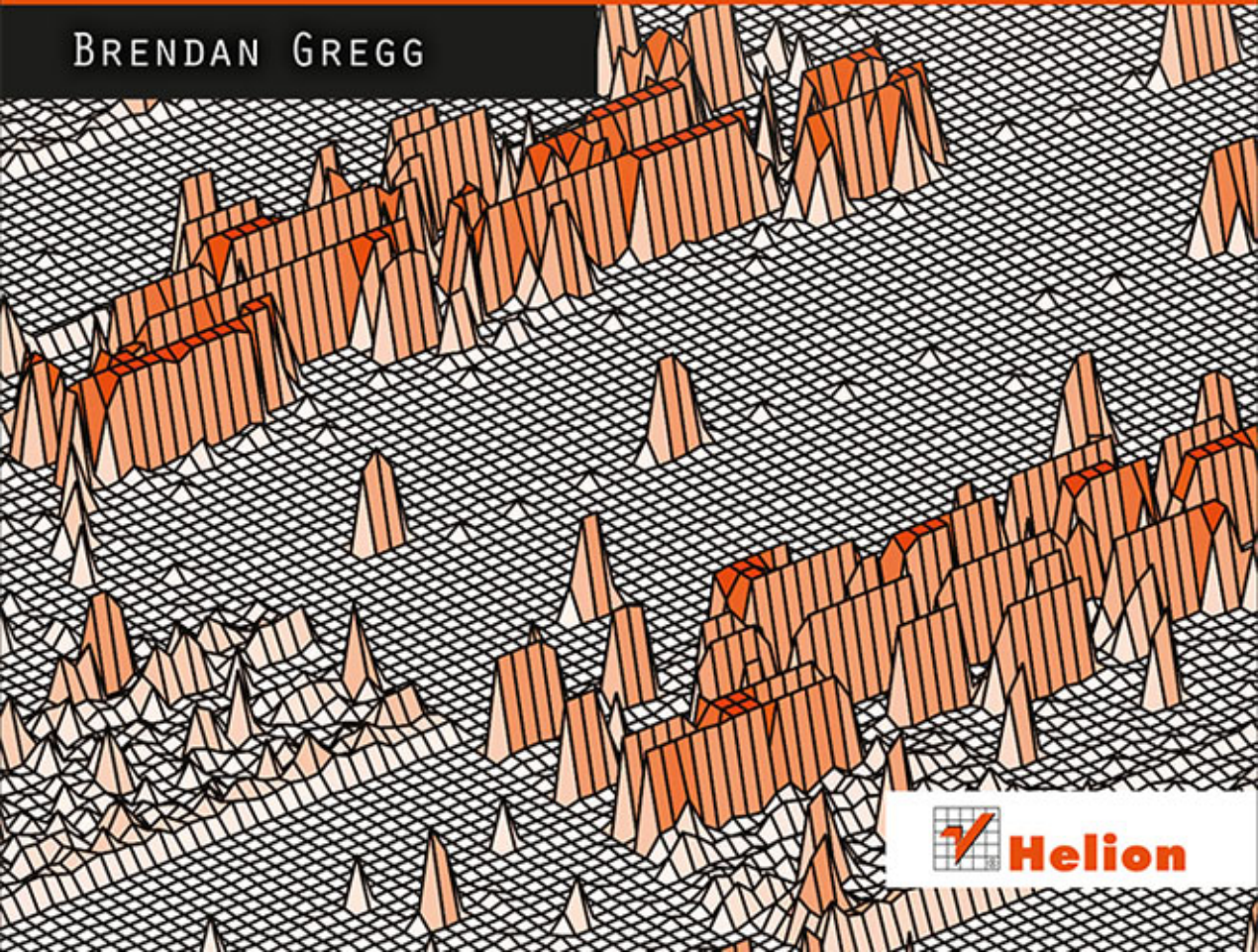
PRENTICE HALL

Wydajne systemy komputerowe

Przewodnik dla administratorów systemów lokalnych i w chmurze

Brendan Gregg



Helion

Tytuł oryginału: Systems Performance: Enterprise and the Cloud

Tłumaczenie: Robert Górczyński

ISBN: 978-83-246-9157-9

Authorized translation from the English language edition, entitled: SYSTEMS PERFORMANCE: ENTERPRISE AND THE CLOUD; ISBN 0133390098;
by Brendan Gregg; published by Pearson Education, Inc; publishing as Prentice Hall.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wysyko
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

5

Aplikacje

Najlepsze miejsce dostosowania wydajności znajduje się w pobliżu wykonywanej pracy, to znaczy w aplikacjach. Obejmują one bazy danych, serwery WWW, serwery aplikacji, programy przeznaczone do równoważenia obciążenia, serwery plików itd. W kolejnych rozdziałach będziemy zajmować się aplikacjami z perspektywy konsumowanych przez nie zasobów: procesora, pamięci, systemów plików, dysków i sieci. Natomiast w tym rozdziale przyjrzymy się samym aplikacjom.

