji, będąca wskaźnikiem ptr prowadzącym do pamięci pod adresem Addr*

Prosta koncepcja wskaźników pozwala nam budować złożone struktury danych, takie jak listy łączone lub drzewa, ponieważ struktury danych w pamięci mogą się odwoływać do innych elementów, tworząc bardziej złożone struktury (zobacz rysunek 1-9).



Rysunek 1-9 *Wskaźniki używane do zbudowania struktury listy łączonej dwukierunkowej, gdzie każdy element wskazuje na swój element poprzedni oraz następny*

Co więcej, wskaźniki mogą zapewniać tak zwaną *arytmetykę wskaźnikową*. Można do nich dodawać lub odejmować od nich jakieś wartości, aby odwoływać się do miejsca w pamięci położonego odpowiednio względem danego wskaźnika. Na przykład

operator inkrementacji może zwiększać wartość wskaźnika o wartość rozmiaru wskazywanego obiektu, a nie o jeden bajt (jak można by się spodziewać).

W wysokopoziomowych językach, takich jak Java lub C# wskaźniki często nie są dostępne albo muszą być jawnie włączane w trybie tzw. kodu niebezpiecznego. Przyczyny takiego stanu rzeczy okażą się jaśniejsze, gdy będziemy omawiać ręczne zarządzanie pamięcią przy użyciu wskaźników w dalszej części tego rozdziału.

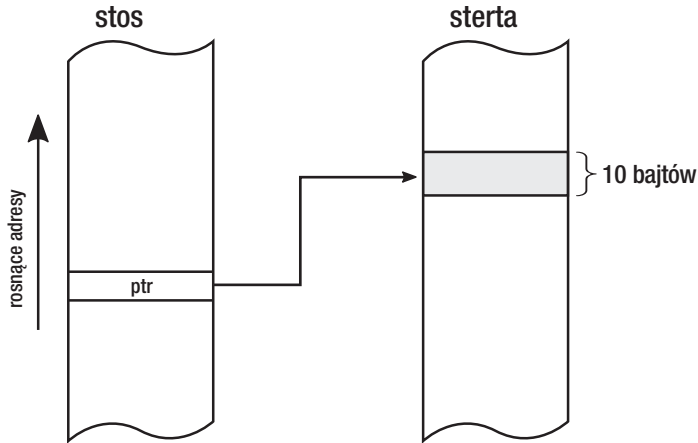
Sterta

Dochodzimy wreszcie do najważniejszego pojęcia w kontekście zarządzania pamięcią w .NET. *Sterta* (*heap*), czasami też nazywana *magazynem swobodnym* (*Free Store*), jest obszarem pamięci używanym do dynamicznego alokowania obiektów. Magazyn swobodny jest lepszą nazwą, ponieważ nie sugeruje żadnej konkretnej struktury wewnętrznej, ale raczej określa cel tego obszaru. Właściwie można by zapytać, jakie jest powiązanie pomiędzy stertą jako strukturą danych a stertą w odniesieniu do dynamicznego zarządzania pamięcią. Prawda jest taka, że nie ma takiego powiązania. Podczas gdy stos jest dobrze zorganizowany (jest oparty na pojęciu struktury danych LIFO), to sterta bardziej przypomina „czarną skrzynkę”, którą możemy prosić o zapewnienie pamięci, nie zajmując się tym, skąd będzie ona pochodzić. Dlatego więc wcześniej wspomniane określenie „magazyn swobodny” albo „pula” byłoby chyba bardziej odpowiednią nazwą. Nazwa sterta została chyba użyta na początku w tradycyjnym znaczeniu angielskiego wyrazu „heap”, określającym „bezładny zbiór rzeczy” w przeciwieństwie do dobrze uporządkowanego stosu. Historycznie rzecz biorąc, alokacja na stercie została wprowadzona w języku ALGOL 68, ale ten standard nie przyjął się zbyt szeroko. Prawdopodobnie stąd właśnie pochodzi określenie „sterta”. Faktem jest, że prawdziwe historyczne pochodzenie tej nazwy jest obecnie dość trudne do ustalenia.

Sterta jest mechanizmem pamięciowym mogącym zapewniać ciągły blok pamięci o określonym rozmiarze. Ta operacja jest nazywana *dynamiczną alokacją pamięci*, ponieważ zarówno rozmiar, jak i faktyczne położenie danego obszaru pamięci nie muszą być znane w czasie kompilacji. Ponieważ położenie w pamięci nie jest znane w czasie kompilacji, do pamięci alokowanej dynamicznie musimy odwoływać się przez wskaźnik. Z tego względu pojęcia wskaźnika i sterty są ze sobą naturalnie powiązane.

Adres zwracany przez jakąś funkcję typu „przydziel mi X bajtów pamięci” powinien być oczywiście zapamiętany w jakimś wskaźniku, aby możliwe było dalsze odwoływanie się do utworzonego bloku pamięci. Wskaźnik ten może być przechowywany na stole (zobacz rysunek 1-10), na samej stercie lub gdziekolwiek indziej.

```
PTR ptr = allocate(10);
```

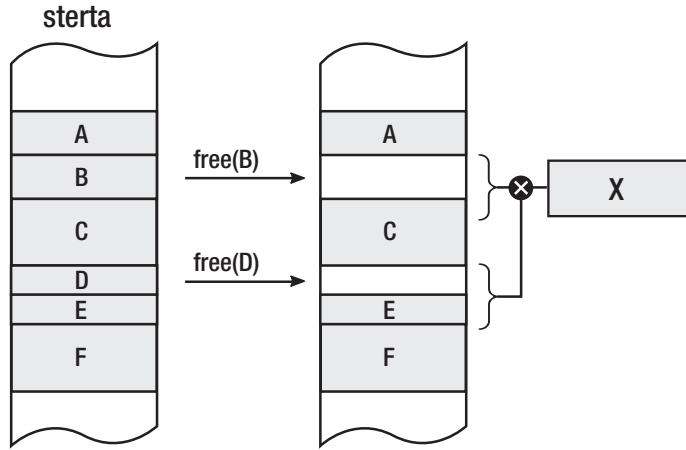


Rysunek 1-10 Stos ze wskaźnikiem *ptr* oraz 10-bajtowy blok pamięci na stercie

Operacja odwrotna do operacji alokacji jest nazywana *dealokacją* i polega na zwróceniu danego bloku pamięci do puli pamięci dostępnej do wykorzystania w przyszłości. Dokładny sposób alokowania przez stertę obszaru o danym rozmiarze stanowi szczegół implementacyjny. Istnieje wiele mechanizmów alokujących i niektóre z nich poznamy wkrótce.

Poprzez alokację i dealokację wielu bloków pamięci możemy doprowadzić do sytuacji, gdy nie będzie wystarczająco dużego ciągłego bloku na dany obiekt, choć sumaryczna dostępna na stercie ilość wolnego miejsca byłaby wystarczająca. Taka sytuacja jest określana *fragmentacją* sterty i może prowadzić do znacznej niewydolności wykorzystania pamięci. Rysunek 1-11 ilustruje taki problem, gdy nie ma wystarczająco dużego wolnego ciągłego miejsca na obiekt X. Istnieje wiele różnych strategii wykorzystywanych przez mechanizmy alokujące przy zarządzaniu pamięcią w możliwie optymalny sposób w celu unikania fragmentacji.

Warto też zauważyć, że kolejnym szczegółem implementacyjnym może być to, czy mamy tylko pojedynczą stertę, czy wiele wystąpień stert w pojedynczym procesie (zobaczmy to przy dokładniejszym omawianiu .NET).



Rysunek 1-11 Fragmentacja – po usunięciu obiektów B i D nie ma wystarczającego miejsca na nowy obiekt X, choć w sumie jest dość wolnego miejsca

Podsumujmy krótko różnice pomiędzy stosem i stertą w tabeli 1-1.

Tabela 1-1 Porównanie funkcjonalności stosu i sterty

Właściwość	Stos	Sterta
Czas życia	Zakres zmiennych lokalnych (odkładane na stos na wejściu do funkcji, zdejmowane ze stosu na wyjściu z funkcji)	Jawnie określony (przez alokację i ewentualne zwolnienie pamięci)
Zakres	Lokalny (dla wątku ¹)	Globalny (dostępne dla każdego, kto ma wskaźnik)
Dostęp	Zmienna lokalna, argumenty funkcji	Wskaźnik
Czas dostępu	Szybki (prawdopodobnie region w pamięci podręcznej procesora)	Wolniejszy (może być nawet tymczasowo zapisywany na twardego dysku)
Alokacja	Przesuwanie wskaźnika stosu	Różne możliwe strategie
Czas alokacji	Bardzo szybki	Wolniejszy (zależy od strategii alokacji)
Zwalnianie	Przesuwanie wskaźnika stosu	Różne możliwe strategie
Użycie	Parametry podprogramów, zmienne lokalne, ramki aktywacji, niewielkie dane o rozmiarze znanym w czasie kompilacji	Wszystko

Właściwość	Stos	Sterta
Pojemność	Ograniczona (zwykle kilka MB na wątek)	Nieograniczona (w ramach dostępnej pojemności pamięci i dysku)
Zmienny rozmiar	Nie	Tak ²
Fragmentacja	Nie	Prawdopodobna
Główne zagrożenia	Przepelnienie stosu	Wyciek pamięci (zapomnienie o zwolnieniu zaalokowanej pamięci), fragmentacja

1 Nie jest to całkiem prawdziwe, gdyż możemy przekazywać wskaźnik do zmiennej na stosie do innych wątków. Jednakże z pewnością nie jest to standardowe użycie.

2 Ze względu na dynamiczną naturę sterty, istnieją funkcje pozwalające nam zmieniać rozmiar (realokować) danego bloku pamięci.

Oprócz tych różnic stos i sterta najczęściej są lokalizowane po przeciwnych stronach przestrzeni adresowej procesu. Wróćmy do szczegółowego ułożenia stosu i sterty w przestrzeni adresowej procesu podczas rozważań nad niskopoziomowym zarządzaniem pamięcią w rozdziale 2. W każdym razie należy pamiętać, że jest to tylko szczegół implementacyjny. Dzięki zapewnieniu abstrakcji dla typów wartościowych i referencyjnych (które zostaną wprowadzone w rozdziale 4) nie powinniśmy przejmować się, gdzie są one tworzone.

Przejdźmy teraz do omówienia ręcznego i automatycznego zarządzania pamięcią. Ellis i Stroustrup pisali w podręczniku *The Annotated C++ Reference Manual*:

Programiści C uważają, że zarządzanie pamięcią jest zbyt ważnym zadaniem, żeby zostawiać je komputerowi. Programiści Lisp uważają, że zarządzanie pamięcią jest zbyt ważnym zadaniem, żeby zostawiać je użytkownikowi.

Ręczne zarządzanie pamięcią

Jak na razie mieliśmy do czynienia z „ręcznym zarządzaniem pamięcią”. Oznacza to w szczególności, że programista odpowiada za jawne alokowanie pamięci, a gdy nie jest już dłużej potrzebna, powinien ją dealokować. To naprawdę ręczna praca. To jak z ręczną skrzynią biegów w większości samochodów europejskich. Jestem z Europy i po prostu jesteśmy przyzwyczajeni do ręcznego zmieniania biegów. Musimy pomyśleć, czy już jest dobry moment na zmianę biegu, czy powinniśmy poczekać parę sekund, aż obroty silnika będą odpowiednio wysokie. Ma to jedną dużą zaletę – mamy pełną kontrolę nad samochodem. Sami odpowiadamy za to, czy silnik jest wykorzystywany optymalnie, czy nie. A ponieważ ludzie wciąż lepiej adaptują się do zmieniających się warunków, dobrzy kierowcy mogą

to robić lepiej niż automatyczna skrzynia biegów. Oczywiście jest też jedna duża wada. Zamiast myśleć o naszym głównym celu – dotarciu z punktu A do punktu B, musimy dodatkowo myśleć o zmienianiu biegów – setki, tysiące razy podczas długiej podróży. Jest to zarówno czasochłonne, jak i męczące. Znam osoby, które twierdzą, że jest to zajmujące, a oddanie kontroli automatycznej skrzyni biegów jest nudne. Mogę się nawet z nimi zgodzić. Podoba mi się jednak, jak ta przenośnia pasuje do zarządzania pamięcią.

Gdy mówimy o jawnym alokowaniu i dealokowaniu pamięci, odpowiada to dokładnie korzystaniu z ręcznej skrzyni biegów. Zamiast myśleć o naszym głównym celu, którym jest pewnie jakieś biznesowe zadanie do wykonania przez nasz kod, musimy też myśleć o tym, jak zarządzać pamięcią swojego programu. Odciąga to nas od naszego głównego celu i zajmuje naszą cenną uwagę. Zamiast myśleć o algorytmach, logice biznesowej i domenach, musimy również myśleć o tym, kiedy i jak dużo pamięci będziemy potrzebować. Jak długo będzie nam ona potrzebna? Kto będzie odpowiadać za jej zwalnianie? Czy to brzmi jak logika biznesowa? Oczywiście, że nie. Pytanie, czy to dobrze, czy nie, to już inna kwestia.

Dobrze znany język C został zaprojektowany przez Dennisa Rithiego gdzieś we wczesnych latach 70. XX wieku i stał się jednym z najszerzej używanych języków programowania na świecie. Historia wyewoluowania języka C z języka ALGOL poprzez pośrednie języki CPL, BCPL, i B jest sama w sobie dość ciekawa, ale w naszym kontekście ważne jest, że język C wraz z językiem Pascal (wywodzącym się bezpośrednio z języka ALGOL) były w tym okresie dwoma najpopularniejszymi językami z jawnym zarządzaniem pamięcią. Jeśli chodzi o C, to mogę bez wątplenia powiedzieć, że kompilator tego języka został napisany dla niemal każdej architektury sprzętowej kiedykolwiek stworzonej. Nie zdziwiłbym się, gdyby statki kosmiczne obcych miały na swoich pokładach własny kompilator C (pewnie wraz z implementacją stosu TCP/IP). Wpływ tego języka na inne języki programowania jest niewyobrażalnie ogromny. Zatrzymajmy się na chwilę i przyjrzyjmy się mu dokładniej w kontekście zarządzania pamięcią. Pozwoli nam to wymienić kilka charakterystycznych cech ręcznego zarządzania pamięcią.

Popatrzmy na prosty kod przykładowy napisany w C i przedstawiony na listingu 1-4.

Listing 1-4 *Przykładowy program w języku C demonstrujący ręczne zarządzanie pamięcią*

```
#include <stdio.h>

void printReport(int* data)
{
    printf("Report: %d\n", *data);
}
```

```

int main(void) {
    int *ptr;
    ptr = (int*)malloc(sizeof(int));
    if (ptr == 0)
    {
        printf("ERROR: Out of memory\n");
        return 1;
    }
    *ptr = 25;
    printReport(ptr);
    free(ptr);
    ptr = NULL;
    return 0;
}

```

Jest to oczywiście nieco przesadzony przykład, ale dzięki niemu możemy jaśniej zilustrować problem. Możemy zauważyć, że ten krótki kod ma w istocie tylko jeden prosty cel biznesowy: wypisać „raport”. Dla uproszczenia raport ten składa się w tym przypadku z pojedynczej liczby całkowitej, ale możemy sobie wyobrazić, że jest to bardziej skomplikowana struktura zawierająca też wskaźniki do innych struktur danych. Ten prosty cel biznesowy jest obudowany dużą ilością „rytualnego kodu” zajmującego się samą tylko pamięcią. Jest to esencja ręcznego zarządzania pamięcią.

Podsumowując powyższy fragment kodu, oprócz napisania logiki biznesowej, programista musi:

- Zaalokować odpowiednią ilość pamięci na potrzebne dane, korzystając z funkcji `malloc`.
- Rzutować zwrócony ogólny wskaźnik (`void*`) na właściwy typ wskaźnika (`int*`), aby określić, że wskazuje on na wartość liczbową (typ `int` w przypadku języka C).
- Zapamiętać wskaźnik do zaalokowanego regionu pamięci w lokalnej zmiennej wskaźnikowej `ptr`.
- Sprawdzić, czy udało się zaalokować potrzebną ilość pamięci (w przypadku niepowodzenia zwrócony zostanie adres 0).
- Uzyskać dostęp do pamięci wskazywanej przez adres zapisany we wskaźniku, aby zapisać pewne dane (liczbową wartość 25).
- Przekazać wskaźnik do innej funkcji `printReport`, która sama też odczytuje wartość z adresu wskazywanego przez wskaźnik.

- Zwolnić zaalokowaną pamięć, gdy nie jest już dłużej potrzebna, korzystając z funkcji `free`.
- Dla pewności powinniśmy zmienić wskaźnik na wartość specjalną `NULL` (co jest sposobem zaznaczenia, że ten wskaźnik na nic już nie wskazuje – wartość ta odpowiada wartości `06`).

Jak widać, musimy mieć wiele rzeczy na uwadze, gdy ręcznie zarządzamy pamięcią. Co więcej, w każdym z powyższych kroków możemy się pomylić albo o czymś zapomnieć, co może prowadzić do różnych poważnych problemów. Przechodząc przez każdy z tych kroków, zastanówmy się, co złego może się stać:

- Powinniśmy dokładnie wiedzieć, ile pamięci potrzebujemy. W naszym przykładzie wystarczy po prostu użycie `sizeof(int)`, ale co w przypadku, gdy mielibyśmy do czynienia z bardziej złożonymi, zagnieżdżonymi strukturami danych? Łatwo sobie można wyobrazić sytuację, w której alokujemy zbyt mało pamięci ze względu na jakiś drobny błąd w ręcznych obliczeniach wymaganego rozmiaru. Później, gdy będziemy chcieli zapisać coś do takiego regionu pamięci (lub z niego odczytać), otrzymamy prawdopodobnie *błąd segmentacji* (*Segmentation Fault*) – próbując uzyskać dostęp do pamięci, która nie została przez nas zaalokowana lub została zaalokowana do innego celu. Z drugiej strony możemy przez podobny błąd zaalokować zbyt dużo pamięci, co będzie skutkowało mało efektywnym wykorzystaniem pamięci.
- Rzutowanie zawsze może być podatne na błędy i może wprowadzać trudne do zdiagnozowania błędy, jeśli przez przypadek wprowadzimy niezgodność typów. Próbowalibyśmy interpretować wskaźnik jakiegoś typu, tak jakby był zupełnie innego typu, co łatwo prowadzi do niebezpieczeństwa naruszenia dostępu.
- Zapamiętywanie adresu jest prostą sprawą. Co jednak, jeśli zapomnimy to zrobić? Będziemy mieli zaalokowany obszar pamięci i brak możliwości jego zwolnienia – zapomnieliśmy właśnie, jaki jest jego adres! Jest to bezpośrednia droga do problemu wycieku pamięci, gdyż niezwolniona pamięć może się szybko rozrosnąć. Co więcej wskaźnik może być przechowywany w czymś bardziej skomplikowanym niż zmienna lokalna. Co będzie, jeśli zgubimy wskaźnik do złożonego grafu obiektów, ponieważ zwolniliśmy pamięć dla jakiejś struktury, która go zawierała?
- Pojedyncze sprawdzenie, czy byliśmy w stanie zaalokować żadaną ilość pamięci, nie jest zbyt uciążliwe. Ale powtarzanie tego setki razy w każdej funkcji z osobna na pewno takie będzie. Prawdopodobnie zdecydujemy się na pominięcie tego sprawdzania, ale może to doprowadzić nas do niezdefiniowanego zachowania

6 Szczegóły implementacyjne wartości `NULL` w przypadku `.NET` będą wyjaśnione w rozdziale 10.

w wielu miejscach naszej aplikacji, gdy będziemy próbować uzyskać dostęp do pamięci, której w ogóle nie udało się z powodzeniem zaalokować.