Aplikacje mogą być niezwykle skomplikowane, zwłaszcza w środowiskach aplikacji rozproszonych, składających się z wielu komponentów. Analiza wewnętrznych składników aplikacji to najczęściej zadanie dla programistów aplikacji, którzy w tym celu mogą korzystać z opracowanych przez firmy trzecie narzędzi introspekcji. Z kolei dla osób zajmujących się wydajnością systemów, między innymi administratorów systemu i analityków wydajności aplikacji, oznacza to przeprowadzenie konfiguracji aplikacji w taki sposób, aby jak najlepiej wykorzystać zasoby systemowe. Ponadto konieczne jest przygotowanie charakterystyki pokazującej, jak aplikacja używa systemu, oraz przeprowadzenie analizy najczęściej występujących patologii.

W tym rozdziale koncentrujemy się na podstawach aplikacji, fundamentalnych zasadach dotyczących ich wydajności, językach programowania, kompilatorach i strategiach stosowanych podczas ogólnej analizy wydajności aplikacji.

5.1. Podstawy dotyczące aplikacji

Przed przejściem do zagadnień związanych z wydajnością aplikacji w pierwszej kolejności należy poznać rolę aplikacji, jej podstawową charakterystykę oraz ekosystem w przemyśle. W ten sposób powstanie kontekst, w którym będziesz mógł poznać działanie aplikacji. Zyskasz ponadto możliwość przybliżenia sobie najczęściej występujących kwestii z zakresu wydajności, a także możliwość dostosowania wydajności aplikacji oraz przeprowadzenia dalszej analizy. Aby dowiedzieć się, jak wygląda wspomniany kontekst aplikacji, spróbuj udzielić odpowiedzi na następujące pytania:

- **Funkcja**. Jaka jest rola aplikacji? Czy to jest serwer bazy danych, serwer WWW, program do równoważenia obciążenia, serwer plików itd.?
- **Operacja**. Jakiego rodzaju operacje i żądania są wykonywane przez aplikacje? W przypadku baz danych będą to *zapytania* (i *polecenia*), z kolei serwery WWW wykonują *żądania HTTP* itd. Operacje mogą być mierzone jako częstotliwość, co pozwala sprawdzić obciążenie i zaplanować pojemność.
- **Tryb procesora**. Na jakim poziomie została zaimplementowana aplikacja: jako oprogramowanie użytkownika czy jądra? Większość aplikacji jest przeznaczona do działania na poziomie użytkownika, jest uruchamiana jako jeden proces lub więcej procesów. Niektóre aplikacje są jednak implementowane jako usługi dla jądra, na przykład NFS.
- **Konfiguracja**. W jaki sposób jest skonfigurowana aplikacja i dlaczego właśnie tak? Tego rodzaju informacje można znaleźć w pliku konfiguracyjnym lub za pomocą narzędzi administracyjnych. Sprawdź, czy zmianie uległy jakiekolwiek modyfikowalne parametry związane z wydajnością, między innymi wielkości buforów, pamięci podręcznej, możliwość równoczesnego działania (procesów lub wątków) bądź też inne opcje.
- **Metryki**. Czy aplikacja dostarcza jakichkolwiek metryk, na przykład częstotliwości operacji? Tego rodzaju metryki mogą być podawane w postaci dołączonych narzędzi, opracowanych przez firmę trzecią, żądań API lub wskutek przetwarzania dzienników zdarzeń generowanych podczas działania aplikacji.
- **Dzienniki zdarzeń**. Czy aplikacja tworzy dzienniki zdarzeń? Tworzenie jakich dzienników zdarzeń włączono w aplikacji? Jakie metryki dotyczące wydajności, na przykład opóźnienie, są dostępne we wspomnianych dziennikach zdarzeń? Na przykład baza danych MySQL zawiera *dziennik zdarzeń rejestrujący wolno wykonywane zapytania*. Dostarcza on niezwykle cennych informacji szczegółowych o wydajności każdego zapytania, którego wykonanie trwało dłużej niż zdefiniowany czas.
- **Wersja**. Czy aplikacja jest w najnowszej wersji? Czy w informacjach o nowym wydaniu znajdują się jakiekolwiek wzmianki o poprawkach lub usprawnieniach w zakresie wydajności jej działania?
- **Błędy**. Czy dla aplikacji istnieje baza danych pozwalająca na zgłaszanie błędów? Czy w tej bazie danych znajdują się informacje o błędach „wydajności" w używanej wersji aplikacji? Jeżeli uważasz, że wydajność działania aplikacji

jest niezadowalająca, przejrzyj tego rodzaju bazę błędów i zobacz, czy jakakolwiek podobna kwestia została wcześniej zgłoszona, jak była analizowana oraz co zrobiono, aby naprawić błąd.

- **Społeczność**. Czy dla danej aplikacji istnieje społeczność, z którą można dzielić się odkrytymi problemami dotyczącymi wydajności? Społeczność może obejmować między innymi fora, blogi, czaty (IRC), spotkania i konferencje. Po spotkaniach i konferencjach najczęściej są zamieszczane w internecie prezentacje i materiały wideo, które jeszcze przez wiele lat mogą stanowić użyteczne zasoby. Ponadto może istnieć tak zwany *menedżer społeczności*, który będzie informował o uaktualnieniach i nowościach.
- **Książki**. Czy danej aplikacji poświęcono jakiekolwiek książki, na przykład dotyczące wydajności?
- **Eksperci**. Czy istnieją jacykolwiek uznani eksperci dla danej aplikacji? Poznanie ich nazwisk może pomóc w wyszukaniu przygotowanych przez nich materiałów.

Niezależnie od źródła Twoim celem jest ogólne poznanie aplikacji — do czego jest przeznaczona, w jaki sposób działa oraz jaką wydajność oferuje. Wręcz nieocenionym zasobem jest *wykres funkcjonalny* (o ile będziesz mógł taki znaleźć), pokazujący wewnętrzne komponenty aplikacji.

W kolejnych punktach zostaną przedstawione następne podstawowe informacje dotyczące aplikacji: definiowania celów, optymalizacji, monitorowania oraz notacji „duże O”.

5.1.1. Cele

Dzięki zdefiniowaniu celów w zakresie wydajności można wyznaczyć kierunek prowadzonej analizy wydajności oraz wybrać odpowiednie kroki, które trzeba będzie podjąć. Bez wyraźnego zdefiniowania celów istnieje ryzyko, że analiza wydajności zamieni się w błądzenie po omacku w nadziei na znalezienie odpowiedniego rozwiązania.

W przypadku wydajności aplikacji pracę można rozpocząć od sprawdzenia operacji wykonywanych przez aplikację (jak wcześniej opisano) oraz zdefiniowania celów wydajności. Wspomnianymi celami mogą być:

- **Opóźnienie**. Maksymalne skrócenie czasu udzielania odpowiedzi przez aplikację.
- **Przepustowość**. Uzyskanie jak największej częstotliwości wykonywania operacji lub największego możliwego transferu danych.
- **Wykorzystanie zasobów**. Efektywność dla danego obciążenia aplikacji.

Znacznie lepiej będzie, jeśli cele wydajności okażą się możliwe do zmierzenia za pomocą metryk spełniających wymagania biznesowe i dotyczące jakości usługi, na przykład:

- Średnie opóźnienie w udzielaniu odpowiedzi przez aplikację wynosi 5 ms.
- 95% żądań jest obsługiwanych w czasie krótszym niż 100 ms.

- Eliminacja opóźnienia związanego z elementami odstającymi: 0 żądań, których wykonanie zajmuje ponad 1000 ms.
- Maksymalna przepustowość wynosi 10 000 żądań aplikacji kierowanych co sekundę do serwera.
- Średni poziom wykorzystania dysku wynosi poniżej 50% dla 10 000 żądań wykonywanych przez aplikację co sekundę.

Po wybraniu odpowiednich celów można przystąpić do pracy polegającej na usunięciu wszystkich czynników uniemożliwiających osiągnięcie wyznaczonych celów. W przypadku opóźnienia czynnikiem ograniczającym mogą być dyskowe lub sieciowe operacje wejścia-wyjścia, natomiast dla celu związanego z przepustowością przeszkodą może okazać się poziom użycia procesora. Strategie omówione w tym i kolejnych rozdziałach pomogą w identyfikacji czynników ograniczających.

W przypadku celów związanych z przepustowością warto zwrócić uwagę, że nie wszystkie operacje są równe pod względem wydajności lub kosztu. Jeżeli celem jest osiągnięcie określonej częstotliwości wykonywania operacji, bardzo ważne może być określenie rodzaju operacji. Istnieje prawdopodobieństwo wystąpienia rozbieżności na podstawie obciążenia oczekiwanego i zmierzonego.

W podrozdziale 5.2 „Techniki sprawdzania wydajności aplikacji" będą omówione najczęściej stosowane metody poprawy wydajności działania aplikacji. Niektóre z nich mogą mieć sens dla jednej grupy celów, ale już nie dla innej. Na przykład zwiększenie operacji wejścia-wyjścia może poprawić przepustowość, ale kosztem większego opóźnienia. Podczas dostrajania wydajności aplikacji pamiętaj o zdefiniowanych celach i stosuj odpowiednie rozwiązania pozwalające na ich osiągnięcie.

5.1.2. Optymalizacja najczęstszego sposobu użycia aplikacji

Wewnętrzne komponenty aplikacji mogą być naprawdę skomplikowane i zawierać wiele różnych, możliwych ścieżek kodu oraz typów zachowania. To może być szczególnie widoczne w trakcie analizy kodu źródłowego: aplikacje nierzadko składają się z dziesiątek tysięcy wierszy kodu, natomiast w przypadku jądra systemu operacyjnego będą to już setki tysięcy. Losowe wybranie obszaru optymalizacji może być przyczyną ogromnej pracy, której skutkiem będzie jedynie niewielki wzrost wydajności działania aplikacji.