- Odczytywanie danych, do których prowadzą wskaźniki, jest zawsze niebezpieczne. Nigdy nie wiadomo, co będzie pod adresem, do którego prowadzi wskaźnik. Czy nadal mamy tam prawidłowy obiekt, czy może został on już zwolniony? Czy dany wskaźnik jest w ogóle prawidłowy? Czy wskazuje na właściwą przestrzeń adresową w pamięci użytkownika? Pełna kontrola nad wskaźnikami w językach, takich jak C prowadzi do tego rodzaju obaw. Ręczna kontrola nad wskaźnikami prowadzi do poważnych naruszeń bezpieczeństwa – sam programista musi zadbać o to, aby nie udostępniać danych poza obszarami, w których powinny być dostępne zgodnie z aktualnym modelem typów.
- Przekazywanie wskaźnika pomiędzy funkcjami i wątkami jedynie mnoży te obawy w środowisku wielowątkowym.
- Musimy pamiętać o zwalnianiu zaalokowanej pamięci. Jeśli pominiemy ten krok, uzyskamy wyciek pamięci. W przykładzie tak prostym, jak powyżej, oczywiście trudno będzie zapomnieć o wywołaniu funkcji `free`. Jest to jednak dużo bardziej problematyczne w bardziej złożonych bazach kodu, gdzie odpowiedzialność za poszczególne struktury danych może nie być tak oczywista i gdzie wskaźniki do takich struktur mogą być przekazywane w różne miejsca. Istnieje też jeszcze jedno ryzyko – nikt nas nie może powstrzymać przed zwolnieniem pamięci, która już została zwolniona. Jest to kolejna okazja do niezdefiniowanego zachowania i prawdopodobnie może spowodować błąd segmentacji.
- Wreszcie na koniec powinniśmy oznaczać nasz wskaźnik jako `NULL` (albo `0`) w celu określenia, że już nie wskazuje na prawidłowy obiekt. W przeciwnym razie będziemy mieć do czynienia z tak zwanym *wiszącym wskaźnikiem* (*dangling pointer*), który wcześniej lub później doprowadzi do błędu segmentacji lub innego niezdefiniowanego zachowania, ponieważ ktoś może próbować odczytać wskazywane przez niego dane, sądząc, że nadal prowadzi do prawidłowych danych.

Jak widać, z perspektywy programisty jawne alokowanie i dealokowanie pamięci może się stać naprawdę nieporęczne. Jest to bardzo potężna funkcjonalność, która z pewnością ma swoje zastosowania. Tam, gdzie znaczenie ma jak największa wydajność, a programista musi być w 100% pewien, co się dzieje wewnątrz programu – podejście to może być przydatne. Jednak „duża władza oznacza dużą odpowiedzialność”, więc jest to broń obojętna. W miarę rozwoju inżynierii oprogramowania również języki stawały się coraz bardziej zaawansowane, pomagając programiście uciec przed wszystkimi tymi troskami.

Idąc dalej, bezpośredni następca języka C – język C++ nie zmienił wiele w tym zakresie. Jednakże warto poświęcić kilka chwil językowi C++, ponieważ jest tak popularny

i wprowadza inne szeroko używane pojęcia. Jak wszyscy wiemy, jest to język z ręcznym zarządzaniem pamięcią. Tłumacząc poprzedni przykład na C++, uzyskamy kod pokazany na listingu 1-5.

Listing 1-5 *Przykładowy program w języku C++ pokazujący ręczne zarządzanie pamięcią*

```
#include <iostream>

void printReport(int* data)
{
    std::cout << "Report: " << *data << "\n";
}

int main()
{
    try
    {
        int* ptr;
        ptr = new int();
        *ptr = 25;
        printReport(ptr);
        delete ptr;
        ptr = 0;
        return 0;
    }
    catch (std::bad_alloc& ba)
    {
        std::cout << "ERROR: Out of memory\n";
        return 1;
    }
}
```

W kontekście naszych poprzednich rozważań możemy dostrzec kilka istotnych usprawnień:

- Operator `new` zajmuje się alokacją odpowiedniej ilości pamięci, wiedząc, ile jej będzie potrzebować, dzięki wsparciu kompilatora (który podpowiada rozmiar danego obiektu).

- Nie musimy rzutować otrzymanego wskaźnika na odpowiedni typ. To usuwa pewne problemy z bezpieczeństwem typów, które rozważaliśmy wcześniej.
- Obsługa błędów również się poprawiła, ponieważ nie jesteśmy zobligowani do ręcznego sprawdzania powodzenia alokacji, gdyż w przypadku problemu zostanie wyrzucony wyjątek.

Nadal jednak widać dużo dodatkowego kodu w tym przykładzie. Pojawił się również nowy problem. Co będzie, jeśli funkcja `printReport()` wyrzuci wyjątek? Bez odpowiedniej obsługi błędów możemy łatwo pominąć operator `delete` i doprowadzić do wycieku pamięci. Naprawienie naszego kodu przykładowego jest łatwe, ale może nie być to takie oczywiste w bardziej złożonych aplikacjach, gdzie odpowiedzialność za dane (kto i na jakim poziomie powinien usuwać takie wskaźniki) może nie być tak trywialna.

Wszystkie problemy, które widzieliśmy w tym rozdziale, są dodatkowo potęgowane w środowiskach wielowątkowych, gdy wskaźniki mogą być wspólnie wykorzystywane przez wiele jednostek wykonawczych. Trzeba brać pod uwagę odpowiednią synchronizację, aby nie dopuścić do wymieszania nieprawidłowych danych. Na przykład, co będzie, jeśli jeden wątek sprawdza, czy dany wskaźnik jest poprawny (nie `NULL`), natomiast inny zaraz potem zwolni pamięć przez niego wskazywaną? Takie sytuacje mogą prowadzić do nieregularnych i bardzo trudnych do diagnozowania problemów. W świecie jawnego zarządzania pamięcią, to programista odpowiada za zapewnianie odpowiedniego mechanizmu synchronizacji w celu uniknięcia takich sytuacji.

Przykład w języku C++ zaprezentowany na listingu 1-5 celowo nie jest zgodny z obecnymi wzorcami wykorzystania pamięci w tym języku. Powinien wykorzystywać jakąś technikę RAII (Resource Acquisition Is Initialization – inicjowanie przy pozyskaniu zasobu) – gdzie zasób (taki jak pamięć) jest reprezentowany przez zmienną lokalną typu implementującego jakiegoś rodzaju logikę związaną z własnością pamięci. Taki przykład zostanie przedstawiony później na listingu 1-10. Choć, jak się przekonamy, takie wzorce pomagają w rozwiązywaniu pewnych problemów, to nie zmieniają wiele w naszej ogólnej dyskusji na temat ręcznego i automatycznego zarządzania pamięcią.

Automatyczne zarządzanie pamięcią

Aby przezwyciężyć problemy z ręcznym zarządzaniem pamięcią i zapewnić programiście bardziej przyjazny sposób jej obsługi, pojawiły się różne podejścia do automatycznego zarządzania pamięcią. Dobrze jest wiedzieć, że nawet jeden z najstarszych wysokopoziomowych języków programowania – LISP (zaproponowany ok. 1958 roku – tylko kilka

lat po języku FORTRAN) miał sporo do zaoferowania na tym polu. W języku głównie funkcjonalnym i opierającym się w znacznym stopniu na przetwarzaniu list – ręczne zarządzanie pamięcią byłoby bardzo niewygodne. Paradygmat programowania funkcjonalnego traktuje programy jako obliczanie wyników połączonych funkcji i stanowczo unika modyfikowania danych (mutacji) i związanych z tym efektów ubocznych. Alokowanie i dealokowanie pamięci jest operacją mocno modyfikującą i ma oczywiste efekty uboczne. Obsługa pamięci w taki sposób w kodzie funkcjonalnym zaburzyłaby ten kod elementami imperatywnymi, podczas gdy LISP został zaprojektowany jako język wysoce deklaratywny. Jak powiedział twórca języka LISP: „konieczność jawnego czyszczenia list wyglądałaby niezwykle brzydko”. Trzeba było więc opracować coś bardziej skomplikowanego. Pierwsze wersje języka LISP miały wbudowaną funkcję `eralist` (`erase list` – wyczyść listę), ale została ona usunięta po wprowadzeniu automatycznego zarządzania pamięcią.

LISP był bardzo innowacyjnym językiem i jego projekt pomógł w wynalezieniu wielu ważnych pomysłów informatycznych, a jednym z nich było automatyczne zarządzanie pamięcią. John McCarthy, wynalazca języka LISP, jest też uważany za jednego z „ojców założycieli” sztucznej inteligencji oraz autora pierwszych algorytmów odświeżania pamięci. Wiele pomysłów wymyślonych na potrzeby tego języka nadal spełnia swoje zadanie i jest wykorzystywana w dzisiejszych językach programowania. Można z pewnością stwierdzić, że automatyczne zarządzanie pamięcią narodziło się w języku LISP. Pierwsza praca napisana przez McCarthy’ego w roku 1958 wprowadzała algorytm `Mark and Sweep` (oznacz i zamieć), który zbadamy dokładnie w późniejszych rozdziałach, ponieważ jest wciąż używany w środowisku .NET i wielu innych miejscach.

Język LISP dzięki swojej wyrazistości i zwięzłości może przedstawić nasz program przykładowy w prostej formie pokazanej na listingu 1-6.

Listing 1-6 *Przykładowy program w języku LISP pokazujący automatyczne zarządzanie pamięcią*

```
(defun printReport(data)
  (write-line (format nil "Report: ~a" data))
)

(prog
  ((ptr 25))
  (printReport ptr)
)
```

Dzięki automatycznemu zarządzaniu pamięcią cała otoczka kodu zniknęła i możemy wyraźnie zobaczyć właściwy cel biznesowy programu – wypisanie „raportu”.

John McCarthy przytoczył ciekawą anegdotę w swojej pracy dotyczącej projektowania języka LISP: „Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I” (funkcje rekurencyjne wyrażeń symbolicznych i ich obliczanie przez maszynę, część I). Opisał tam krótko ten mechanizm, ale nazwał go po prostu „odzyskiwaniem”. Później dodał przypis do tego fragmentu:

Nazywaliśmy już ten proces „odśmiecaniem pamięci”, ale wycofałem się z użycia go w tej pracy – w przeciwnym razie korektorki z Research Laboratory of Electronics by jej nie przepuściły.

Nie tylko sama nazwa, ale też cały pomysł na ten mechanizm był opisany i gotowy do implementacji. Obecnie określenia mechanizm automatycznego zarządzania pamięcią i *odśmiecanie pamięci* (*garbage collection*) są używane wymiennie. Możemy to zdefiniować jako mechanizm, który zdejmuje z programisty odpowiedzialność za ręczne zarządzanie pamięcią, tak że raz utworzone obiekty są automatycznie usuwane (a pamięć przez nie wykorzystywana jest odzyskiwana), gdy nie są już dłużej potrzebne.

Jednym z wniosków, jakie chciałbym przekazać w tej książce, jest fakt, że nawet gdy zarządzanie pamięcią jest w pełni automatyczne, może to powodować problemy. Na potwierdzenie warto zacytować interesujący fakt związany z pierwszą implementacją odśmiecania pamięci w języku LISP. Jak wspomina McCarthy w książce *History of Programming Languages I* (*Historia języków programowania I*), podczas pierwszej publicznej demonstracji języka LISP na jednym z sympozjów na uczelni MIT urządzenie Flexowriter (elektryczna maszyna do pisania z tego okresu) zaczęło ze względu na drobne niedopatrzenie drukować wiele stron z komunikatem błędu zaczynającym się od zdania:

ODŚMIECANIE ZOSTAŁO WYWOŁANE. PONIŻEJ PODANO KILKA INTERESUJĄCYCH DANYCH

Z tego względu prezentację trzeba było odwołać, a publiczność miała ubaw. Tylko John wiedział, że spowodowane to było błędnym użyciem mechanizmu odśmiecania pamięci. Choć był to raczej błąd ludzki niż błąd algorytmu, możemy powiedzieć, że mechanizm odśmiecania pamięci miał kłopoty od samego początku!

Alokator, mutator i kolektor

Mutatory i inne pojęcia, z którymi zapoznamy się w tym rozdziale, są ważnymi terminami w akademickich badaniach nad automatycznym zarządzaniem pamięcią. Dzięki jasnym definicjom możemy je później jednoznacznie odróżniać w pracach akademickich i opracowaniach technicznych. Można na przykład powiedzieć o „koszcie mutatora” dla

określonych algorytmów. Przy rozważaniu różnych projektów odśmiecania pamięci często będzie pojawiać się dyskusja dotycząca wpływu kolektora na mutator i vice versa. Przyjrzyjmy się bliżej tym terminom.

Mutator

Wśród kilku podstawowych pojęć związanych z zarządzaniem pamięcią jednym z najbardziej podstawowych i jednocześnie dość ważnych jest abstrakcyjne pojęcie zwane *mutatorem*. W swojej najprostszej wersji możemy zdefiniować mutator jako jednostkę odpowiedzialną za wykonywanie kodu aplikacji. Nazwa ta wywodzi się z faktu, że mutator wywołuje mutację (zmienia) stanu pamięci – obiekty są alokowane lub modyfikowane, a referencje pomiędzy nimi się zmieniają. Innymi słowy mutator jest maszyną kierującą wszystkimi zmianami w aplikacji w odniesieniu do pamięci. Wyrażenie to zostało ukute (wśród wielu innych w tej samej pracy) przez Edsgera Dijkstrę w 1978 roku w pracy „On-the Fly Garbage Collection: An Exercise in Cooperation” (Odśmiecanie pamięci w locie: ćwiczenia ze współdziałania), gdzie możemy znaleźć szczegółowe opracowanie na ten temat. Interesującym faktem jest, że propozycja Dijkstry z tej dość starej pracy jest nadal wykorzystywana z dobrym skutkiem, na przykład w języku Go w 2015 roku.

Podoba mi się abstrakcyjne pojęcie mutatora, ponieważ zapewnia elegancką i czystą klasyfikację elementów wewnątrz określonej platformy lub środowiska uruchomieniowego. Możemy zdefiniować mutator jako wszystko to, co ma możliwość modyfikowania pamięci albo poprzez modyfikowanie istniejących obiektów, albo przez tworzenie nowych. Choć nie jest to całkiem ściśle, możemy dodatkowo rozszerzyć tę definicję na wszystko to, co może odczytywać pamięć (gdyż odczyt jest kluczową operacją przy wykonywaniu programu). Prowadzi to nas do ważnej obserwacji. Do pełnej możliwości działania mutator musi zapewniać uruchomionej aplikacji trzy rodzaje operacji:

- `New(ilość)` – alokowanie podanej ilości pamięci, która następnie będzie wykorzystywana przez nowo utworzony obiekt. Warto zauważyć, że na tym poziomie abstrakcji nie bierzemy pod uwagę informacji o typie obiektu, która może być lub nie być dostępna z poziomu środowiska uruchomieniowego. Zapewniamy jedynie wymagany rozmiar pamięci do zaalokowania.
- `Write(adres, wartość)` – zapisanie określonej wartości pod podanym adresem. Tutaj również abstrahujemy od tego, czy bierzemy pod uwagę pole obiektu (w programowaniu zorientowanym obiektowo), zmienną globalną czy jakiś inny rodzaj organizacji danych.
- `Read(adres)` – odczytanie wartości z podanego adresu.

W najprostszym świecie, gdzie nie istnieją żadne algorytmy odświeczania pamięci, te trzy operacje mają trywialną implementację (zapisaną w pseudokodzie zbliżonym do języka C na listingu 1-7).

Listing 1-7 *Trzy główne metody implementacji mutatora bez automatycznego zarządzania pamięcią*

```
Mutator.New(amount)
{
    return Allocator.Allocate(amount);
}

Mutator.Write(address, value)
{
    *address = value;
}

Mutator.Read(address) : value
{
    return *address;
}
```

Jednak w świecie automatycznego odświeczania pamięci te trzy operacje mają miejsce, gdy mutator kooperuje z mechanizmem odświeczania pamięci (*kolektorem*) i mechanizmem alokacji (*alokatorem*). Jak ta kooperacja wygląda i jak bardzo narusza ona prostotę powyższych implementacji, stanowi jeden z najważniejszych problemów projektowych. Jednym z najbardziej typowych usprawnień, które spotkamy w tej książce, jest dodanie tak zwanej *bariery* – będzie to albo *bariera odczytu* (*read barrier*), albo *bariera zapisu* (*write barrier*). Bariery jest sposobem na dołączenie dodatkowej operacji przed lub po właściwych operacjach. Bariery pozwalają nam na bezpośrednią lub pośrednią synchronizację z mechanizmem odświeczania pamięci w celu uzyskiwania informacji o wykonywaniu programu i zużyciu pamięci. Trzy metody z listingu 1-7 są punktami wstrzykiwania, do których mechanizm odświeczania pamięci może chcieć się podłączyć. Wrócimy do kilku najczęstszych możliwych odmian w kolejnych rozdziałach, gdy będziemy opisywać różne algorytmy odświeczania pamięci.

W codziennej rzeczywistości programisty najczęstszą implementacją abstrakcji mutatora jest dobrze znany *wątek* (*thread*). Idealnie pasuje to do definicji – jest to pojedynczy element, który uruchamia kod i mutuje (zmienia) obiekty oraz grafy referencji pomiędzy obiektami. Jest to dla nas całkiem intuicyjne, ponieważ znaczna większość

najbardziej popularnych środowisk uruchomieniowych wykorzystuje tę implementację. Wśród wielu innych funkcjonalności wątki komunikują się poprzez dodatkową warstwę z systemem operacyjnym, aby realizować operacje New, Write i Read.

Mutatory nie muszą być implementowane jako wątki w sensie wątków systemu operacyjnego. Popularnym przykładem może być ekosystem Erlang z jego procesami – są one zarządzane jak niezwykle lekkie podprogramy istniejące w samym środowisku uruchomieniowym. Mogą być one traktowane jako tak zwane „zielone wątki”, ale w terminologii maszyny wirtualnej Erlang lepiej nazywać je „zielonymi procesami”, gdyż oddzielenie wymuszane przez środowisko uruchomieniowe jest silniejsze niż pomiędzy elementami przypominającymi wątki. Oznacza to, że są to jednostki zarządzane na poziomie środowiska uruchomieniowego, a nie na poziomie systemu operacyjnego. Inna typowa implementacja mutatora mogłaby się opierać na tak zwanych *włóknach* (*fibers*), lekkich jednostkach wykonawczych implementowanych zarówno w systemie Linux, jak i Windows.