Jednym ze sposobów efektywnego poprawienia wydajności aplikacji jest wyszukanie najczęściej stosowanej ścieżki kodu dla danego obciążenia produkcyjnego i przystąpienie do jej usprawnienia. Jeżeli czynnikiem ograniczającym aplikację jest procesor, to może oznaczać, że ścieżki kodu dużą ilość czasu przebywają w procesorze. Przy ograniczeniu związanym z operacjami wejścia-wyjścia warto wyszukać ścieżki kodu, które najczęściej prowadzą do powstania operacji wejścia-wyjścia. Wspomniane ścieżki można znaleźć dzięki analizie i profilowaniu aplikacji, między innymi śledzeniu stosu, co będzie omówione w dalszych rozdziałach. Jeszcze lepszą możliwość poznania najczęstszego sposobu użycia aplikacji dają oferowane przez nią narzędzia monitorowania.

5.1.3. Monitorowanie

Jak zostało to omówione w wielu rozdziałach niniejszej książki, największy przyrost wydajności w systemie operacyjnym można uzyskać przez *eliminację niepotrzebnie wykonywanej pracy*. Ta sama zasada dotyczy aplikacji.

Ten fakt bardzo często jest przeoczany podczas wyboru aplikacji na podstawie jej wydajności. Jeżeli przeprowadzone testy wydajności pokazały, że aplikacja A jest o 10% szybsza od B, to kuszące może być wybranie właśnie aplikacji A. Jednak jeśli aplikacja B w przeciwieństwie do A dostarcza bogatego zestawu narzędzi monitorowania, to istnieje prawdopodobieństwo, że w dłuższej perspektywie czasu lepszym wyborem będzie aplikacja B. Dzięki wspomnianym narzędziom monitorowania będzie można wyszukać i wyeliminować niepotrzebnie wykonywaną pracę, a ponadto lepiej poznać i dostosować wykonywane operacje. Osiągnięta wtedy poprawa wydajności może przekraczać początkową różnicę, wynoszącą 10%.

5.1.4. Notacja „duże O”

Powszechnie traktowana jako dziedzina naukowa, notacja „duże O” jest wykorzystywana do analizy poziomu skomplikowania algorytmów oraz modelowania sposobu ich działania, gdy nastąpi wzrost zbioru danych wejściowych. Podczas tworzenia aplikacji, dzięki wspomnianej analizie, programiści zyskują więc możliwość wyboru efektywnych algorytmów (patrz odwołania [Knuth 76] i [Knuth 97] na końcu rozdziału).

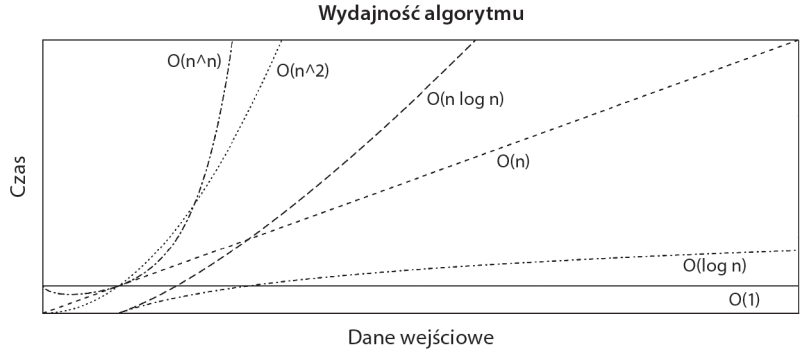
Najczęściej stosowane algorytmy i notacje „duże O” zostały wymienione w tabeli 5.1.

Tabela 5.1. Przykładowe notacje „duże O”

Notacja	Przykład
O(1)	Test boolowski.
O(log n)	Przeszukiwanie binarne posortowanej tablicy.
O(n)	Przeszukiwanie liniowe listy połączonej.
O(n log n)	Szybkie sortowanie (przypadek średni).
O(n^2)	Sortowanie bąbelkowe (przypadek średni).
O(2^n)	Dzielenie liczb na czynniki pierwsze, wzrost wykładniczy.
O(n!)	Problem komiwojażera.

Omawiana notacja pozwala programistom oszacować poprawę szybkości działania różnych algorytmów i ustalić, które obszary kodu prowadzą do największych przyrostów w wydajności działania. Na przykład w przypadku posortowanej tablicy 100 elementów różnica między operacjami wyszukiwania liniowego i binarnego ma współczynnik wynoszący 21 (100/log(100)).

Wydajność algorytmów wymienionych w tabeli 5.1 pokazano na rysunku 5.1, na którym widać także ich trend podczas skalowania.



Rysunek 5.1. Czas działania kontra wielkość danych wejściowych w przypadku różnych algorytmów

Tego rodzaju klasyfikacja pomaga analitykowi wydajności systemu przekonać się, że pewne algorytmy będą bardzo kiepsko działały podczas skalowania. Problemy związane z wydajnością mogą się pojawić, gdy aplikacja będzie musiała obsłużyć większą liczbę użytkowników lub obiektów danych niż wcześniej. Na tym etapie algorytm taki jak O(n^2) może okazać się patologiczny. Rozwiązania dostępne dla programistów to między innymi zastosowanie znacznie efektywniejszego algorytmu lub inne partycjonowanie danych.

Notacja „duże O" ignoruje pewne stałe koszty związane z wyborem poszczególnych algorytmów. W przypadkach, gdzie *n* (wielkość danych wejściowych) jest małe, wspomniane koszty mogą być dominujące.

5.2. Techniki sprawdzania wydajności aplikacji

W tym podrozdziale zostaną przedstawione niektóre najczęściej stosowane techniki umożliwiające poprawę wydajności działania aplikacji: wybór wielkości operacji wejścia-wyjścia, stosowanie pamięci podręcznej, buforowania, techniki odpytania (ang. *polling*), współbieżności i jednoczesnego wykonywania operacji, a także nieblokujących operacji wejścia-wyjścia i powiązania z procesorem. W dokumentacji aplikacji znajdziesz informacje, które z wymienionych technik mogą być stosowane, a także wszelkie funkcje dodatkowe, charakterystyczne dla danej aplikacji.

5.2.1. Ustalenie wielkości operacji wejścia-wyjścia

Koszt związany z przeprowadzaniem operacji wejścia-wyjścia obejmuje inicjalizację buforów, wykonanie wywołania systemowego, przełączenie kontekstu, alokację metadanych jądra, sprawdzenie ograniczeń i uprawnień procesu, mapowanie adresów na urządzenia, wykonanie kodu jądra i sterownika w celu przeprowadzenia operacji wejścia-wyjścia i wreszcie zwolnienie metadanych oraz buforów. Wspomniany koszt inicjalizacji dotyczy zarówno małych, jak i dużych operacji wejścia-wyjścia. Z perspektywy wydajności — im więcej danych jest przekazywanych w trakcie każdej operacji wejścia-wyjścia, tym lepiej.

Wzrost wielkości operacji wejścia-wyjścia to najczęściej stosowana w aplikacjach strategia poprawy przepustowości. Gdy weźmie się pod uwagę stały koszt operacji wejścia-wyjścia, zwykle znacznie efektywniejsze będzie przekazanie 128 KB danych w postaci pojedynczej operacji wejścia-wyjścia niż 128 operacji o wielkości 1 KB. Dyskowe operacje wejścia-wyjścia charakteryzują się szczególnie wysokim kosztem pojedynczej operacji, co wiąże się z czasem wyszukiwania danych.

Istnieje sytuacja, w której aplikacja nie potrzebuje większych operacji wejścia-wyjścia. Baza danych wykonująca losowy odczyt 8 KB danych będzie działała wolniej w przypadku operacji wejścia-wyjścia o wielkości 128 KB, ponieważ transfer 120 KB będzie zmarnowany. Tutaj pojawia się kwestia opóźnienia wejścia-wyjścia, które można ograniczyć przez zastosowanie mniejszej wielkości operacji wejścia-wyjścia, bardziej odpowiadającej żądaniu wykonywanemu przez aplikację. Niepotrzebnie duże operacje wejścia-wyjścia marnują także przestrzeń bufora.

5.2.2. Pamięć podręczna

System operacyjny stosuje różne rodzaje pamięci podręcznej w celu poprawy wydajności odczytu danych z systemu plików oraz alokacji pamięci. Aplikacje bardzo często używają pamięci podręcznych z podobnych powodów. Zamiast za każdym razem wykonywać kosztowną operację, wyniki najczęściej przeprowadzanych operacji mogą być buforowane lokalnie w celu późniejszego użycia. Bardzo dobrym przykładem będzie tutaj bufor bazy danych, który przechowuje wyniki najczęściej wykonywanych zapytań.

Podczas wdrażania aplikacji często wykonywanym zadaniem jest określenie dostępnych lub włączonych pamięci podręcznych, a następnie konfiguracja ich wielkości w sposób najbardziej dopasowany do systemu.

Bardzo ważnym aspektem pamięci podręcznej jest sposób zachowania spójności, tak aby operacje wyszukiwania nie zwracały nieaktualnych danych. Nosi to nazwę **koherencji pamięci podręcznej** i może być operacją kosztowną do wykonania — najlepiej, żeby koszt nie przekraczał korzyści wynikających z użycia pamięci podręcznej.

Pamięć podręczna poprawia wydajność odczytu, natomiast do poprawienia wydajności operacji zapisu bardzo często stosowane są bufory.