Alokator

Mutator musi być w stanie wykorzystywać operację New, którą omówiliśmy we wcześniejszym punkcie. Jeśli chodzi o wewnętrzną implementację tych metod, wcześniej lub później musimy wspomnieć o innym bardzo ważnym pojęciu – *alokatorze*. Alokator jest jednostką odpowiedzialną za zarządzanie dynamiczną alokacją i dealokacją pamięci. Jak wspominaliśmy wcześniej, w starszych językach, takich jak ALGOL lub FORTRAN nie było alokatora, gdyż w ogóle nie było dynamicznej alokacji pamięci.

Alokator musi zapewniać dwie główne operacje:

- `Allocator.Allocate(ilość)` – alokuje określoną ilość pamięci. Może to być oczywiście poszerzone o metody będące w stanie alokować pamięć dla określonego typu obiektu, jeśli informacja o typie jest dostępna dla alokatora. Jak widzieliśmy, jest to wewnętrznie wykorzystywane przez operację `Mutator.New`.
- `Allocator.Deallocate(adres)` – zwalnia pamięć pod podanym adresem, aby była dostępna dla przyszłych alokacji. Trzeba zwrócić uwagę, że w przypadku automatycznego zarządzania pamięcią ta metoda jest wewnętrzna i nie jest dostępna dla mutatora (dlatego żaden kod użytkownika nie może jej wywoływać jawnie).

Pomysł ten może wydawać się całkiem prosty, żeby nie powiedzieć trywialny. Jak jednak zobaczymy, nie jest on taki łatwy, jak można by się spodziewać. Istnieje wiele różnych aspektów projektu alokatora. Jak zawsze wszystko opiera się na kompromisach zwłaszcza pomiędzy wydajnością, złożonością implementacji (co prowadzi bezpośrednio do łatwości utrzymania) i innymi elementami. Zapoznamy się dokładniej z dwoma najpopularniejszymi rodzajami alokatorów: *sekwencyjnym* (*sequential allocator*) i opartym na *liście*

wolnych obszarów (*free-list allocator*). Ponieważ jest to jednak szczegół implementacyjny, lepiej będzie zapoznać się z nimi w konkretnym kontekście .NET w rozdziale 4.

Kolektor

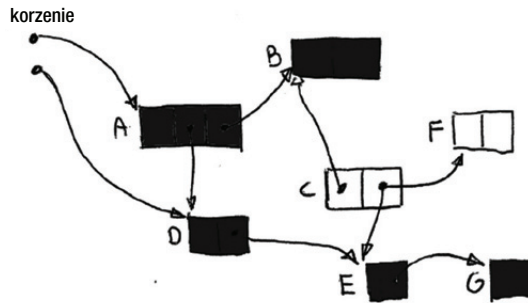
Zdefiniowaliśmy mutator jako jednostkę, która odpowiada za wykonywanie kodu aplikacji, podobnie możemy zdefiniować *kolektor* (*collector*) jako jednostkę, która wykonuje kod odśmiecania pamięci (automatycznego odzyskiwania pamięci). Innymi słowy możemy traktować kolektor jako fragment oprogramowania (kod) albo wątek go wykonujący, albo jedno i drugie. W zależności od kontekstu.

Skąd kolektor wie, które obiekty nie są już potrzebne i można je dealokować? Jest to niemożliwy problem do rozwiązania, ponieważ powinien *de facto* zgadywać przyszłość – czy określony obiekt będzie jeszcze używany? Zależy to od kodu, który będzie wykonywany, a to może z kolei zależeć od czynników, takich jak działania użytkownika, dane zewnętrzne, itd. Idealny kolektor powinien wiedzieć, czy obiekt *żyje* – żywe obiekty to takie, które będą jeszcze potrzebne. Z kolei – *nieżywe obiekty* (albo *śmieci*) nie będą wykorzystywane i mogą zostać zniszczone. Dlatego też kolektor jest nazywany powszechnie mechanizmem odśmiecania pamięci albo kolektorem (zbieraczem) śmieci (*Garbage Collector* lub w skrócie GC).

Istnieje ciekawa konsekwencja współpracy pomiędzy mutatorem, alokatorem i kolektorem. Przypomnijmy ponownie, że ponieważ nie ma udostępnionej publicznie metody `Allocator.Deallocate`, to Mutator nie ma możliwości jawnego zwalniania uzyskanej pamięci. Mutator może tylko prosić o zaalokowanie kolejnych obszarów pamięci, tak jakby dostępne było ich niewyczerpane źródło. To w istocie oznacza, że mechanizm odśmiecania pamięci jest w istocie symulacją komputera z nieskończoną ilością pamięci. To, jak działa ta symulacja i jak jest skuteczna, stanowi szczegół implementacyjny.

Można by wymyślić specjalny mechanizm odśmiecania pamięci, który w ogóle nie zwalnia zaalokowanej pamięci. Nazywa się go *zerowym mechanizmem odśmiecania pamięci* (*Zero Garbage Collector*). Działałby poprawnie jedynie na komputerach z nieskończoną ilością pamięci, które niestety jeszcze nie istnieją. Jednak takie mechanizmy odśmiecania pamięci mają swoje praktyczne zastosowanie. Mogą być na przykład używane w przypadku bardzo krótko żyjących programów, gdzie nieograniczony wzrost pamięci jest dopuszczalny. Być może staną się coraz bardziej popularne w świecie bezserwerowych, krótko działających, pojedynczych funkcji. Przykładowy szkic takiego mechanizmu odśmiecania pamięci dla .NET został przedstawiony w rozdziale 15.

Ponieważ znajomość czasu życia obiektu jest niemożliwa⁷, kolektor opiera się na innej właściwości obiektu – czy dany obiekt jest *osiągalny* dla dowolnego mutatora. *Osiągalność* (*reachability*) obiektu oznacza, że istnieje sekwencja referencji (zaczynająca się z dowolnego obszaru pamięci dostępnej dla mutatora) pomiędzy obiektami, która ostatecznie prowadzi do danego obiektu (zobacz rysunek 1-12). Osiągalność oczywiście nie oznacza żywotności obiektu, ale jest najlepszym przybliżeniem, jakie mamy. Jeśli obiekt nie jest osiągalny z żadnego mutatora, nie może być już używany, więc jest nieżywy i można bezpiecznie odzyskać po nim pamięć. Stwierdzenie przeciwne oczywiście nie jest prawdziwe. Osiągalny obiekt może pozostać zawsze osiągalny (dostępny poprzez referencje jakiegoś złożonego grafu), ale z powodu warunków wykonywania może nigdy nie być wykorzystywany i przez to być „nieżywy”. W istocie większość wycieków w przypadku zarządzanej pamięci pojawia się pomiędzy żywotnością a osiągalnością elementów.



Rysunek 1-12 Osiągalność – obiekty C i F nie są osiągalne, ponieważ nie ma prowadzącej do nich ścieżki od korzeni (lokalizacji mutatora)

Punkty początkowe mutatora w sensie osiągalności są nazywane *korzeniami* (*roots*). Czym są dokładnie, to zależy od konkretnej implementacji mutatora. W większości przypadków, gdy mutator jest po prostu wątkiem (reprezentowanym przez naturalny wątek oparty na systemie operacyjnym), korzenie mogą być:

- Zmiennymi lokalnymi i argumentami podprogramów – umieszczanymi na stosie lub zapisywanymi w rejestrach.
- Statycznie alokowanymi obiektami (np. zmiennymi globalnymi) – umieszczanymi na sterwie.
- Innymi wewnętrznymi strukturami danych przechowywanymi w samym kolektorze.

⁷ W rozdziale 4 omówimy analizę ucieczkową – metodę ustalania prawdziwej żywotności wskaźników przynajmniej w pewnych specjalnych przypadkach.

Zdobywszy wiedzę na temat tych trzech głównych elementów składowych – mutatora, alokatora i kolektora – możemy teraz przejść do zaznajomienia się z wieloma różnymi podejściami do automatycznego zarządzania pamięcią. Choć kuszące jest przedstawienie wyczerpującej listy ze szczegółowym opisem każdego z nich, przekracza to jednak możliwości tej książki. Zamiast tego zapoznamy się z kilkoma głównymi, najpopularniejszymi podejściami, jakie możemy spotkać w dzisiejszych językach.

Zliczanie referencji

Jedna z dwóch najpopularniejszych metod automatycznego zarządzania pamięcią jest nazywana *zliczaniem referencji* (*reference counting*). Pomysł jest bardzo prosty. Opiera się on na zliczaniu liczby referencji do obiektu. Każdy obiekt ma swój własny *licznik referencji* (*reference counter*). Gdy obiekt jest przypisywany do zmiennej lub pola – liczba referencji do niego jest zwiększana. Jednocześnie zmniejszany jest licznik referencji dla obiektu, na który ta zmienna wcześniej wskazywała.

Żywotność obiektów w podejściu zliczania referencji jest śledzona przez liczbę obiektów odwołujących się do danego obiektu. Jeśli licznik spadnie do zera, to nikt nie odwołuje się do danego obiektu i może on zostać zdealokowany. Co jednak, jeśli licznik nie spadnie do zera? Nic to nie mówi o żywotności obiektu – określa to tylko, że ktoś utrzymuje referencję do obiektu, ale nie przesądza, czy zostanie on wykorzystany. Zliczanie referencji jest więc kolejnym, mniej jednoznacznym sposobem zgadywania żywotności obiektu.

Wracając do naszego trywialnego przykładu mutatora z listingu 1-7, w przypadku zliczania referencji można by go opisać, tak jak pokazano na listingu 1-8.

Listing 1-8 Pseudokod opisujący prosty algorytm zliczania referencji

```
Mutator.New(amount)
{
    obj = Allocator.Allocate(amount);
    obj.counter = 0;
    return obj;
}

Mutator.Write(address, value)
{
    if (address != NULL)
        ReferenceCountingCollector.DecreaseCounter(address);
    *address = value;
```

```

    if (value != NULL)
        value.counter++;
}

```

```

ReferenceCountingCollector.DecreaseCounter(address)
{
    *address.counter--;
    if (*address.counter == 0)
        Allocator.Deallocate(address)
}

```

Zachowanie zliczania referencji jest zilustrowane przez prosty program na rysunku 1-13 i listingu 1-9. Trzy proste wiersze kodu zostały przepisane w formie metod mutatora, aby pokazać, jak zmieniają się referencje.

Listing 1-9 Przykładowy pseudokod ilustrujący zliczanie referencji

```

o1 = new SomeObject();
o2 = new SomeObject();
o1 = o2;

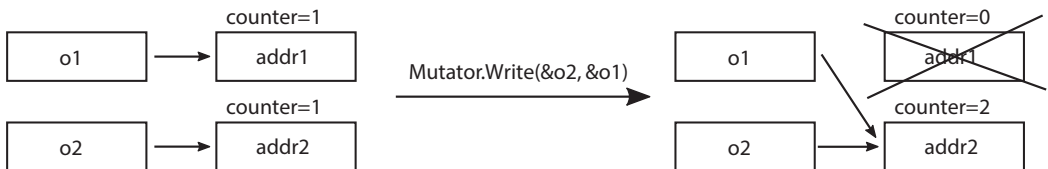
```

// staje się:

```

addr1 = Mutator.New(SizeOf(SomeObject))    // addr1.counter = 0
Mutator.Write(&o1, addr1)                   // addr1.counter = 1
addr2 = Mutator.New(SizeOf(SomeObject))    // addr2.counter = 0
Mutator.Write(&o2, addr2)                   // addr2.counter = 1
Mutator.Write(&o1, &o2)                     //
addr1.counter = 0; addr2.counter = 2

```



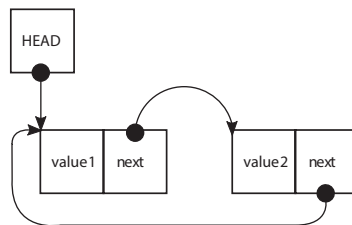
Rysunek 1-13 Ilustracja zliczania referencji z listingu 1-8

Jak widać na listingu 1-9, dodatkowe elementy zostały dodane do operacji `Mutator.Write`. Musi ona sprawdzić i zmodyfikować dane licznika oraz zająć się dealokacją, jeżeli licznik spadnie do zera. Staje się to jeszcze bardziej skomplikowane w środowisku wielowątkowym (gdzie wiele mutatorów działa równocześnie). W takim przypadku operacje te powinny być bezpieczne dla wątków, więc synchronizacja dodaje swoje własne dodatkowe koszty. `Mutator.Write` jest bardzo powszechną operacją (wprowadzaną przez każde przypisanie), więc dodatkowe jej elementy znacznie zwiększają koszt całego wykonania programu. Co więcej, z punktu widzenia implementacji nie jest oczywiste, gdzie przechowywać liczniki obiektów. Może to być dedykowana przestrzeń w pamięci lub jakiegoś rodzaju nagłówek utrzymywany możliwie blisko samego obiektu. W obu przypadkach nie zmienia to faktu, że każde przypisanie generuje dodatkowe zapisy w pamięci, co jest bardzo niekorzystne. Może to prowadzić też do nieefektywnego wykorzystania pamięci podręcznej procesora, ale tym zagadnieniem zajmiemy się dokładniej w następnym rozdziale.

Jeśli wrócimy do własności osiągalności, o której wspominaliśmy wcześniej, to można powiedzieć, że zliczanie referencji przybliża szacowanie żywotności poprzez lokalne referencje i nie śledzi globalnego stanu obiektowego grafu referencji. W szczególności bez dodatkowych usprawnień może błędnie działać w przypadku referencji cyklicznych. Można je znaleźć w popularnych strukturach danych, takich jak listy dwukierunkowe (zobacz rysunek 1-14). W takim przypadku licznik referencji nigdy nie spadnie do zera, gdyż struktura danych z `value1` i struktura danych z `value2` wskazują na siebie nawzajem.

Tworzenie referencji cyklicznych może być jednak utrudnione na poziomie języka, co jest korzystne. W takim przypadku algorytm zliczania referencji może być używany bez większych obaw o wycieki pamięci wynikające z tego problemu.

Jedną z dużych zalet oraz źródeł popularności zliczania referencji jest fakt, że nie wymaga żadnego wsparcia ze strony środowiska uruchomieniowego. Może być implementowane jako mechanizm dodatkowy dla konkretnych typów w formie biblioteki zewnętrznej. Oznacza to, że możemy pozostawić oryginalne operacje `Mutator.New` i `Mutator.Write` bez zmian i po prostu wprowadzić dla nich odpowiedniki wyższego poziomu, takie jak klasy z odpowiednio przeładowanymi operatorami i konstruktoraми. Tak dzieje się na przykład w przypadku najpopularniejszych implementacji w C++.



Rysunek 1-14 Problem z cyklicznymi referencjami w zliczaniu referencji

Wprowadzone zostały tak zwane *wskaźniki inteligentne* (*smart/intelligent pointers*), które stanowią bardziej skomplikowany sposób zarządzania czasem życia obiektów, na które wskazują. Z punktu widzenia implementacji wskaźniki inteligentne w C++ są w istocie po prostu klasami szablonowymi, które zachowują się jak normalne wskaźniki poprzez odpowiednie przeładowanie operatorów. W przypadku C++ możemy korzystać z ich dwóch rodzajów:

- wskaźnik `unique_ptr`, który spełnia założenie unikalnej własności (wskaźnik jest wyłącznym właścicielem obiektu, który ma być zniszczony, gdy tylko wskaźnik `unique_ptr` wyjdzie poza zasięg albo inny obiekt zostanie do niego przypisany).
- wskaźnik `shared_ptr`, który realizuje założenie zliczania referencji.

Rozwijając nasz kod przykładowy z listingu 1-5 przy użyciu wskaźników inteligentnych, możemy uzyskać kod C++ przedstawiony na listingu 1-10.

Listing 1-10 Przykładowy program w C++ pokazujący zautomatyzowane zarządzanie pamięcią przy użyciu wskaźników inteligentnych

```
#include <iostream>
#include <memory>

void printReport(std::shared_ptr<int> data)
{
    std::cout << "Report: " << *data << "\n";
}

int main()
{
    try
    {
        std::shared_ptr<int> ptr(new int());
        *ptr = 25;
        printReport(ptr);
        return 0;
    }
    catch (std::bad_alloc& ba)
    {
        std::cout << "ERROR: Out of memory\n";
        return 1;
    }
}
```

Gdybyśmy wywołali metodę `data.use_count()` wewnątrz funkcji `printReport`, otrzymalibyśmy wartość 2, ponieważ wewnątrz tej funkcji dwa różne współdzielone wskaźniki wskazują na ten sam obiekt. Z drugiej strony po wyjściu z zakresu bloku `try` metoda `use_count` zwróci 0, ponieważ żadne inteligentne wskaźniki nie wskazują już na nasz obiekt.

Trzeba zwrócić uwagę, że kod z listingu 1-10 nie jest zgodny z dobrymi praktykami języka C++. Przekazywanie inteligentnego wskaźnika tylko w celu odczytania wskazywanych danych powinno raczej odbywać się poprzez stałą referencję (`const&`) a nie wartość, ale to nie zwiększyłoby liczni-
nika referencji; nie byłoby więc przydatne dla naszych celów przykładowych.

Możemy zauważyć dalszą dużą poprawę w takim kodzie, ponieważ:

- Nie musimy ręcznie niszczyć obiektu przy użyciu operatora `delete`.
- Obsługa wyjątków jest uproszczona, ponieważ w razie wyrzucenia jakiegoś wyjątku przez funkcję `printReport()` inteligentny wskaźnik po prostu wyjdzie poza zakres regionu `try` (i wszelkich zawartych zakresów), więc będzie automatycznie zniszczony. Wszystko dzięki wspomnianej wcześniej zasadzie *RAII* (*Resource Acquisition Is Initialization – inicjowanie przy pozyskaniu zasobu*), która zajmuje się czasem życia obiektu w oparciu o zakres zmiennej wskaźnikowej.

Wskaźniki inteligentne mogą być też używane jako pola w klasach, co sprawia, że mogą być mocnymi i przydatnymi narzędziami.

Problem w tym, że wskaźniki inteligentne w C++ zostały wprowadzone na poziomie biblioteki standardowej, a nie samego języka. Inne biblioteki wprowadzały swoje własne implementacje i czasami problematyczne było doprowadzenie do tego, żeby się ze sobą dobrze dogadywały. Qt ma swój typ `QtSharedPointer`, `wxWidgets` ma `wxSharedPtr<T>` itd. Tak już musi być, jeśli nie ma wsparcia kompilatora i języka. Dlatego właśnie automatyczne zarządzanie pamięcią jest tak kluczowe w programowaniu zorientowanym na komponenty⁸, jak w przypadku .NET. Gdy powstawała platforma .NET, przeniesienie odpowiedzialności za zarządzanie pamięcią z programisty na samo środowisko uruchomieniowe było jedną z głównych, kluczowych decyzji projektowych. Wspólna platforma zajmująca się tym, jak obiekty są tworzone, zarządzane i odzyskiwane oznacza, że każdy komponent będzie z tego korzystał w taki sam sposób i nie ma żadnego dodatkowego łącznika pomiędzy komponentami poza samym środowiskiem uruchomieniowym.