5.2.3. Buforowanie

W celu poprawienia wydajności zapisu dane mogą być umieszczane w buforze przed ich przekazaniem na następny poziom. Takie rozwiązanie powoduje zwiększenie wielkości operacji wejścia-wyjścia, a tym samym jej efektywności. W zależności od typu operacji zapisu może się to wiązać również ze wzrostem opóźnienia, ponieważ pierwsze dane umieszczone w buforze czekają na kolejne i dopiero później będą przekazane dalej.

Bufor cykliczny jest rodzajem stałego bufora, który można wykorzystywać do nieustannych transferów między komponentami. Działa więc jak bufor asynchroniczny. Tego rodzaju bufor jest implementowany za pomocą wskaźników początkowego i końcowego; są one przesuwane o każdy komponent, gdy dane są dodawane lub usuwane.

5.2.4. Technika odpytywania

Odpytywanie to technika, w której system czeka na wystąpienie zdarzenia, nieustannie sprawdzając stan zdarzenia w pętli, i stosuje pauzy między kolejnymi sprawdzeniami. Wraz z techniką odpytywania pojawiają się pewne potencjalne problemy dotyczące wydajności:

- kosztowne obciążenie procesora wynikające z regularnie powtarzających się operacji sprawdzeń,
- duże opóźnienie między wystąpieniami zdarzenia i kolejnymi operacjami sprawdzenia.

Jeżeli to rzeczywiście będzie problem wydajności, w aplikacjach można zmienić zachowanie na nasłuchiwanie występujących zdarzeń. W takiej sytuacji aplikacja jest informowana natychmiast o zdarzeniu i wywołuje żądaną procedurę.

Wywołanie systemowe poll()

Istnieje wywołanie systemowe `poll()` przeznaczone do sprawdzania stanu deskryptorów plików, co pełni funkcję podobną do techniki odpytywania. Ponieważ rozwiązanie jest oparte na zdarzeniach, nie dotyczy go koszt wydajności występujący w przypadku odpytywania.

Interfejs wywołania systemowego `poll()`obsługuje wiele deskryptorów plików umieszczonych w tablicy, co wymaga od aplikacji przeprowadzenia operacji skanowania tablicy po wystąpieniu zdarzenia, aby znaleźć powiązane z nim deskryptory plików. Takie skanowanie można uznać za notację „duże O" typu O(n) — patrz punkt 5.1.4. Obciążenie związane ze wspomnianą operacją skanowania może stać się problemem wydajności podczas skalowania. Dostępne są jeszcze inne interfejsy. Linux oferuje wywołanie `epoll()`, które pozwala na uniknięcie skanowania, a tym samym można je określić jako O(1). Z kolei Solaris posiada podobną funkcję o nazwie *porty zdarzeń*, która używa `port_get(3C)` zamiast `poll()`.

5.2.5. Współbieżność i równoległość

Systemy dzielenia czasu (łącznie ze wszystkimi pochodnymi systemu UNIX) zapewniają **współbieżność** programów, czyli możliwość jednoczesnego wczytywania i działania wielu programów. Wprawdzie ich czasy działania mogą wzajemnie na siebie nachodzić, ale programy niekoniecznie będą natychmiast wykonywane przez procesor. Każdy ze wspomnianych programów może być procesem aplikacji.

Poza współbieżnym wykonywaniem różnych aplikacji poszczególne funkcje w ramach aplikacji również mogą być współbieżne. Stało się to możliwe dzięki użyciu wielu procesów (**wieloprocesowość**) lub wątków (**wielowątkowość**), z których każdy wykonuje swoje zadanie.

Inne podejście to **współbieżność oparta na zdarzeniach**, w którym aplikacja obsługuje różne funkcje i przełącza się między nimi, gdy występują odpowiednie zdarzenia. Tego rodzaju podejście jest używane na przykład w środowisku uruchomieniowym

Node.js. W ten sposób zapewniona zostaje współbieżność, ale odbywa się to w ramach pojedynczego wątku lub procesu, co na pewno przyczynia się do powstania wąskiego gardła wydajności, ponieważ może być wykorzystany tylko jeden procesor.

Aby wykorzystać możliwości oferowane przez system wieloprocesorowy, aplikacja musi używać jednocześnie wielu procesorów — nosi to nazwę **równoległości**. W aplikacji można osiągnąć równoległość za pomocą wieloprocesowości lub wielowątkowości. Z powodów przedstawionych w rozdziale 6. „Procesory" wiele wątków (lub odpowiadających im zadań) oferuje znacznie większą efektywność i dlatego preferowane jest zastosowanie właśnie takiego rozwiązania.

Oprócz większej przepustowości pracy procesora wiele wątków (lub procesów) pozwala na równoczesne przeprowadzanie operacji wejścia-wyjścia, ponieważ inne wątki mogą działać wtedy, gdy zablokowany wątek oczekuje na operację wejścia-wyjścia.

Z tego powodu, że w programowaniu wielowątkowym współdzielona jest ta sama przestrzeń adresowa, jaką ma proces, wątki zyskują możliwość bezpośredniego odczytu i zapisu tej samej pamięci, bez konieczności korzystania z kosztowych interfejsów, takich jak komunikacja międzyprocesowa (ang. *Inter-Process Communication*, IPC), stosowana w programowaniu wieloprocesowym. W celu zachowania spójności używana jest synchronizacja początkowa, aby dane nie zostały uszkodzone na skutek ich jednoczesnego odczytu i zapisu. Takie rozwiązanie można zastosować w połączeniu z tabelami hash, poprawiając tym samym wydajność.

Synchronizacja początkowa

Synchronizacja początkowa pilnuje dostępu do pamięci, podobnie jak sygnalizacja świetlna reguluje ruch na skrzyżowaniu. Niczym wspomniana sygnalizacja synchronizacja początkowa wstrzymuje pewien ruch i tym samym zmusza dane do oczekiwania (opóźnienie). Istnieją trzy rodzaje najczęściej stosowanej synchronizacji początkowej:

- **Blokady muteksu**. Tylko nakładający blokadę ma prawo do działania, pozostali są zablokowani i muszą czekać na swoją kolej.
- **Wirujące blokady** (ang. *spinlocks*). Wirująca blokada pozwala nakładającemu na działanie, podczas gdy pozostali czekają na jej zwolnienie, nieustannie w pętli sprawdzając stan blokady. Wprawdzie takie rozwiązanie może zapewnić dostęp charakteryzujący się małym opóźnieniem — zablokowany wątek nigdy nie opuszcza procesora i jest gotowy do działania w ciągu dosłownie kilku cykli, ale oznacza to marnowanie zasobów procesora podczas oczekiwania na zwolnienie blokady i sprawdzanie jej stanu.
- **Blokady odczytu i zapisu**. Blokady odczytu i zapisu zapewniają spójność danych, ponieważ zezwalają na wiele jednoczesnych operacji odczytu lub tylko jedną operację zapisu i żadną odczytu.

Blokady muteksu są implementowane przez bibliotekę lub jądro jako **adaptacyjne blokady muteksu**: hybryda metod muteksu i wirujących. Blokada wirująca występuje wtedy, gdy nakładający blokadę działa aktualnie w innym procesorze, natomiast zwykła — w przeciwnym razie (lub po osiągnięciu maksymalnej liczby dozwolonych blokad wirujących). Adaptacyjne blokady muteksu są zoptymalizowane w celu zapewnienia

dostępu charakteryzującego się małym opóźnieniem bez marnowania zasobów procesora. Dlatego też od wielu lat są stosowane w systemach Solaris. W 2009 roku zostały zaimplementowane również w systemie Linux, gdzie nazwano je **adaptacyjnymi wirującymi blokadami muteksu** (patrz odwołanie [1] na końcu rozdziału).

Analiza problemów wydajności obejmuje także sprawdzanie blokad, co niewątpliwie jest bardzo czasochłonnym zajęciem i wymaga znajomości kodu źródłowego aplikacji. To zwykle jest zajęcie dla programisty.

Tabele hash

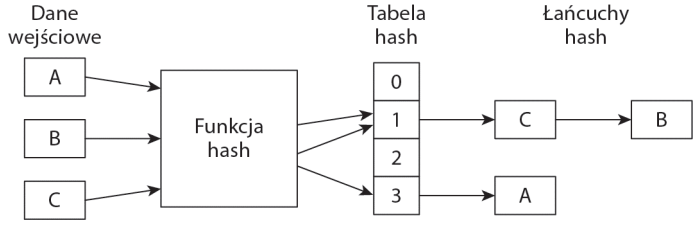
Tabele hash blokad mogą być używane w celu zastosowania minimalnej liczby blokad dla jak największej liczby struktur danych. W tym miejscu przedstawiono jedynie podsumowanie tabel hash — to jest bardzo zaawansowany temat, którego dokładne poznanie wymaga wiedzy z zakresu programowania.

Zapoznaj się z dwoma następującymi podejściami:

- **Pojedyncza globalna blokada muteksu dla wszystkich struktur danych**. Wprawdzie takie rozwiązanie jest proste, ale współbieżny dostęp będzie charakteryzował się wówczas występowaniem rywalizacji i opóźnieniem związanym z oczekiwaniem na uzyskanie dostępu. Wiele wątków wymagających nałożenia blokady będzie **serializowanych**, czyli wykonywanych po kolei zamiast jednocześnie.
- **Oddzielna blokada muteksu dla poszczególnych struktur danych**. Co prawda takie rozwiązanie ogranicza stan rywalizacji do absolutnego minimum — w trakcie jednoczesnego dostępu do tej samej struktury danych, ale blokada wiąże się z obciążeniem powodowanym przez pamięć masową i procesor podczas tworzenia i znoszenia blokady dla każdej struktury danych.

Tabela hash blokad może być rozwiązaniem przejściowym i jest odpowiednia, gdy spodziewasz się dużej rywalizacji o dostęp do zasobów. Rozwiązanie polega na utworzeniu stałej liczby blokad i użyciu algorytmu hash do wyboru blokady, która ma być stosowana z określoną strukturą danych. W ten sposób unika się kosztu związanego z tworzeniem i znoszeniem blokady dla struktury danych, a ponadto eliminowane są problemy występujące w przypadku istnienia tylko jednej blokady.