Jeśli chodzi o C++, warto zauważyć, że Bjorne zezwolił na bardziej złożone odśmiecanie pamięci w standardzie C++ – nie jest ono zabronione, po prostu nie zostało

8 Składa się to z wielu mniejszych, wymiennych zależności.

jeszcze zaimplementowane. Co więcej, dzięki elastyczności C++ możliwe jest korzystanie z odśmiecania pamięci z wykorzystaniem Memory Pool System albo kolektora Boehm-Demers-Weiser – co wkrótce pokażemy.

Inne języki mogą wprowadzać wskaźniki inteligentne (i zliczanie referencji) bezpośrednio w swoich projektach i tak jest w przypadku Rust – nowoczesnego, niskopoziomowego języka programowania stworzonego przez firmę Mozilla. Wymusza on bezpieczeństwo danych na poziomie kompilacji, wprowadzając pojęcie wskaźników inteligentnych (właściwie kilku różnych ich rodzajów) do języka. Silnie wykorzystuje znaczenie własności i zasadę RAII, co pozwala sprawdzać w czasie kompilacji, czy nie ma naruszeń, takich jak odczytywanie referencji wiszącego wskaźnika. Innym wartym zauważenia użyciem zliczania referencji jest automatyczne zliczanie referencji wbudowane w język Swift.

Krótkie zestawienie wad i zalet zliczania referencji przedstawia się następująco:

Zalety:

- Deterministyczny moment dealokacji – wiemy, że dealokacja nastąpi, gdy licznik referencji spadnie do zera. Dlatego pamięć zostanie odzyskana, jeśli nie jest już dłużej potrzebna.
- Mniejsze wymagania pamięciowe – ponieważ pamięć jest odzyskiwana od razu, gdy obiekty nie są już dłużej wykorzystywane, nie ma obciążenia związanego ze zużyciem pamięci przez obiekty czekające na odśmiecenie.
- Może być zaimplementowane bez żadnego wsparcia ze strony środowiska uruchomieniowego.

Wady:

- Taka naiwna implementacja, jak na listingu 1-8 wprowadza bardzo duże obciążenie dla mutatora.
- Wielowątkowe operacje na licznikach referencji wymagają dobrze przemyślanej synchronizacji, co może wprowadzać dodatkowe obciążenia.
- Bez żadnych dodatkowych usprawnień referencje cykliczne nie mogą być odzyskiwane.

Istnieją ulepszenia dla naiwnych algorytmów zliczania referencji, takie jak *zliczanie referencji z opóźnieniem* (*deferred reference counting*) albo *zespalane zliczanie referencji* (*coalesced reference counting*), które eliminują niektóre z tych problemów kosztem pewnych zalet (głównie natychmiastowego odzyskiwania pamięci). Opisywanie ich tutaj wykracza jednak poza zakres tej książki.

Kolektor śledzący

Ustalanie osiągalności obiektów jest trudne, ponieważ jest to globalny atrybut obiektu (zależy od całego grafu obiektów dla całego programu) a proste, jawne wywołanie mające na celu zwolnienie obiektu jest bardzo lokalne. W tym lokalnym kontekście nie jesteśmy świadomi kontekstu globalnego – czy inne obiekty teraz korzystają z danego obiektu? Zliczanie referencji stara się to przewyciężyć, przyglądając się tylko temu lokalnemu kontekstowi z pewnymi dodatkowymi informacjami – liczbą referencji do obiektu. To oczywiście może prowadzić do problemów z referencjami cyklicznymi i ma inne wady, jak widzieliśmy wcześniej.

Kolektor śledzący (*tracking collector*, mechanizm odśmiecania pamięci ze śledzeniem) jest oparty na znajomości globalnego kontekstu czasu życia obiektu i może podejmować lepsze decyzje odnośnie tego, czy jest dobry moment na usunięcie obiektu (odzyskanie pamięci). Jest to w istocie tak popularne podejście, że zwykle, gdy ktoś wspomina o odśmiecaniu pamięci, to zwykle ma na myśli kolektor śledzący. Możemy go spotkać w środowiskach uruchomieniowych, takich jak .NET, różne implementacje JVM itd.

Kolektor śledzący opiera się głównie na tym, że ustala prawdziwą osiągalność obiektu zaczynając od korzeni mutatora i rekurencyjnie śledząc cały graf obiektów programu. Oczywiście nie jest to trywialne zadanie, ponieważ pamięć procesu może zajmować kilka GB a śledzenie wszystkich referencji między obiektami w tak dużym zbiorze danych może być trudne, zwłaszcza gdy mutatory cały czas działają i zmieniają wszystkie te referencje. Najbardziej typowe podejście do śledzącego odśmiecania pamięci składa się z dwóch głównych kroków:

- *Oznaczanie (Mark)* – podczas tego kroku kolektor określa, które obiekty w pamięci mogą zostać odzyskane przez ustalenie ich osiągalności.
- *Zbieranie (Collect)* – podczas tego kroku kolektor odzyskuje pamięć obiektów, dla których ustalono, że nie są już osiągalne.

Implementacja tej prostej dwufazowej logiki może zostać rozszerzona, tak jak to się dokładnie dzieje w przypadku .NET i można ją wtedy określić jako oznaczanie-planowanie-zamiatanie-kompaktowanie (Mark-Plan-Sweep-Compact). Szczegóły wewnętrznego działania tych faz zobaczymy w następnych rozdziałach. Teraz przyjrzymy się po prostu krokom oznaczania i zbierania w dość ogólnym zakresie, gdyż wywołują one też pewne interesujące problemy.

Faza oznaczania (Mark)

Podczas kroku oznaczania (Mark) kolektor ustala, które obiekty w pamięci powinny być odzyskane przez ustalenie ich osiągalności. Zaczynając od korzeni mutatora, kolektor przechodzi przez cały graf obiektów i oznacza te, które zostały odwiedzone. Te obiekty, które nie są oznaczone na końcu fazy oznaczania, nie są osiągalne. Dzięki oznaczaniu obiektów nie ma problemu z referencjami cyklicznymi. Jeśli podczas przechodzenia przez graf obiektów wrócimy do wcześniej odwiedzonego obiektu, przerywamy dalsze przemierzanie tej ścieżki, ponieważ obiekt jest już oznaczony.

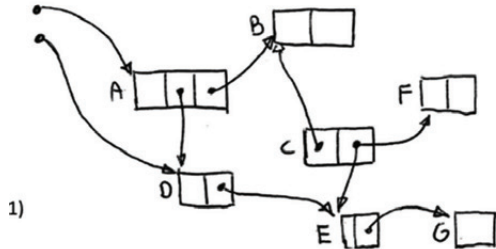
Kilka początkowych kroków takiego algorytmu przedstawiono na rysunku 1-15. Zaczynając od korzeni, podróżujemy po grafie obiektów poprzez referencje między obiektami. Szczegółem implementacyjnym jest, w jakiej kolejności będziemy podróżować po tym grafie. Rysunek 1-15 pokazuje podejście, przy którym zagłębiamy się w graf obiektów i przedstawia trzy możliwe stany każdego obiektu:

- Obiekt jeszcze nieodwiedzony, zaznaczony jako biały prostokąt.
- Obiekt zapamiętany do odwiedzenia, zaznaczony jako jasnoszary prostokąt.
- Obiekt już odwiedzony (oznaczony jako osiągalny), zaznaczony jako ciemnoszary prostokąt.

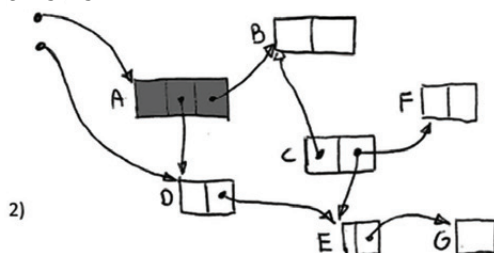
Pierwsze kroki przedstawione na rysunku 1-15 mogą zostać opisane następująco (każdy krok opisuje odpowiadający mu fragment rysunku):

1. Początkowo wszystkie obiekty nie zostały jeszcze odwiedzone.
2. Obiekt A jest dodawany do obiektów przeznaczonych do odwiedzenia jako pierwszy korzeń.
3. Ponieważ obiekt A ma wskaźniki (jako pola) do obiektów B i D, są one dodawane do obiektów przeznaczonych do odwiedzenia. Sam obiekt A na tym etapie jest oznaczany jako osiągalny.
4. Następny obiekt z zestawu obiektów „do odwiedzenia” jest odwiedzany – obiekt B. Ponieważ nie ma żadnych wychodzących referencji, jest po prostu oznaczany jako osiągalny.
5. Następny obiekt z zestawu obiektów „do odwiedzenia” jest odwiedzany – obiekt D. Zawiera pojedynczą referencję do obiektu E, więc obiekt ten jest zapamiętywany jako obiekt przeznaczony do odwiedzenia. Sam obiekt D jest oznaczany jako osiągalny.

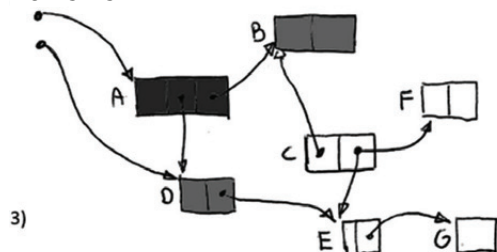
korzenie



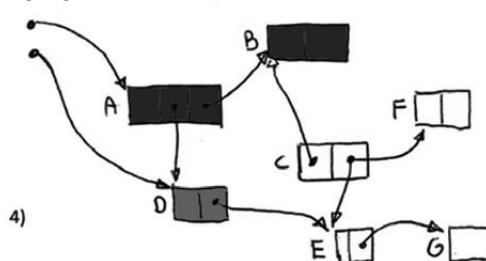
korzenie



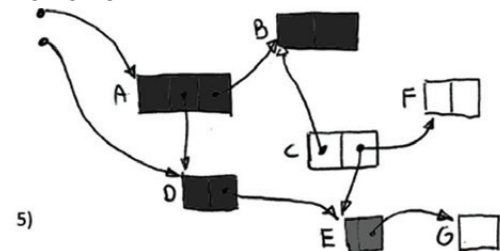
korzenie



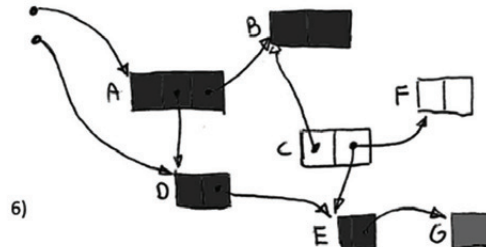
korzenie



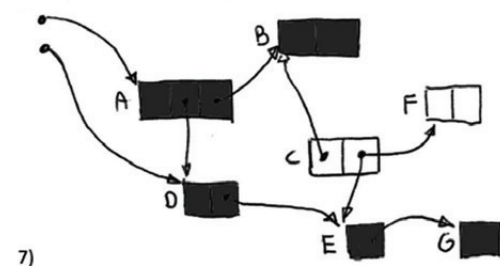
korzenie



korzenie



korzenie



Rysunek 1-15 Kilka pierwszych kroków fazy oznaczania

6. Wychodząca z obiektu E referencja do obiektu G powoduje zapamiętanie tego obiektu jako przeznaczonego do odwiedzenia. Sam obiekt E jest oznaczany jako osiągalny.
7. Ostatni obiekt z zestawu obiektów „do odwiedzenia” jest odwiedzany – obiekt G. Nie zawiera żadnych referencji wychodzących, więc jest po prostu oznaczany jako osiągalny. Na tym etapie nie ma już żadnych obiektów do odwiedzenia, więc zidentyfikowaliśmy, że obiekty C i F nie są osiągalne (są nieżywe).

Oczywiście przechodzenie przez taki graf jest trudne podczas normalnej pracy mutatora, gdyż graf stale się zmienia ze względu na normalne wykonywanie programu – tworzenie nowych obiektów, zmiennych, przypisywanie danych do pól obiektu, itd. Dlatego w niektórych implementacjach odśmiecania pamięci wszystkie mutatory są po prostu zatrzymywane na czas przebiegu fazy oznaczania. Pozwala to na bezpieczne i spójne przechodzenie przez graf. Oczywiście jak tylko wątki wznowią działanie, wiedza zdobyta przez kolektor oparta na grafie obiektów staje się nieaktualna. Nie jest to problem w przypadku nieosiągalnych obiektów – jeśli były wcześniej nieosiągalne, to nigdy nie staną się z powrotem osiągalne. Jednakże istnieje wiele implementacji odśmiecania pamięci, gdzie faza oznaczania jest wykonywana w sposób współbieżny, więc proces oznaczania może przebiegać równocześnie z wykonywaniem kodu mutatora. Jest tak w przypadku popularnych algorytmów, takich jak CMS (Concurrent Mark Sweep – jednoczesne oznaczanie i zmiatanie) w JVM, G1 w JVM oraz w samym .NET. Jak dokładnie takie współbieżne oznaczanie jest implementowane w .NET, opiszemy dokładnie w rozdziale 11.

Istnieje jeden nie całkiem oczywisty problem z fazą oznaczania. Aby śledzić osiągalność obiektów, kolektor powinien być w stanie ustalić korzenie i wiedzieć, gdzie na sterce są umieszczane referencje do innych obiektów. Jest to trywialny problem, jeśli środowisko uruchomieniowe dostarcza takich informacji. Można też go przezwyciężyć w inny sposób.

Konserwatywne odśmiecanie pamięci

Ten typ kolektora może być traktowany jako rozwiązanie prowizoryczne. Może być używany, gdy środowisko uruchomieniowe lub kompilator nie obsługują odśmiecania bezpośrednio przez podawanie dokładnych informacji o typie (układzie obiektu w pamięci), a kolektor nie otrzymuje wsparcia od mutatora przy działaniu na wskaźnikach. Jeśli tak zwany *kolektor konserwatywny* (*conservative collector*) chce dowiedzieć się, które obiekty są osiągalne, skanuje cały stos, obszary danych statycznych i rejestry. Nie mając dodatkowej pomocy, nie wie, co jest wskaźnikiem a co nie – próbuje po prostu zgadywać. Robi to poprzez sprawdzanie kilku rzeczy (wszystko zależy od konkretnej implementacji kolektora), ale najważniejszą jest sprawdzenie, czy interpretacja danego słowa jako adresu (wskaźnika) wskazuje na prawidłowy region sterły zarządzany przez alokator? Jeśli tak, kolektor *konserwatywnie* (stąd

jego nazwa) zakłada, że jest to w istocie wskaźnik. Traktuje go jako odwołanie, które trzeba sprawdzić w ogólnej fazie oznaczania przy przechodzeniu przez graf obiektów.

Oczywiście kolektor może się pomylić w swojej ocenie, co będzie prowadzić do niedokładności – losowe dane mogą wyglądać jak prawidłowy wskaźnik do odpowiedniego adresu. Będzie to prowadzić do utrzymywania pamięci, która mogłaby być odśmiecona (zebrana). Nie jest to bardzo powszechny problem, gdyż większość wartości liczbowych w pamięci jest raczej mała (liczniki, dane finansowe, indeksy), więc jedynym problemem mogą być zagęszczone dane binarne, takie jak bitmapy, liczby zmiennoprzecinkowe lub pewne bloki adresów IP⁹. Istnieją pewne subtelne ulepszenia algorytmu, które pomagają przezwyciężyć ten problem, ale nie będziemy się nimi tutaj zajmować. Co więcej, konserwatywne raportowanie oznacza, że nie jesteśmy w stanie przenosić obiektów w pamięci. Wynika to z tego, że trzeba by aktualizować wskaźniki do przenoszonych obiektów, co oczywiście nie jest możliwe, jeśli nie mamy pewności, czy coś wyglądającego jak wskaźnik jest faktycznie wskaźnikiem.

Kto mógłby więc potrzebować takiego rodzaju kolektor? Jego główną zaletą jest to, że może działać bez wsparcia środowiska uruchomieniowego – w istocie po prostu skanuje pamięć, a więc wsparcie (śledzenie referencji) nie jest potrzebne. Jest to więc na przykład wygodne podejście przy opracowywaniu nowego środowiska uruchomieniowego, gdy pełna informacja o typach dla odśmiecania pamięci nie jest jeszcze opracowana. Rozwijanie pozostałych części systemu może się odbywać bez przeszkód. Gdy zapewnianie odpowiednich informacji o typach będzie już zaimplementowane, można po prostu wyłączyć konserwatywne śledzenie. Firma Microsoft wykorzystywała takie podejście przy opracowywaniu niektórych wersji swojego środowiska uruchomieniowego¹⁰.

Jednakże kolektor konserwatywny wymaga wsparcia alokatora w celu uniknięcia problemów związanych z nieznanym układem pamięci obiektu. Może on na przykład tak organizować alokację obiektów, aby były grupowane w segmentach obiektów o równym rozmiarze. Skanowanie konserwatywne takich regionów jest możliwe, ponieważ granice obiektów są zdefiniowane jako proste wielokrotności rozmiaru obiektów w określonym segmencie.

W wielu językach alokator można zastąpić na poziomie języka (biblioteki), co prowadzi do popularności konserwatywnego odśmiecania pamięci w formie biblioteki.

9 Boehm GC i inne konserwatywne mechanizmy odśmiecania pamięci pozwalają nam alokować blok lub region ze specjalną flagą (jak GC_MALLOC_ATOMIC w przypadku Boehm), która wskazuje kolektorowi, że dany blok nie będzie zawierać żadnych wskaźników i nie powinien być skanowany. Możemy więc wykorzystywać taki blok do przechowywania gęstych danych binarnych, takich jak bitmapy.

10 Interesującym faktem jest to, że .NET zawiera wewnątrz implementację konserwatywnego kolektora, która jest domyślnie wyłączona.

Jedną z najczęściej używanych implementacji niezależnych od API i przeznaczonych dla języków C oraz C++ jest kolektor *Boehm–Demers–Weiser GC* (w skrócie *Boehm GC*).

Był on używany na przykład w środowisku Mono (implementacji CLR z otwartym kodem źródłowym) aż do wersji 2.8 (rok 2010), w której wprowadzono tzw. odśmiecanie pamięci *SGen* – nieco mieszane podejście, które nadal konserwatywnie skanuje stos i rejestry, ale skanowanie sterty jest wspierane przez informacje o typach zapewniane przez środowisko uruchomieniowe.

Podsumujmy krótko główne punkty dotyczące konserwatywnego odśmiecania pamięci:

Zalety:

- Łatwiejsze w środowiskach bez podstawowego wsparcia dla odśmiecania pamięci – na przykład wczesne etapy tworzenia środowiska uruchomieniowego lub języki niezarządzane.

Wady:

- Niedokładność – wszystko, co może wyglądać jak prawidłowy wskaźnik, blokuje odzyskanie pamięci – choć nie jest to typowa sytuacja i może zostać przezwyciężona przez ulepszenie algorytmu i dodatkowe flagi.
- W prostszym podejściu obiekty nie mogą być przenoszone (kompaktowanie) – ponieważ kolektor nie ma pewności, co faktycznie jest wskaźnikiem (a nie może po prostu zaktualizować wartości, co do której jedynie zakłada, że może być wskaźnikiem).

Dokładne odśmiecanie pamięci

W sytuacji tak zwanego dokładnego odśmiecania pamięci jest znacznie prościej, ponieważ kompilator i/lub środowisko uruchomieniowe zapewniają kolektorowi pełne informacje o układzie obiektów w pamięci. Może też obsługiwać tzw. przechodzenie przez stos (*stack crawling*, wyliczanie wszystkich korzeni na stosie). W takim przypadku nie trzeba zgadywać. Zaczynając od dobrze zdefiniowanych korzeni, po prostu skanuje pamięć obiekt po obiekcie. Mając adres pamięci wskazujący na początek obiektu (albo tak zwany wskaźnik wewnętrzny wskazujący na wnętrze obiektu i wiedzę potrzebną do prawidłowej interpretacji takiego odwołania), kolektor po prostu wie, gdzie są umieszczone wychodzące referencje (wskaźniki), więc może rekurencyjnie je sprawdzać podczas przechodzenia przez graf.

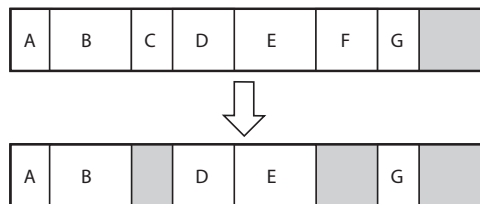
.NET wykorzystuje dokładne odśmiecanie pamięci, więc poznamy jego wewnętrzne działanie w kolejnych rozdziałach. Właściwie całe rozdziały od 7 do 10 będą temu poświęcone.

Faza zbierania (Collect)

Po tym jak kolektor śledzący znajdzie osiągalne obiekty, może odzyskać pamięć ze wszystkich pozostałych, nieżywych obiektów. Faza zbierania kolektora może być zaprojektowana na wiele różnych sposobów ze względu na wiele różnych aspektów. Ciężko opisać wszystkie możliwe kombinacje i warianty w jednym, krótkim akapicie. Należy jednak wyróżnić dwa główne podejścia, na których skupiają się różne implementacje.

Zamiatanie (Sweep)

W tym podejściu nieżywe obiekty są po prostu oznaczane jako wolna przestrzeń, która może być później ponownie wykorzystana. Może to być bardzo szybka operacja, ponieważ (w przykładowej implementacji) wystarczy zmienić pojedynczy bit w bloku pamięci. Taką sytuację pokazano na rysunku 1-16, gdzie nieużywane już obiekty C i F (kontynuując przykład z rysunku 1-15) stają się dostępnymi obszarami pamięci jedynie przez oznaczenie ich jako wolnych obszarów.



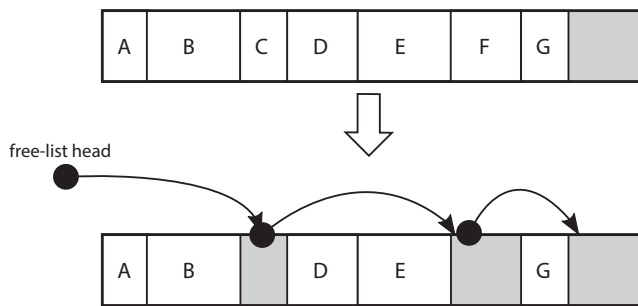
Rysunek 1-16 *Zamiatanie – naiwna implementacja*

Następnie (w takiej naiwnej implementacji) podczas alokacji pamięć jest skanowana w poszukiwaniu pustej luki nie mniejszej niż rozmiar obiektu do utworzenia.

Nietrywialne implementacje mogą budować struktury danych przechowujące informacje o wolnych blokach pamięci w celu ich szybszego uzyskania, zwykle w postaci tzw. *list wolnych obszarów* (*free-lists*, pokazane na rysunku 1-17). Co więcej, listy te muszą być na tyle inteligentne, aby łączyć sąsiadujące ze sobą wolne bloki pamięci. Dalsza optymalizacja może prowadzić do przechowywania zestawów list wolnych obszarów dla luk w pamięci o określonym rozmiarze. Jeśli chodzi o szczegóły implementacyjne, istnieją różne sposoby skanowania takich list. Dwa najpopularniejsze podejścia wykorzystują metody *najlepszego dopasowania* i *pierwszego dopasowania*. W metodzie pierwszego dopasowania zatrzymujemy skanowanie listy wolnych obszarów od razu, gdy zostanie znaleziony jakiś pasujący blok wolnej pamięci. W metodzie najlepszego dopasowania

zawsze skanujemy wszystkie elementy listy wolnych obszarów, aby znaleźć najlepiej pasujący obszar dla wymaganego rozmiaru. Pierwsze podejście jest szybsze, ale może prowadzić do większej fragmentacji, a drugie dokładnie na odwrót.

Choć jest dość szybkie, podejście wykorzystujące zmiatanie ma jedną główną wadę – w końcu prowadzi do większej lub mniejszej fragmentacji pamięci. W miarę jak obiekty są tworzone i niszczone, coraz więcej mniejszych lub większych obszarów wolnej przestrzeni pojawia się na sterzie. Może to prowadzić do sytuacji, gdy mamy dość wolnej pamięci w sumie na nowy obiekt, ale nie ma pojedynczego, ciągłego, wolnego obszaru, który mógłby go pomieścić. Widzieliśmy taką sytuację na rysunku 1-11, gdy opisywaliśmy ogólnie alokację na sterze.

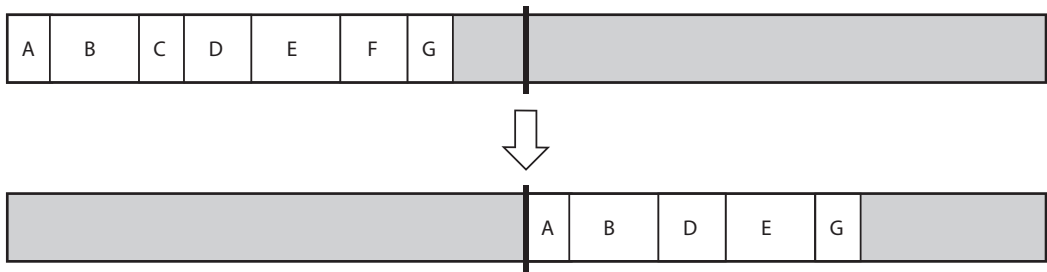


Rysunek 1-17 Zmiatanie – implementacja listy wolnych obszarów

Kompaktowanie (Compact)

W tym podejściu fragmentacja jest eliminowana kosztem mniejszej wydajności, ponieważ wymaga przenoszenia obiektów w pamięci. Obiekty są przenoszone w sposób, który zmniejsza luki utworzone po usuniętych obiektach. Można tutaj wyróżnić dwa różne, główne podejścia.

W prostszym z punktu widzenia implementacji wariantcie *kompaktowania kopiującego* (*copying compacting*) wszystkie żywe (osiągalne) obiekty są kopiowane do innego regionu pamięci za każdym razem, gdy następuje odświeżenie pamięci (zobacz rysunek 1-18). Kompaktowanie jest prostą konsekwencją kopiowania każdego żywego obiektu i ustawiania ich jeden za drugim z pominięciem tych, które już nie są potrzebne. Oczywiście wywołuje to duży ruch w pamięci, gdyż wszystkie żywe obiekty muszą zostać przekopiowane. Obciąża to też znacznie pamięć, ponieważ musimy utrzymywać dwa razy więcej pamięci, niż byłoby to normalnie potrzebne.

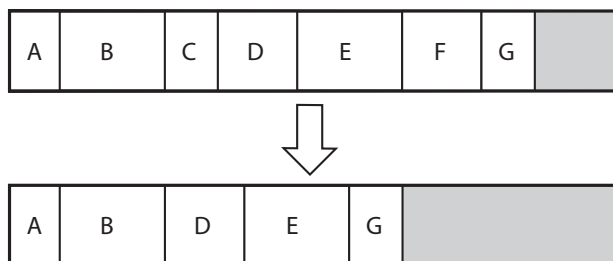


Rysunek 1-18 Kompaktowanie – implementacja kopiująca

Ze względu na te słabości wydawałoby się, że ten algorytm nie ma praktycznego zastosowania. Może jednak być skutecznie używany. Musimy tylko pamiętać o korzystaniu z niego tylko w przypadku pewnych, niewielkich regionów pamięci, a nie dla całej pamięci procesu. Jest tak w przypadku niektórych implementacji JVM, gdy kompaktowanie kopiujące jest wykorzystywane dla mniejszych regionów pamięci.

W bardziej skomplikowanym scenariuszu można zaimplementować *kompaktowanie w miejscu* (*in-place compacting*). Obiekty są przysuwane do siebie tak, aby usuwać luki pomiędzy nimi (zobacz rysunek 1-19). Jest to najbardziej intuicyjne rozwiązanie i przypomina składanie klocków Lego. Z punktu widzenia implementacji nie jest to trywialne, ale nadal wykonalne. Głównym problemem, jaki możemy tu napotkać, jest kwestia, jak obiekty mogą być przesuwane względem siebie bez nadpisywania jednych drugimi i bez użycia jakiegoś tymczasowego bufora?

Jak zobaczymy w rozdziale 9, .NET wykorzystuje to podejście z zastosowaniem bardzo sprytniej struktury danych używanej do optymalizacji, więc tam znajdziemy rozwiązanie powyższej kwestii.



Rysunek 1-19 Kompaktowanie – implementacja w miejscu

Porównanie mechanizmów odśmiecania pamięci

Można by zadać pytanie: który mechanizm odśmiecania pamięci jest lepszy? Czy jest to HotSpot Java 1.8 czy też .NET 4.6? A może Python lub Ruby mają lepsze mechanizmy odśmiecania pamięci? Co właściwie miałyby oznaczać określenie „lepszy mechanizm odśmiecania pamięci”? Pierwszą i najważniejszą zasadą przy porównywaniu algorytmów odśmiecania pamięci jest to, że każde porównanie jest zawsze bardzo niejednoznaczne. Wynika to z tego, że niezwykle trudno jest wydzielić konkretne mechanizmy odśmiecania pamięci i porównywać je jako takie między sobą. Są one bardzo spójne ze środowiskiem uruchomieniowym, więc zasadniczo nie jest możliwe testowanie ich w izolacji. Trudno jest więc uzyskać jakiegokolwiek w pełni obiektywne porównanie. Gdybyśmy chcieli porównać wydajność różnych mechanizmów odśmiecania pamięci – możemy wykorzystać miary, takie jak przepustowość, opóźnienie i czas wstrzymania (zobaczmy różnice pomiędzy tymi pojęciami w rozdziale 3). Wszystkie te pomiary będą dokonywane w kontekście całego środowiska uruchomieniowego, a nie wyłącznie samego mechanizmu odśmiecania pamięci. Mechanizmy platformy albo środowiska uruchomieniowego (na przykład wzorce alokacji, wewnętrzne pule obiektów, dodatkowe kompilacje lub inne ukryte mechanizmy wewnętrzne) mogą mieć zauważalny wpływ na wydajność, tak że sam udział w niej mechanizmów odśmiecania pamięci może być pomijalny. Co więcej, w każdym mechanizmie odśmiecania pamięci istnieje wiele metod ich dostrajania, które sprawiają, że będą sobie radzić lepiej przy określonych rodzajach obciążeń. Niektóre mogą być zoptymalizowane na szybką reakcję w środowisku interaktywnym, a inne na przetwarzanie ogromnych zbiorów danych. Jeszcze inne mogą próbować dynamicznie zmieniać swoją charakterystykę, dopasowując się do bieżącego obciążenia. Co więcej, różne mechanizmy odśmiecania pamięci mogą zachowywać się różnie ze względu na wykorzystywaną konfigurację sprzętową (optymalizacja dla określonej architektury procesora, liczby rdzeni procesora lub architektury pamięci).

Oczywiście możemy porównywać algorytmy wykorzystywane w mechanizmach odśmiecania pamięci i zapewniane przez nie funkcjonalności. Istnieje wiele różnych sposobów kategoryzowania mechanizmów odśmiecania pamięci. Jak już widzieliśmy, definiujemy odśmiecanie pamięci jako konserwatywne (Mono do wersji 2.8) lub dokładne (.NET) albo nawet jako mieszankę obu tych podejść (Mono 2.8+). Jedne implementują zmiatanie, inne kompaktowanie, a jeszcze inne obie te metody. Innym ważnym rozróżnieniem jest to, jak dany mechanizm odśmiecania pamięci dzieli pamięć na partycje. W rozdziale 5 zobaczmy dokładnie, jak sarta może być dzielona na mniejsze części. W niektórych częściach może wykorzystywać zliczanie referencji, a w innych nie. Jak jest zaimplementowany alokator? Czy odśmiecanie pamięci jest równoległe, czy współbieżne (rozdział 11)? Przy tak wielu możliwych różnicach funkcjonalnych naprawdę ciężko powiedzieć, która kombinacja jest „lepszą” – po prostu nie ma jednego idealnego rozwiązania.

Krótkie zestawienie wad i zalet kolektora śledzącego wygląda następująco:

Zalety:

- Zupełna przezroczystość z perspektywy programisty – pamięć jest wyabstrahowana, tak jakby była nieskończona, bez konieczności zajmowania się zwalnianiem pamięci po niepotrzebnych już obiektach.
- Brak problemów z referencjami cyklicznymi.
- Brak dużego obciążenia dla mutatorów.

Wady:

- Bardziej skomplikowana implementacja.
- Niedeterministyczne zwalnianie obiektów – będą zwalniane po pewnym czasie od momentu, gdy staną się nieosiągalne.
- Zatrzymanie środowiska konieczne dla fazy oznaczania – ale tylko w odmianie niewspółbieżnej.
- Większe ograniczenia pamięciowe – jeśli obiekty nie są odzyskiwane tak szybko, jak to konieczne, może to wprowadzać dodatkowe obciążenie pamięci (więcej śmieci żyje przez pewien okres czasu).

Głównie ze względu na pierwszą zaletę, śledzące odśmiecanie pamięci jest tak popularne w różnych środowiskach uruchomieniowych.

Nieco historii

Po zapoznaniu się z solidną dawką podstawowej wiedzy teoretycznej przyjrzyjmy się teraz krótko historii automatycznego zarządzania pamięcią w kontekście różnych języków programowania.

LISP jest jednym z najdłużej istniejących języków z wieloma pojawiającymi się i znikającymi dialektami, z których dwoma popularnymi są Common LISP oraz Scheme. Bez wątpienia jednak najpopularniejszy obecnie jest dialekt znany jako Clojure, który pozwala między innymi kompilować kod do maszyny Java Virtual Machine, środowiska Common Language Runtime (.NET) i języka JavaScript. To sprawia, że jest on bardzo elastyczny i potężny. Spotykają się w nim język LISP i mechanizm odśmiecania pamięci.

Nie tylko jednak język funkcjonalny, taki jak LISP obsługiwał zautomatyzowane zarządzanie pamięcią w czasach swojej popularności. Każda historia związana z językami programowania nie powinna ignorować wpływu innego niezwykle ważnego języka – Simula. Język ten nazywany jest pierwszym w pełni obiektowo zorientowanym językiem,

który wprowadził pojęcia obiektów oraz klas, dziedziczenia, polimorfizmu i inne fundamentalne filary programowania zorientowanego obiektowo. Wszystkie języki, poczynając od Smalltalk, a później od C++, poprzez języki Java i C# aż do języków Python czy Ruby, w jakimś stopniu inspirowały się tym językiem. Co ważne, język Simula 67 oferował automatyczne zarządzanie pamięcią, które było początkowo połączeniem zliczania referencji i śledzącego odśmiecania pamięci, ale w trakcie rozwoju języka zostało zastąpione kolektorem kompaktującym zainspirowanym przez język LISP. Wraz z jego następcą – językiem Smalltalk – odśmiecanie pamięci stało się popularnym wyborem wśród projektantów języków. Rosnąca złożoność oprogramowania doprowadziła projektantów języków do wprowadzenia mniej lub bardziej skomplikowanych sposobów pomagających programistom w zarządzaniu pamięcią.

Popularność WWW i początek ery Internetu w latach 90. XX wieku skierowała tworzenie oprogramowania w stronę programowania wysokopoziomowego. Czasy królowania C i C++ mijały. Ich niskopoziomowa kontrola nad systemem nie miała wartości w kontekście programowania aplikacji WWW oraz masowego rozwoju aplikacji działających po stronie serwera. Wraz z niezwykle szybkim rozwojem Internetu zwiększyła się złożoność aplikacji WWW i potrzeba szybszego tworzenia kodu.

Nie dałoby się opowiedzieć historii automatycznego zarządzania pamięcią bez wspomnienia o języku i platformie Java. Mechanizmy odśmiecania pamięci były jednymi z pierwszych i fundamentalnych założeń, które ten nowy język, planowany przez Sun Microsystems jako „lepszy C++” powinien był spełniać. Począwszy od lat 90., gdy ten projekt wystartował jako wewnętrzny język Oak, zawierał on mechanizmy oznaczania i zmiatania. Pierwsza publicznie dostępna wersja Java 1.0a została ogłoszona w roku 1994. Wraz ze wzrostem popularności języka Java, świadomość istnienia mechanizmów odśmiecania pamięci stale rosła. Od tego czasu automatyczne zarządzanie pamięcią stało się normą dla wszystkich projektantów języków wysokopoziomowych.

Gdy narodził się język Java, powstały dwa inne popularne języki – Python oraz Ruby. Oba języki były wyposażone w automatyczne zarządzanie pamięcią z tych samych powodów, które zostały wspomniane wcześniej. Język Python przed wersją 2.0 miał tylko zliczanie referencji, ale później dołączył bardziej złożone sposoby zajmowania się referencjami cyklicznymi. Język Ruby zapewnia prostszy mechanizm oparty na podejściu „oznacz i zamieć”.

W naszej krótkiej historii nie możemy zignorować języka JavaScript, który pojawił się w tym samym okresie co Java. Choć podobieństwo nazwy do języka Java wynikało bardziej z przyczyn marketingowych niż rzeczywistego podobieństwa, język JavaScript również powstał jako wysokopoziomowy język skryptowy. Nie było w nim miejsca na ręczne zarządzanie pamięcią. Celem było umożliwienie działania zawartości HTML na wysokim poziomie bez zaprzętania sobie głowy takimi aspektami, jak wykorzystanie

pamięci. Środowisko uruchomieniowe JavaScript było odpowiedzialne za te zadania. W kontekście bardziej długotrwałych zastosowań JavaScript – aplikacji jednostronicowych i usług wewnętrznych node.js – znaczenie automatycznego odśmiecania pamięci w silnikach JavaScript staje się coraz ważniejsza. Na przykład bardzo popularny silnik języka JavaScript V8, wykorzystywany przez node.js, stosuje podejście oznaczania, zamiatania i kompaktowania ze swoimi własnymi, dodatkowymi optymalizacjami.

Jak można zauważyć, nawet języki z automatycznym zarządzaniem pamięcią istniały od 50 lat, a faktyczny wzrost ich popularności nastąpił w latach 90. XX wieku. W tym miejscu możemy przejść do historii najważniejszego i najciekawszego dla nas środowiska – historii .NET Framework.

Co ważniejsze, firma Microsoft opracowała w tych czasach swoją własną implementację języka JavaScript o nazwie JScript. Język JScript jest ważną częścią naszej historii, ponieważ stworzył podstawy dla rozwiązań użytych przy tworzeniu .NET. Oczywiście najbardziej ciekawi nas zagadnienie zarządzania pamięcią. Wszystko zaczęło się od napisania języka JScript przez cztery osoby w przeciągu kilku tygodni. Jedną z nich był Patrick Dussud, który może być bez wątpienia nazwany ojcem odśmiecania pamięci w .NET. Napisał on prosty mechanizm konserwatywnego odśmiecania pamięci.

Zanim zaczął pracować nad CLR, Patrick Dussud pracował nad JVM. Firma Microsoft w pewnym momencie poważnie rozważała swoją własną implementację środowiska JVM zamiast tworzenia czegoś nowego, co znamy obecnie jako środowisko uruchomieniowe .NET. Zainspirowany przez JVM i opierając się na już zaimplementowanej wersji dla JScript, napisał on inną wersję mechanizmu konserwatywnego odśmiecania pamięci. Zespół, który w przyszłości częściowo stworzył CLR, szybko odkrył, że JVM wprowadza pewne niewygodne ograniczenia. Po pierwsze, od nowo tworzonego środowiska oczekiwano mocnego wsparcia dla COM i kodu niezarządzanego. Jednym z celów było stworzenie środowiska, w którym ponowna kompilacja programu C++ z nową flagą typu /CLR miała umożliwić uruchamianie go w nowym środowisku. Co więcej, standaryzacja była kłopotliwa i twórcy obawiali się wynikających z niej ograniczeń. Przez moment rozważali wypuszczenie środowiska uruchomieniowego C++ z mechanizmem odśmiecania pamięci w formie rozszerzenia.

Po konsultacjach z przyjacielem (David Moon z firmy Symbolics, zajmujący się generacyjnym odśmiecaniem pamięci) Patrick podjął decyzję o napisaniu „możliwie najlepszego mechanizmu odśmiecania pamięci” od podstaw i zaimplementował prototyp w Common LISP. Dlaczego wybrany został ten język? Był to język, z którym miał do czynienia od wielu lat i w którym po prostu dobrze mu się pracowało. Ponadto miał doświadczenie w korzystaniu z „najlepszych narzędzi do debugowania” na owe czasy. Po napisaniu wersji w LISP napisał konwerter transpilujący¹¹ ten kod na C++. W ten

¹¹ Transpilacja jest kompilacją z jednego kodu źródłowego na inny kod źródłowy.

sposób powstał eksperymentalny mechanizm odśmiecania pamięci dla eksperymentalnej implementacji JVM. Gdy zaczęły się prace nad CLR, część tego eksperymentalnego kodu została wykorzystana w projekcie pisanym od podstaw w C++. Jest więc jedynie legendą opowieść, że kod odśmiecania pamięci w CLR został całkowicie przekonwertowany z języka LISP.

Po zapoznaniu się z podstawami teoretycznymi i małą dawką historii czas na zapoznanie się z pierwszą z wielu zasad, które zostaną wprowadzone w tej książce.

Podsumowanie

W tym rozdziale omówiliśmy bardzo szeroki zakres materiału. Wspominanym tutaj zagadnieniom można by z łatwością poświęcić kilka osobnych książek. Zaczynając od takich podstawowych pojęć jak bity i bajty, poznaliśmy główne rodzaje architektur komputerów – harwardzką i von Neumanna. Poznaliśmy podstawy budowy komputerów, w tym definicje rejestrów, adresów i słów. Po zapoznaniu się z pojęciami statycznej i dynamicznej alokacji, wskaźnika, stosu i sterty przeszliśmy do omówienia najważniejszych pojęć – automatycznego zarządzania pamięcią określanego też jako odśmiecanie pamięci. Przy okazji poznaliśmy też niewygody związane z ręcznym zarządzaniem pamięcią i powody jego automatyzacji. Krótko omówiliśmy pojęcia fundamentalne dla implementacji .NET, takie jak śledzące odśmiecanie pamięci i jego fazy oznaczania, zmiatania i kompaktowania. Przyjrzymy się im bliżej w kolejnych rozdziałach tej książki. Wszystko o czym mówiliśmy, zostało pokazane w odniesieniu do historii i w ogólniejszym kontekście, co pozwoliło nam spojrzeć na to zagadnienie z szerszej perspektywy.

Wiedza, którą tutaj zdobyliśmy, pozwoli nam lepiej zrozumieć kolejne rozdziały. Z rozdziału na rozdział będziemy się zbliżać do kwestii praktycznej implementacji środowiska .NET. Jednakże bez zrozumienia szerszego kontekstu przedstawionego w tym rozdziale, nie byłoby to pełne spojrzenie. Zapraszam teraz do rozdziału 2, gdzie przejdziemy od podstaw teoretycznych do podstaw niskopoziomowego projektowania komputerów i pamięci.

Zasada 1 – Ucz się

Zastosowanie: Tak ogólna, jak to tylko możliwe.

Uzasadnienie: Najbardziej ogólna zasada w tej książce. Ma zastosowanie w znacznie szerszym zakresie niż tylko zarządzanie pamięcią. Oznacza po prostu, że powinniśmy zawsze nastawiać się na poszerzanie swojej wiedzy, aby dążyć do bycia lepszymi

profesjonalistami. Wiedza nie przychodzi sama z siebie. Musimy na nią zasłużyć. Jest to żmudny, czasochłonny i mozolny proces. Dlatego musimy się stale motywować. Czy tak oczywista prawda zasługuje na osobną zasadę? Myślę, że tak. W codziennym życiu można o tym łatwo zapomnieć. Wydaje się nam, że codzienne zadania mogą nas czegoś nauczyć. Do pewnego stopnia tak z pewnością jest. Ale oczywiste jest, że w celu wyjścia ze sfery komfortu musimy podjąć kilka kroków. Świadomie. To oznacza sięgnięcie po książkę, obejrzenie prezentacji w Internecie, przeczytanie artykułu. Możliwości są liczne i nie ma sensu wszystkich ich tutaj wymieniać. Jednakże jest to tak podstawowa kwestia, że musi znaleźć się na liście zasad każdego profesjonalisty. Jeśli ktoś nie jest przekonany, warto zainteresować się pojęciem rzemiosła programistycznego i jego manifestem dostępnym pod adresem <http://manifesto.softwarecraftsmanship.org>. Jestem też dużym fanem pojęcia *wyczucia mechanicznego* (*mechanical sympathy*), które pochodzi od kierowcy rajdowego Jackiego Stewarta:

Nie trzeba być inżynierem, aby być kierowcą rajdowym, ale trzeba mieć wyczucie mechaniczne.

To pojęcie zostało później wprowadzone do świata IT przez Martina Thompsona. Co ono oznacza? Oczywiście nie trzeba być mechanikiem, aby być kierowcą rajdowym. Ale bez pewnego głębszego zrozumienia, jak działa samochód, jakie są jego mechanizmy, jak funkcjonuje silnik, jakie siły na niego wpływają – ciężko jest być naprawdę dobrym kierowcą wyścigowym. Trzeba „czuć samochód”, aby harmonijnie z nim współpracować. Trzeba mieć wyczucie mechaniczne. Tak samo jest w przypadku nas programistów. Oczywiście możemy skupiać się tylko na platformach, takich jak .NET lub JVM i na tym poprzestać. Ale wtedy będziemy jedynie niedzielnymi kierowcami, oglądającymi samochód z perspektywy kierownicy i kilku pedałów.

Jak stosować: W tak ogólnej zasadzie nie ma jednego prostego podejścia. Można czytać książki dotyczące działania komputera lub wybranej platformy. Można korzystać z wielu dostępnych w sieci usług szkoleniowych. Można oglądać prezentacje lub uczestniczyć w konferencjach i lokalnych grupach użytkowników. Można zacząć blog i pisać o takich tematach, ponieważ nie ma lepszego sposobu na nauczenie się czegoś, niż uczenie tego innych. Jest tak wiele możliwości, że nawet nie będę próbował wymienić ich wszystkich. Trzeba mieć tylko na uwadze motto „ucz się” i starać się implementować tę zasadę w swoim życiu!

Rozdział 2

Niskopoziomowe zarządzanie pamięcią

Aby zrozumieć, jak działa zarządzanie pamięcią, musimy uzyskać szerszy kontekst. W poprzednim rozdziale poznaliśmy teoretyczne podstawy tego zagadnienia. Możemy teraz bezpośrednio przejść do omówienia szczegółów działania automatycznego zarządzania pamięcią, funkcjonowania mechanizmu odśmiecania pamięci oraz rozważań, gdzie mogą występować wycieki pamięci. Jeśli jednak naprawdę chcemy „poczuć” ten temat, dobrze byłoby poświęcić kilka chwil na przypomnienie sobie jeszcze jednego aspektu tego zagadnienia. Pozwoli nam to lepiej zrozumieć różne decyzje projektowe, które zostały podjęte przez twórców mechanizmów odśmiecania pamięci w .NET (a także w innych zarządzanych środowiskach uruchomieniowych). Twórcy takich mechanizmów nie żyją w próżni i muszą adaptować się do istniejących okoliczności, ograniczeń i mechanizmów rządzących sprzętem komputerowym i systemami operacyjnymi. Tym aspektem zajmiemy się teraz.

Zapraszam więc do rozdziału, w którym poznamy te mechanizmy i ograniczenia. Oczywiście zagadnienia te są same w sobie na tyle rozbudowane, że można by im poświęcić kilka dużych, oddzielnych książek. Skupimy się tylko na podstawach, mniej lub bardziej luźno powiązanych z zarządzaniem pamięcią. Prawdę mówiąc, nie jest łatwo przedstawić takie obszerne zagadnienia w sposób, który nie będzie zbyt przytłaczający i nie będzie obejmował zbyt wielu mniej znaczących kwestii. Jednocześnie chciałem przedstawić to na tyle szczegółowo, aby faktycznie dało się dostrzec wynikający z tego wpływ na zarządzanie pamięcią w .NET. Zapraszam do lektury!

Zapoznanie się z tymi wszystkimi szczegółami pozwoli nam lepiej uporać się ze złożonością zagadnienia zarządzania pamięcią. Nawet jeśli w codziennym zarządzaniu pamięcią kojarzymy je jedynie z wywoływaniem operatora `new`, to dobrze jest mieć świadomość, ile różnych mechanizmów istnieje i na jak wielu poziomach. Sprzęt, system

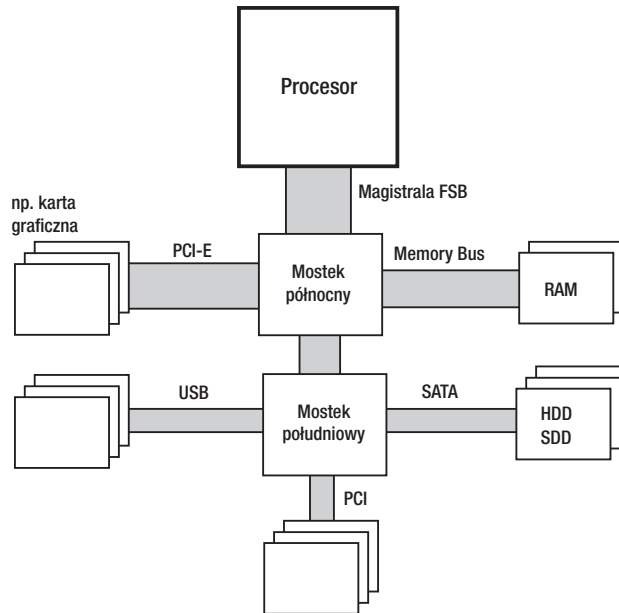
operacyjny, kompilator – wszystkie te elementy wpływają na to, jak działa zarządzanie pamięcią i jak środowisko .NET zostało napisane, choć nie zawsze jest to oczywiste. Ta wiedza jest bardzo zgodna z duchem *wyczucia mechanicznego* (*mechanical sympathy*) przedstawionego w poprzednim rozdziale. Mam też nadzieję, że po prostu przyjemnie będzie się zapoznać z pewnymi drobnymi faktami, o których tutaj wspomnimy.

To powiedziawszy, można spokojnie potraktować ten rozdział jako opcjonalny. Zapewnia on mnóstwo informacji teoretycznych, które, choć pomocne w zrozumieniu zagadnienia zarządzania pamięcią, nie są konieczne do zrozumienia pozostałych części tej książki. Jeśli więc ktoś się spieszy albo po prostu chce przejść od razu do bardziej praktycznych informacji i przykładów na temat .NET, to może jedynie przejrzeć lub nawet całkiem pominąć ten rozdział (aby ewentualnie wrócić do niego w bardziej sprzyjającym czasie).

Sprzęt

Jak działa nowoczesny komputer? Wydaje się, że każdy programista w jakimś stopniu będzie w stanie odpowiedzieć na to pytanie. Jeśli ktoś studiował informatykę, być może będzie mógł sobie przypomnieć jakieś informacje z wykładów. Jeśli ktoś jest samoukiem, to pewnie coś gdzieś czytał na ten temat. Moglibyśmy wyrecytować z pamięci następujące fakty: komputer składa się z procesora, który jest główną jednostką przetwarzającą, czyli wykonującą programy. Ma dostęp do pamięci RAM (która jest szybka) i dysków twardych (które są wolne). Jest też karta graficzna, która jest bardzo ważna dla graczy (i różnego rodzaju projektantów grafiki) i która jest odpowiedzialna za generowanie obrazu wyświetlanego na monitorze. Takie ogólne spojrzenie na to zagadnienie nie wystarczy jednak do naszych celów. Musimy zająć się tym tematem dokładniej. Dla potrzeb naszych rozważań pozwolę sobie wprowadzić architekturę nowoczesnego komputera w formie poniższego diagramu na rysunku 2-1.

Rynek nowoczesnych komputerów osobistych jest zdominowany przez komputery PC i Mac. Na ich podstawie opracowałem schematyczny, ogólny diagram architektury komputera. Jeśli to będzie potrzebne, będę wspominał o pewnych możliwych niuansach, takich jak procesory ARM lub bardziej złożone maszyny serwerowe.



Rysunek 2-1 Architektura komputera – procesor, pamięć RAM, mostek północny, mostek południowy, itd. Szerokość magistrali odpowiada proporcjonalnie ilości przekazywanych danych (mniej więcej).

Możemy wymienić następujące główne składniki architektury typowego komputera:

- *Procesor (CPU, central processing unit* – centralna jednostka przetwarzająca) – główna jednostka odpowiadająca za wykonywanie instrukcji. Widzieliśmy to już w rozdziale 1. Tutaj znajdują się takie elementy, jak jednostki arytmetyczno-logiczne (ALU), jednostki zmiennoprzecinkowe (FPU) rejestry oraz *potoki wykonawcze* instrukcji – odpowiedzialne za sprawne wykonywanie instrukcji dzielonych na zbiory mniejszych operacji i wykonywanych (jeśli to możliwe) równolegle.
- *Magistrala FSB (Front Side Bus)* – magistrala danych łącząca CPU z mostkiem północnym.
- *Mostek północny (Northbridge)* – jednostka, która zawiera głównie kontroler pamięci odpowiedzialny za sterowanie komunikacją pomiędzy pamięcią a procesorem.
- *Pamięć RAM (Random Access Memory* – pamięć o dostępie swobodnym) – główna pamięć komputera. Przechowuje dane i kod programów, gdy włączone jest zasilanie – dlatego też określana jest mianem *dynamicznej pamięci RAM (DRAM)* albo *pamięci ulotnej (volatile memory)*.

- *Magistrala pamięciowa (Memory Bus)* – magistrala danych, która łączy pamięć RAM z mostkiem północnym.
- *Mostek południowy (Southbridge)* – układ obsługujący wszystkie funkcje wejścia/wyjścia (I/O) komputera, takie jak USB, dźwięk, wyjścia szeregowo, BIOS systemu, szynę ISA, kontroler przerwań oraz kanały IDE – kontrolery pamięci masowej, takie jak PATA i/lub SATA.
- *Pamięć masowa (Storage I/O)* – nieulotna pamięć, która przechowuje dane, w tym popularne dyski HDD lub SSD.

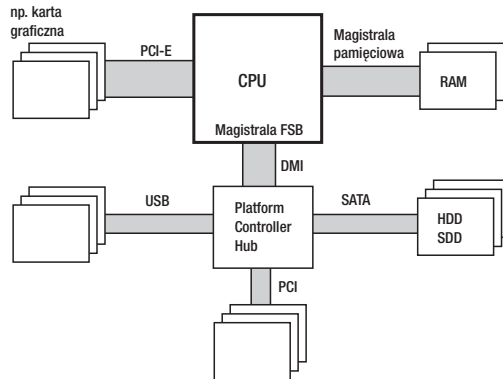
Warto wspomnieć, że dawniej układy CPU, mostka północnego i mostka południowego były osobnymi układami scalonymi, ale obecnie są ściśle zintegrowane. Mikroarchitektury Nehalem (firmy Intel) i Zen (firmy AMD) obejmują mostek północny, który w takim przypadku jest często określany jako *uncore* (funkcje poza rdzeniem) albo *agent systemowy* (*system agent*). Ta ewolucja architektury została pokazana na rysunku 2-2.

Taka integracja pomaga, ponieważ kontroler pamięci (wewnątrz mostka północnego) umieszczony blisko jednostek wykonawczych procesora ogranicza opóźnienia dzięki mniejszym fizycznie odległościom i ulepszonym współdziałaniu. Nadal jednak istnieją na rynku procesory (z których najpopularniejsza jest rodzina AMD FX), które mają oddzielone układy CPU, mostka północnego i mostka południowego.

Głównym problemem związanym z zarządzaniem pamięcią jest rozbieżność pomiędzy wydajnością obecnych procesorów a podsystemami pamięci wewnętrznej i pamięci masowej. Procesor jest znacznie szybszy niż pamięć, więc każdy dostęp do pamięci wprowadza niepożądane opóźnienia. Gdy procesor musi czekać na dostęp do danych w pamięci (odczyt lub zapis), nazywamy to *zwłoką*. Im częściej taka zwłoka będzie występować, tym gorsze będzie wykorzystanie procesora, gdyż jego moc będzie tracona na oczekiwanie na dane.

Typowy obecny procesor działa z częstotliwością 3 GHz lub większą. Natomiast pamięć działa z wewnętrznym zegarem o częstotliwościach rzędu jedynie 200–400 MHz. Różnica w wydajności jest ogromna. Budowanie układów pamięci RAM działających z częstotliwością procesora byłoby zbyt drogie. Wynika to z tego, jak są budowane nowoczesne pamięci RAM – ładowanie i rozładowywanie wewnętrznych kondensatorów zabiera czas, który jest bardzo trudno ograniczyć.

Można być zaskoczonym informacją, że pamięć działa z tak niskimi częstotliwościami. Właściwie w sklepach komputerowych kupujemy moduły pamięci oznaczone częstotliwościami typu 1600 lub 2400 MHz, które są znacznie bliższe prędkościom procesorów. Skąd się biorą takie liczby? Jak zobaczymy, takie specyfikacje są tylko częścią bardziej złożonej prawdy.



Rysunek 2-2 Nowoczesny sprzęt – procesor z mostkiem północnym w środku, pamięć RAM, mostek południowy, nazywany w terminologii firmy Intel PCH – Platform Controller Hub (koncentrator kontrolerów platformy), itd. Szerokość magistrali odpowiada proporcjonalnie ilości przekazywanych danych (mniej więcej).

Taka integracja pomaga, ponieważ kontroler pamięci (wewnątrz mostka północnego) umieszczony blisko jednostek wykonawczych procesora ogranicza opóźnienia dzięki mniejszym fizycznie odległościom i ulepszonym współdziałaniu. Nadal jednak istnieją na rynku procesory (z których najpopularniejsza jest rodzina AMD FX), które mają oddzielone układy CPU, mostka północnego i mostka południowego.

Głównym problemem związanym z zarządzaniem pamięcią jest rozbieżność pomiędzy wydajnością obecnych procesorów a podsystemami pamięci wewnętrznej i pamięci masowej. Procesor jest znacznie szybszy niż pamięć, więc każdy dostęp do pamięci wprowadza niepożądane opóźnienia. Gdy procesor musi czekać na dostęp do danych w pamięci (odczyt lub zapis), nazywamy to *zwłoką*. Im częściej taka zwłoka będzie występować, tym gorsze będzie wykorzystanie procesora, gdyż jego moc będzie tracona na oczekiwanie na dane.

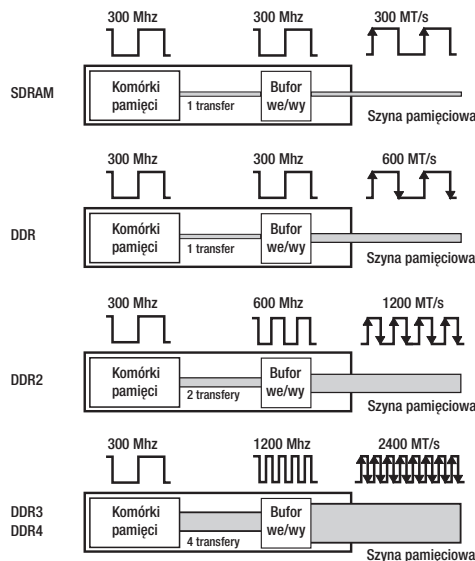
Typowy obecny procesor działa z częstotliwością 3 GHz lub większą. Natomiast pamięć działa z wewnętrznym zegarem o częstotliwościach rzędu jedynie 200–400 MHz. Różnica w wydajności jest ogromna. Budowanie układów pamięci RAM działających z częstotliwością procesora byłoby zbyt drogie. Wynika to z tego, jak są budowane nowoczesne pamięci RAM – ładowanie i rozładowywanie wewnętrznych kondensatorów zabiera czas, który jest bardzo trudno ograniczyć.

Można być zaskoczonym informacją, że pamięć działa z tak niskimi częstotliwościami. Właściwie w sklepach komputerowych kupujemy moduły pamięci oznaczone częstotliwościami typu 1600 lub 2400 MHz, które są znacznie bliższe prędkościom procesorów. Skąd się biorą takie liczby? Jak zobaczymy, takie specyfikacje są tylko częścią bardziej złożonej prawdy.

Moduł pamięci składa się z *wewnętrznych komórek pamięciowych* (przechowujących dane) i dodatkowych buforów do przewyżczenia ich ograniczeń związanych z niskimi wewnętrznymi częstotliwościami zegara. Wykorzystywane są pewne dodatkowe sztuczki (zobacz rysunek 2-3). Większość z nich opiera się na mnożeniu odczytów danych:

- Dwukrotne przesyłanie danych z wewnętrznej komórki pamięciowej w pojedynczym cyklu zegara. Dokładnie rzecz biorąc, ma to miejsce zarówno na opadającej, jak i wznoszącej się krzywej sygnału. Stąd nazwa zdecydowanie najbardziej popularnej pamięci różnych generacji – DDR (*Double Data Rate* – podwójne tempo danych). Ta technika jest też określana *podwójnym pompowaniem* (*double-pumping*).
- Wykorzystywanie wewnętrznego buforowania do dokonywania kilku odczytów jednocześnie w jednym cyklu zegara pamięci. Pozwala to nam pomnożyć ilość zapewnianych danych widzianych z zewnątrz w porównaniu z ilością danych przychodzących z częstotliwością wewnętrzną. Interfejs pamięci DDR2 podwaja częstotliwość zewnętrznego zegara, natomiast DDR3 i DDR4 zwiększają ją czterokrotnie.

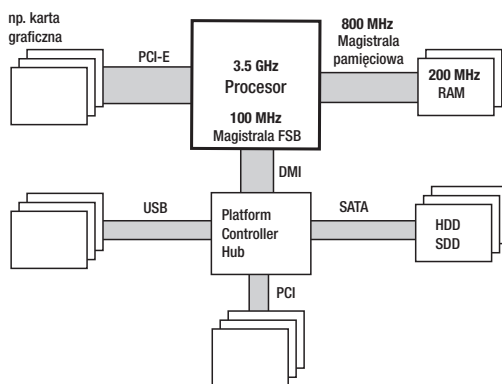
Te techniki są obecnie wykorzystywane w modułach DDR w przeciwieństwie do znacznie prostszych modułów SDRAM (synchroniczna pamięć DRAM) wykorzystywanych w przeszłości.



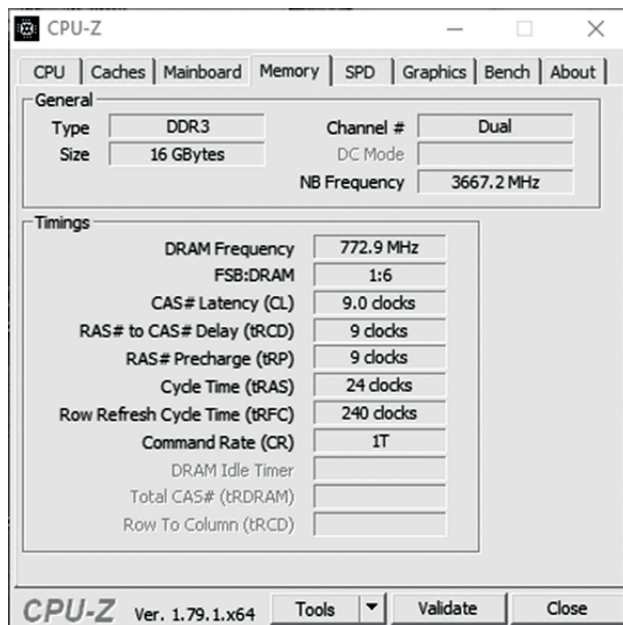
Rysunek 2-3 Wewnętrzna budowa pamięci SDRAM, DDR, DDR2, DDR3, DDR4. Przykład modułów pamięci z wewnętrznym zegarem 300 MHz. MT/s oznacza „mega-transfery na sekundę”.

Przyjrzyjmy się typowemu układowi pamięci DDR4 o parametrach 16 GB 2400 MHz (opisywanemu w specyfikacjach jako DDR4-2400, PC4-19200). W takim przypadku wewnętrzny zegar pamięci DRAM działa z częstotliwością 300 MHz. Zegar magistrali pamięciowej jest zwiększany czterokrotnie do 1200 MHz dzięki wewnętrznemu buforowi wejścia/wyjścia. Dodatkowo, ponieważ w każdym cyklu zegara mamy dwa transfery (oba zbocza sygnału zegara), otrzymujemy wskaźnik transferu danych na poziomie 2400 MT/s (mega-transfery na sekundę). Stąd właśnie bierze się podawana w specyfikacji częstotliwość 2400 MHz. Upraszczając, ze względu na naturę podwójnego pompowania w pamięci DDR, prędkość jest zwykle podawana jako podwojona częstotliwość zegara wejścia/wyjścia, która jest z kolei wielokrotnością wewnętrznego zegara pamięci. Podawanie tej wartości w MHz jest jedynie marketingowym uproszczeniem. Druga sygnatura – PC4-19200 – bierze się z maksymalnej teoretycznej wydajności takiej pamięci – jest to 2400 MT/s pomnożone przez 8 bajtów (transferowane jest pojedyncze słowo o długości 64 bitów), daje to wynik 19200 MB/s.

Przyjrzyjmy się dla przykładu mojemu biurkowemu komputerowi PC w kontekście całej architektury. Jest on wyposażony w procesor Intel Core i7-4770K (generacja Haswell) działający z częstotliwością 3,5 GHz. Częstotliwość magistrali Front Side Bus wynosi jedynie 100 MHz. Wykorzystywana pamięć DDR3-1600 (PC3-12800) ma wewnętrzny zegar pamięci 200 MHz, a dzięki mechanizmowi DDR3 zegar magistrali wejścia/wyjścia ma częstotliwość 800 MHz. Zostało to zilustrowane na rysunku 2-4. Możemy to wszystko potwierdzić, wykorzystując narzędzia do diagnostyki sprzętu, takie jak CPU-Z (zobacz rysunek 2-5).



Rysunek 2-4 Przykład nowoczesnej architektury sprzętowej (Intel Core i74770K oraz DDR3-1600)



Rysunek 2-5 Zrzut ekranu z narzędzia CPU-Z – karta Memory pokazuje częstotliwości mostka północnego (NB Frequency) i pamięci DRAM (DRAM Frequency) wraz ze stosunkiem częstotliwości FSB:DRAM (który niestety nie jest poprawnie podawany w tej wersji narzędzia i powinien wynosić 1:8)

Niezależnie od wszystkich opisanych tutaj ulepszeń pamięci DDR, procesory są wciąż szybsze niż wykorzystywana przez nie pamięć. Żeby przezwyciężyć ten problem, podobne podejście jest stosowane na wielu poziomach – polega ono na przenoszeniu niektórych elementów danych bliżej elementu, który ma bardziej wydajne (i droższe) jednostki pamięciowe. Takie podejście jest określane *przechowywaniem w pamięci podręcznej* lub *buforowaniem (caching)*.

W przypadku pamięci masowej, takiej jak dyski HDD, dane są zwykle buforowane w pamięci RAM – albo w szybszej, lecz mniejszej dedykowanej pamięci masowej, takiej jak niewielki układ SSD wewnątrz hybrydowego napędu dedykowany najczęściej używanym danym. W przypadku pamięci RAM dane są buforowane wewnątrz pamięci podręcznej procesora, co wkrótce dokładnie zobaczymy.

Oczywiście istnieją bardziej ogólne optymalizacje pamięci RAM obejmujące lepsze zaprojektowanie sprzętu, lepsze kontrolery pamięci i optymalizacje DMA (Direct Memory Access – bezpośredni dostęp do pamięci) dla urządzeń. Jednakże nie będziemy się w tej książce zajmować techniką DMA, gdyż nie jest bezpośrednio związana z danymi programu, a regiony te nie są zarządzane przez mechanizmy odświeżania pamięci.

Pamięć

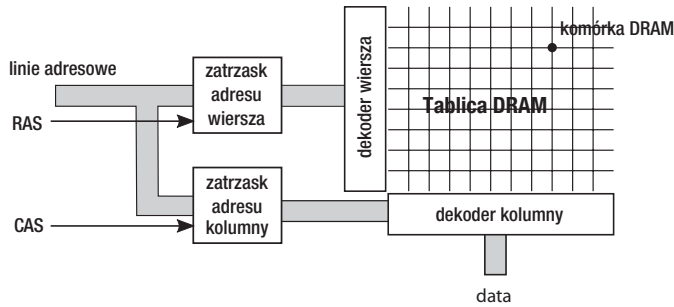
Wyczerpująca książka na temat pamięci komputerowej powinna przynajmniej pobieżnie uwzględniać, jak obecne pamięci są fizycznie zbudowane. Niektórzy mogą być zaskoczeni pewnymi faktami, które zostaną podane w tym podrozdziale. Mam nadzieję, że pozwoli to wszystkim lepiej zrozumieć, dlaczego nowoczesne komputery mają taką, a nie inną architekturę.

Istnieją obecnie dwa główne typy pamięci spotykane w komputerach osobistych i różnią się one znacząco, jeśli chodzi o koszt produkcji i wykorzystania oraz wydajność:

- *Pamięci SRAM (Static Random Access Memory)* – zapewniają bardzo szybki dostęp, ale są dość złożone, składając się z 4-6 tranzystorów na komórkę pamięci (przechowującą pojedynczy bit). Przechowują dane tak długo, jak jest dostępne zasilanie i nie wymagają odświeżania. Ze względu na wysoką prędkość działania są głównie używane w pamięciach podręcznych procesorów.
- *Pamięci DRAM (Dynamic Random Access Memory)* – bardzo prosta konstrukcja komórki (dużo mniejsza niż w przypadku SRAM) składa się z pojedynczego tranzystora i kondensatora. Z powodu upływności kondensatora komórka wymaga stałego odświeżania (co zajmuje cenne milisekundy i opóźnia odczyty pamięci). Sygnał odczytywany z kondensatora musi też być wzmacniany, co dodatkowo komplikuje sprawy. Odczyty i zapisy zajmują nieco czasu i nie są liniowe ze względu na opóźnienia kondensatora (pewien czas jest potrzebny do uzyskania prawidłowego odczytu lub udanego zapisu).

Poświęćmy jeszcze kilka słów technologii DRAM, ponieważ jest ona podstawą powszechnie wykorzystywanej pamięci instalowanej w naszych komputerach. Jak wspomnieliśmy, pojedyncza komórka DRAM składa się z tranzystora oraz kondensatora i przechowuje pojedynczy bit danych. Takie komórki są grupowane w *tablice DRAM*. Adres dostępu do określonej komórki jest zapewniany przez tzw. *linie adresowe (address lines)*.

Byłoby to bardzo skomplikowane i kosztowne, gdyby każda komórka w tablicy pamięci DRAM miała swój własny adres. Na przykład w przypadku adresowania 32-bitowego potrzebny byłby *dekoder linii adresowych (address lines decoder)*, składnik odpowiedzialny za wybór określonej komórki o szerokości 32 bitów. Liczba linii adresowych wpływa w znacznym stopniu na ogólny koszt systemu – im więcej linii, tym więcej pinów i wzajemnych połączeń pomiędzy kontrolerem pamięci a układami (modułami) pamięci RAM. Byłoby to tym bardziej zbyt kosztowne i skomplikowane w przypadku komputerów opartych na słowach 64-bitowych. Z tego powodu linie adresowe są wykorzystywane zarówno jako linie wierszy, jak i linie kolumn (zobacz rysunek 2-6), a zapewnianie pełnego adresu jest podzielone na dwie fazy.



Rysunek 2-6 Przykład układu pamięci DRAM z tablicą DRAM i najważniejszymi kanałami: liniami adresowymi, RAS i CAS

W pojedynczej tablicy linia adresu (wiersza) wybiera wiersz, a linia danych (kolumny) wybiera kolumnę. Pojedynczy bit z określonej komórki jest odczytywany w następującym procesie:

1. Numer wiersza jest umieszczany na liniach adresowych.
2. Interpretacja jest włączana przez sygnał RAS na dedykowanej linii.
3. Numer kolumny jest umieszczany na liniach adresowych.
4. Interpretacja jest włączana przez sygnał CAS.
5. Pobranie danych – pojedynczego bitu (określona komórka DRAM została zaadresowana).

Moduły DRAM, które instalujemy w swoich komputerach, składają się z wielu takich tablic DRAM zorganizowanych w sposób pozwalający nam na dostęp do wielu bitów (pojedynczego słowa) w pojedynczym cyklu zegara.

Czasy przejścia pomiędzy poszczególnymi krokami uzyskania tego pojedynczego bitu silnie wpływają na wydajność pamięci. Czasy te mogą być znajome, ponieważ są one ważnym czynnikiem w specyfikacjach modułów pamięciowych i znacznie wpływają na ich ceny. Wiele osób pewnie kojarzy te czasy podawane jako liczby typu DDR3 9-9-9-24 w specyfikacjach modułów DIMM. Wszystkie te czasy określają cykle zegara wymagane do przeprowadzenia określonego działania. Kolejne z tych liczb mają następujące znaczenie:

- tCL (opóźnienie CAS) – czas pomiędzy bramkowaniem adresu kolumny (CAS – column address strobe) a początkiem odpowiedzi (otrzymywaniem danych).
- tRCD (opóźnienie pomiędzy RAS i CAS) – minimalny czas pomiędzy możliwym wystąpieniem bramkowania adresu wiersza (RAS) i bramkowania adresu kolumny (CAS).

- tRP (wstępne ładowanie wiersza) – czas potrzebny na *wstępne ładowanie* wiersza przed możliwością dostępu do niego. Wiersz nie może zostać użyty bez wcześniejszego przygotowania, które jest zwane wstępnym ładowaniem.
- tRAS (opóźnienie aktywnego wiersza) – minimalny czas, przez który wiersz musi być aktywny, aby można było uzyskać z niego informacje. Jest to zwykle co najmniej suma trzech poprzednich czasów.

Warto zwrócić uwagę na znaczenie tych czasów. Jeśli wiersz i kolumna, którymi jesteśmy zainteresowani, zostały już ustawione, to odczyt jest niemal natychmiastowy. Jeśli chcemy zmienić kolumnę, to zajmie to tCL cykli zegara. Jeśli chcemy zmienić wiersz, sytuacja jest dużo gorsza. Najpierw musi on zostać wstępnie załadowany (tRP cykli), po czym nastąpią opóźnienia RAS i CAS (tCL i tRCD cykli).

Wszystkie te czasy są ważne dla użytkowników komputerów oczekujących maksymalnej wydajności. Zwłaszcza gracze zwracają dużą uwagę na te parametry. Przy zakupie modułów pamięciowych wystarczy wiedzieć, że powinniśmy zwracać uwagę na najniższe możliwe czasy, jeśli wydajność jest absolutnie priorytetowa.

Ciekawi nas jednak wpływ architektury i czasów opóźnień pamięci DRAM na zarządzanie pamięcią. Jak widać, największym kosztem jest zmiana wiersza – czasy sygnału RAS i wstępnego ładowania. Jest to jeden z wielu powodów, dlaczego sekwencyjne wzorce dostępu do pamięci są znacznie szybsze od niesekwencyjnych. Odczytywanie danych hurtem z pojedynczego wiersza (przy zmienianiu tylko kolumny) jest znacznie szybsze niż konieczność częstej zmiany wiersza. Jeśli wzorzec dostępu jest całkowicie losowy, najprawdopodobniej będziemy mieli do czynienia z długotrwałymi zmianami wierszy przy każdym dostępie do pamięci.

Wszystkie przedstawione tutaj informacje mają jeden cel – uzmysłwić, dlaczego niesekwencyjny dostęp do pamięci jest tak niepożądany. Jak jeszcze zobaczymy, nie jest to jedyny powód, dlaczego całkiem losowy dostęp jest najgorszym scenariuszem.

Procesor (CPU)

Przejdźmy teraz do omówienia centralnej jednostki przetwarzającej. Procesor jest kompatybilny z tzw. *architekturą zestawu instrukcji* (ISA – *Instruction Set Architecture*) – definiuje ona między innymi zestaw operacji, które mogą być wykonywane (instrukcje), rejestry i ich znaczenie, sposób adresowania pamięci, itd. W tym znaczeniu ISA jest kontraktem (interfejsem) określanym pomiędzy producentem procesora a jego użytkownikami – programami pisanymi zgodnie z danym kontraktem. Jest to warstwa, którą w programowaniu widzimy na przykład w języku asemblera dla danej architektury. Najczęściej wykorzystywanymi architekturami w świecie ekosystemu .NET

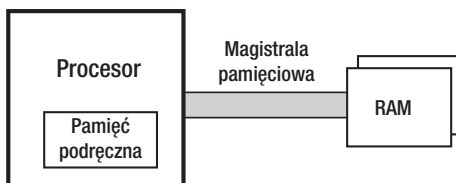
są ISA IA-32 (32-bitowe procesory i386, 32-bitowe procesory Pentium) oraz zgodna z AMD64 (większość nowoczesnych procesorów, w tym Intel Core, AMD FX i Zen, itd.). Architektura ISA jest implementowana przez tzw. *mikroarchitekturę* procesora. Pozwala nam to ulepszać mikroarchitekturę bez wpływania na system i oprogramowanie, a więc w sposób kompatybilny wstecznie.

Uwaga Jest sporo zamieszania z nazwami standardów architektur 64-bitowych i często można spotkać wymiennie używane określenia x86-64, EMT64T, Intel 64 lub AMD64. Pomimo obecności w nich nazw producentów i czasami drobnych różnic możemy założyć na potrzeby tej książki, że można bezpiecznie używać tych nazw zamiennie.

Jak stwierdziliśmy w poprzednim rozdziale, kluczową rolę w działaniu procesora zajmują rejestry, ponieważ obecnie wszystkie komputery są implementowane jako maszyny rejestrowe. W kontekście manipulacji danymi dostęp do rejestrów jest natychmiastowy, w tym sensie, że odbywa się w pojedynczym cyklu procesora i nie wprowadza żadnych dodatkowych opóźnień. Żadne miejsce na dane nie jest położone bliżej procesora niż same rejestry procesora. Oczywiście rejestry przechowują dane potrzebne dla bieżących instrukcji, więc nie mogą być traktowane jako pamięć ogólnego przeznaczenia. W istocie procesor ma zwykle więcej rejestrów, niż mogłoby to się wydawać na podstawie jego architektury ISA. Pozwala to na różnego rodzaju optymalizacje (jak w przypadku tzw. *przemianowywania rejestrów*). Są to jednak szczegóły implementacyjne mikroarchitektury i nie wpływają na mechanizmy zarządzania pamięcią.

Pamięć podręczna procesora

Jak wspominaliśmy wcześniej, w celu złagodzenia różnicy wydajnościowej pomiędzy procesorem a pamięcią RAM wykorzystywany jest składnik pośredni do przechowywania kopii najczęściej używanych i najbardziej potrzebnych danych – pamięć podręczna procesora. Zostało to w bardzo ogólny sposób zilustrowane na rysunku 2-7.



Rysunek 2-7 *Procesor z pamięcią podręczną i relacja z pamięcią RAM*

Pamięć podręczna jest transparentna z punktu widzenia ISA. Programista (a nawet system operacyjny) nie musi koniecznie wiedzieć o jej istnieniu. Nie muszą nią zarządzać. W idealnym świecie właściwe użycie i zarządzanie pamięcią podręczną powinno leżeć w wyłącznej gestii procesora.

Ponieważ chcemy wykorzystywać możliwie najszybszą pamięć jako pamięć podręczną, używane są w tym celu wcześniej wspomniane układy SRAM. Ze względu na koszt i rozmiar (zajmuje cenną przestrzeń wewnątrz procesora) wynikające z tej technologii, pamięć podręczna nie może oczywiście mieć tak wielkich pojemności, jak główna pamięć RAM. W zależności od przyjętych kosztów może odpowiadać szybkością procesorowi lub może być o rząd lub dwa rzędy wielkości wolniejsza.

Trafienia i chybień pamięci podręcznej

Zasada stojąca za pamięcią podręczną jest trywialna. Gdy instrukcja wykonywana przez procesor musi uzyskać dostęp do pamięci (czy to będzie zapis, czy odczyt), najpierw sprawdzana jest pamięć podręczna, czy potrzebne dane są już w niej dostępne. Jeśli tak, to świetnie! Właśnie uzyskaliśmy bardzo szybki dostęp do pamięci, a taka sytuacja jest określana jako *trafienie w pamięci podręcznej* – *cache hit*. Jeśli dane nie są w pamięci podręcznej (tzw. *chybienie w pamięci podręcznej* – *cache miss*), to są tam zapisywane po odczytaniu z pamięci RAM, co jest oczywiście znacznie wolniejszą operacją. *Współczynnik trafień w pamięci podręcznej* (*cache hit ratio*) i *współczynnik chybień w pamięci podręcznej* (*cache miss ratio*) są bardzo ważnymi wskaźnikami informującymi nas, czy nasz kod efektywnie korzysta z pamięci podręcznej.

Lokalność danych

Dlaczego pamięć podręczna jest w ogóle tak przydatna? Pomysł pamięci podręcznej opiera się na bardzo ważnym pojęciu – *lokalności danych* (*data locality*). Możemy rozróżnić dwa rodzaje lokalności:

- *Lokalność czasowa* (*temporal locality*) – jeśli uzyskiwaliśmy dostęp do pewnego regionu pamięci, to prawdopodobnie będziemy sięgać do niego ponownie w bliskiej przyszłości. To całkowicie usprawiedliwia wykorzystanie pamięci podręcznej – odczytujemy pewne dane z pamięci i prawdopodobnie będziemy z nich ponownie korzystać później kilka razy. Dlaczego występuje lokalność czasowa? Jest to dość intuicyjne. Rzadko wykorzystujemy jakieś dane tylko raz. Zwykle ładujemy jakieś struktury danych do zmiennych i później wielokrotnie z tych zmiennych korzystamy. Są to wszelkiego rodzaju liczniki, dane tymczasowe wczytywane z plików, itd.

- *Lokalność przestrzenna (spatial locality)* – jeśli uzyskujemy dostęp do jakiegoś regionu w pamięci, to prawdopodobnie będziemy też uzyskiwać dostęp do danych w jego bliskim sąsiedztwie. Ten typ lokalności może stać się naszym sojusznikiem, jeśli będziemy przechowywać w pamięci podręcznej nieco więcej otaczających danych niż aktualnie potrzebujemy. Na przykład, jeśli potrzebujemy kilku bajtów z pamięci, wczytajmy je i zapiszmy w pamięci podręcznej wraz z kilkoma sąsiednimi. To też jest raczej intuicyjne. Rzadko wykorzystujemy bardzo odizolowane niewielkie obszary pamięci. Wkrótce odkryjemy, że stos i sarta są zorganizowane w segmenty, więc wątki podczas swojego działania zwykle uzyskują dostęp do zgrupowanych obszarów pamięci. Lokalne zmienne lub struktury danych są również zwykle umieszczane blisko siebie.

Warto zwrócić uwagę, że pamięć podręczna będzie skuteczna, jeśli powyższe warunki faktycznie będą spełnione. Jest to jednak broń obosieczna. Jeśli napiszemy program w sposób naruszający lokalność danych, pamięć podręczna stanie się niepotrzebnym obciążeniem. Zobaczmy to później w tym rozdziale.

Implementacja pamięci podręcznej

Ponadto, o ile utrzymywana jest zgodność z modelem pamięci ISA, szczegóły implementacji pamięci podręcznej są teoretycznie nieistotne. Powinna po prostu istnieć, aby przyspieszać dostęp do pamięci i to wszystko. Jednakże jest to doskonały przykład wymyślane go przez Joela Spolsky'ego *prawa wyciekających abstrakcji (the law of leaky abstractions)*:

Wszystkie nietrywialne abstrakcje do pewnego stopnia wyciekają.

Oznacza to, że abstrakcja, która teoretycznie powinna ukrywać szczegóły implementacyjne, w pewnych okolicznościach niestety ukazuje je na zewnątrz. Robi to też zwykle w nieprzewidywalny i/lub niepożądany sposób. Jak to się dzieje w przypadku pamięci podręcznej, powinno wkrótce się wyjaśnić, ale na razie zajmijmy się nieco dokładniej szczegółami implementacyjnymi.

Najważniejszym i najbardziej wpływowym faktem jest to, że dane pomiędzy pamięcią RAM a pamięcią podręczną są transferowane w blokach zwanych *linią pamięci podręcznej (cache line)*. Linia pamięci podręcznej ma określony rozmiar i w większości obecnych komputerów wynosi on 64 bajty. Ważne, aby zapamiętać, że nie możemy odczytać lub zapisać mniej danych z pamięci niż rozmiar linii pamięci podręcznej, a więc 64 bajty. Nawet gdybyśmy chcieli odczytać tylko pojedynczy bit z pamięci, to zapelniony zostanie cały blok linii pamięci podręcznej o szerokości 64 bajtów. Taki sposób działania lepiej wykorzystuje sekwencyjny dostęp do pamięci DRAM (pamiętajmy o opóźnieniach wstępnego ładowania i RAS opisanych wcześniej w tym rozdziale).

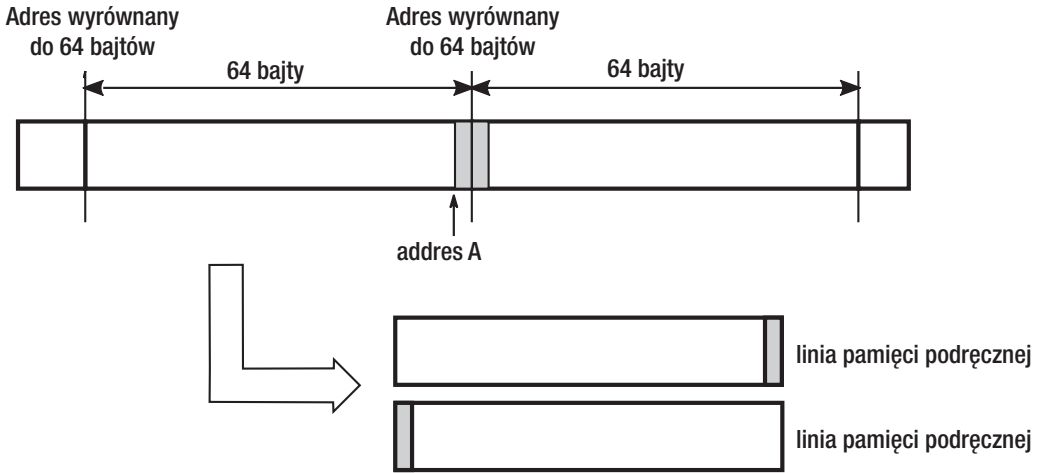
Dostęp do DRAM ma szerokość 64 bitów (8 bajtów), więc do wypełnienia takiej linii pamięci podręcznej wymagane jest osiem transferów z pamięci RAM. To wymaga całkiem sporo cykli procesora, więc istnieją różne techniki to uwzględniające. Jedną z nich jest technika *wczytywania krytycznego słowa najpierw i wczesnego wznawiania*. Sprawia ona, że linia pamięci podręcznej nie wczytuje danych słowo po słowie, ale zaczyna od słowa, które jest najbardziej potrzebne. Wyobraźmy sobie, że w najgorszym przypadku takie 8-bajtowe słowo byłoby na końcu linii pamięci podręcznej, więc musielibyśmy czekać na wszystkie siedem poprzednich transferów, aby uzyskać do niego dostęp. Ta technika najpierw wczytuje najważniejsze słowo. Instrukcje oczekujące na te dane mogą być dalej wykonywane, a reszta linii pamięci podręcznej zostanie wypełniona asynchronicznie.

Ze względu na linie pamięci podręcznej wszystkie dane przechowywane w pamięci są wyrównywane do granic 64-bajtowych obszarów. W najbardziej pesymistycznym scenariuszu przy odczytywaniu dwóch kolejnych bajtów trzeba zużyć dwie linie pamięci podręcznej o całkowitym rozmiarze 128 bajtów. Dane trafią do pamięci podręcznej, ale jeśli żadne inne dane z tego regionu pamięci nie będą potrzebne, będzie to strata czasu. Zilustrowano to na rysunku 2-8, gdzie chcemy odczytać tylko 2 bajty spod adresu A. Niestety adres A znajduje się o 1 bajt przed końcem granicy linii pamięci podręcznej, więc w istocie trzeba będzie wczytać całe dwie linie pamięci podręcznej.

Uwaga Jak wygląda typowy wzorzec dostępu do pamięci? Gdy ktoś chce odczytać dane z pamięci, odpowiednia linia jest tworzona w pamięci podręcznej i wczytywane są do niej 64 bajty danych. Gdy ktoś chce zapisać dane w pamięci, pierwszy krok jest dokładnie taki sam – linia pamięci podręcznej jest wypełniana, o ile jej tam jeszcze nie ma. Te zbuforowane dane są modyfikowane w pamięci podręcznej, gdy ktoś zapisuje dane. Teraz mogą wystąpić dwie strategie:

- *Przepisanie (write-through)* – po dokonaniu zapisu w linii pamięci podręcznej zmodyfikowane dane są natychmiast zapisywane w pamięci głównej. Jest to proste podejście do zaimplementowania, ale powoduje duże obciążenie magistrali pamięciowej.
- *Zapisanie z opóźnieniem (write-back)* – po dokonaniu zapisu w linii pamięci podręcznej jest ona oznaczana jako zmodyfikowana. Później, gdy zabraknie miejsca w pamięci podręcznej na inne dane, ten zmodyfikowany blok jest zapisywany w pamięci głównej (a wpis w pamięci podręcznej jest usuwany). Procesor może zapisywać te bloki od czasu do czasu, jeśli uzna to za stosowne (np. w czasie bezczynności).

Istnieje jeszcze jedna technika optymalizacyjna nazywana *zapisywaniem kombinowanym (write-combining)*. Zapewnia ona, że dana linia pamięci podręcznej z danego obszaru pamięci jest zapisywana w całości (zamiast zapisywania jej poszczególnych słów), wykorzystując fakt szybszego dostępu sekwencyjnego do pamięci.



Rysunek 2-8 Dostęp do dwóch kolejnych bajtów wymaga wypełnienia dwóch linii pamięci podręcznej ze względu na ich niefortunne położenie

Choć jest to właściwie wierzchołek góry lodowej, można by zapytać, dlaczego musimy znać takie szczegóły implementacji sprzętowej? Czy ma to rzeczywiście znaczenie w wygodnym świecie kodu zarządzanego? Kontynuujmy naszą podróż, aby się o tym przekonać.

Koszt niesekwencyjnych wzorców dostępu do pamięci został zilustrowany przez kod przykładowy na listingu 2-1 oraz wyniki w tabeli 2-1. Ten program przykładowy uzyskuje dostęp do tej samej dwuwymiarowej tablicy na dwa sposoby – wiersz po wierszu i kolumna po kolumnie. Wyniki zostały przedstawione dla trzech różnych środowisk: PC (Intel Core i7-4770K 3,5GHz), laptop (Intel Core i7-4712MQ 2,3GHz) oraz płyta Raspberry Pi 2 (ARM Cortex-A7 0,9GHz).

Listing 2-1 Indeksowanie według kolumn i według wierszy przy dostępie to tablicy (tablica liczb całkowitych o rozmiarach 5000x5000)

```
int[, ] tab = new int[n, m];
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < m; ++j)
    {
        tab[i, j] = 1;
    }
}
```

```

int[,] tab = new int[n, m];
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < m; ++j)
    {
        tab[j, i] = 1;
    }
}

```

Tabela 2-1 Wyniki indeksowania według kolumn i według wierszy ($n, m = 5000$)

Wzorzec	PC	Laptop	Raspberry Pi 2
Według wierszy	52 ms	127 ms	918 ms
Według kolumn	401 ms	413 ms	2001 ms

Ten przykład pokazuje, jak niekorzystne dla wydajności może być niesekwencyjne pobieranie danych. Program przykładowy w drugiej wersji wczytuje dane kolumna po kolumnie. W efekcie musimy co jakiś czas zmieniać aktywną linię komórek DRAM. Co więcej bardzo źle wykorzystujemy pamięć podręczną, ponieważ odczytujemy tylko jeden bajt danych, wczytując całą linię pamięci podręcznej. Następnie od razu odczytujemy dane z innego, odległego adresu, więc inna linia pamięci podręcznej musi zostać wypełniona. Różnica w wydajności może być sześciokrotna, jak widać w tabeli 2-1! Procesor zatrzymuje się bardzo często w oczekiwaniu na dostęp do pamięci.

Rysunek 2-9 ilustruje różnicę pomiędzy dostępem do elementów według wierszy i według kolumn dla pewnej niewielkiej tablicy zawierającej wartości od 1 do 40 (a ilustracja zakłada, że cztery takie wartości mieszczą się w pojedynczej linii pamięci podręcznej). Dla celów ilustracyjnych założmy też, że dostęp do tablicy z rysunku 2-9 odbywa się na procesorze z buforem mieszczącym jedynie cztery linie pamięci podręcznej¹. Gdy odczytujemy pamięć wiersz po wierszu (lewa strona rysunku 2-9), to w istocie odczytujemy kolejne liczby całkowite wewnątrz następujących po sobie regionów pamięci:

- Aby odczytać pierwsze cztery elementy (1, 2, 3, 4), wczytywana jest pierwsza linia pamięci podręcznej i wszystkie te elementy są wykorzystywane.
- Aby odczytać kolejne cztery elementy (5, 6, 7, 8), wczytywana jest druga linia pamięci podręcznej i ponownie wszystkie te elementy są wykorzystywane.

¹ W rzeczywistym procesorze „buforem” dla linii pamięci podręcznej jest cała pamięć podręczna procesora, więc zwykle mieści setki lub tysiące wpisów o rozmiarze linii pamięci podręcznej (64 bajty).