Przykład tabeli hash pokazano na rysunku 5.2 — składa się ona z czterech elementów (nazywanych *buckets*), z których każdy zawiera własną blokadę.



Rysunek 5.2. Przykładowa tabela hash

Na rysunku pokazano jedno z podejść w zakresie rozwiązania **kolizji hash**, kiedy to dwie struktury danych wejściowych lub ich większa liczba są przydzielone do tego samego elementu. W pokazanym przykładzie następuje utworzenie łańcucha struktur danych w celu przechowywania ich wszystkich w tym samym elemencie, do którego zostały przypisane przez funkcję hash. Jeżeli struktury danych będą zbyt długie i przetwarzane szeregowo, wspomniane łańcuchy mogą przysporzyć problemów związanych z wydajnością. Funkcję hash i wielkość tabeli można ustalić tak, aby równomiernie rozłożyć struktury danych w poszczególnych elementach i tym samym do minimum ograniczyć długość łańcuchów.

W idealnej sytuacji liczba elementów tabeli hash powinna być równa liczbie procesorów lub większa od niej, aby tym samym zapewnić maksymalną współbieżność. Z kolei algorytm funkcji hash może być prosty, na przykład pobierać najmniej znaczący bit adresu struktury danych i użyć go jako indeksu tablicy blokad, mającej wielkość równą potędze liczby 2. Tego rodzaju prosty algorytm charakteryzuje się dużą szybkością działania, a więc pozwala na szybkie przydzielanie lokalizacji strukturom danych.

W przypadku tabel sąsiadujących w pamięci blokad problem związany z wydajnością może się pojawić, gdy blokady znajdą się w tym samym bloku danych pamięci podręcznej. Dwa procesory uaktualniające różne blokady w tym samym bloku danych napotkają obciążenie związane z koherencją, ponieważ poszczególne procesory będą unieważniać blok danych tego drugiego procesora. Sytuacja taka nosi nazwę **błędnego współdzielenia** (ang. *false sharing*) i jest najczęściej rozwiązywana przez dopełnienie blokad nieużywanymi bajtami, aby w określonym bloku danych w pamięci znajdowała się tylko jedna blokada.

5.2.6. Nieblokujące operacje wejścia-wyjścia

Cykl życiowy procesu w systemie UNIX, pokazany na rysunku 3.7 w rozdziale 3. „Systemy operacyjne", wskazywał na blokowanie procesu i przejście do stanu uśpienia w czasie wykonywania operacji wejścia-wyjścia. Z tym modelem wiąże się kilka problemów dotyczących wydajności:

- W przypadku wielu jednoczesnych operacji wejścia-wyjścia poszczególne operacje w trakcie blokady zabierają wątek (lub proces). W celu zapewnienia obsługi wielu jednoczesnych operacji wejścia-wyjścia aplikacja musi utworzyć wiele wątków (zwykle po jednym dla każdego klienta), co oznacza koszt związany z tworzeniem i niszczeniem wątków.
- W przypadku krótkotrwałych operacji wejścia-wyjścia obciążenie związane z częstym przełączaniem kontekstu może zużywać zasoby procesora i przyczyniać się do zwiększania opóźnienia aplikacji.

Model **nieblokujących operacji wejścia-wyjścia** powoduje asynchroniczne wykonywania tych operacji, bez blokowania bieżącego wątku, który w ten sposób można wykorzystać do innych zadań. To jest funkcja kluczowa Node.js (patrz odwołanie [2] na końcu rozdziału), czyli działającego po stronie serwera środowiska aplikacji JavaScript, które w nieblokujący sposób kieruje kodem do wykonania.

5.2.7. Powiązanie z procesorem

W środowiskach NUMA (ang. *Non-Uniform Memory Access*) korzystna może być sytuacja, gdy proces lub wątek będzie działał w pojedynczym procesorze oraz w tym samym procesorze, co poprzednio po wykonaniu operacji wejścia-wyjścia. W ten sposób poprawia się lokalizację pamięci dla aplikacji, zmniejsza liczbę cykli potrzebnych do wykonania operacji wejścia-wyjścia dla pamięci oraz ogólnie poprawia wydajność działania aplikacji. Oczywiście twórcy systemów operacyjnych doskonale o tym wiedzą i tworzą je w taki sposób, aby wątki aplikacji działały w tym samym procesorze (tzw. koligacja procesora). Te tematy będą poruszane w rozdziale 7., zatytułowanym „Pamięć".

Pewne aplikacje wymuszą zastosowanie przedstawionego zachowania przez *powiązanie się* z procesorem. W niektórych systemach może to spowodować znaczną poprawą wydajności działania. Z drugiej strony, jeśli powiązanie będzie tworzyło konflikt z innym powiązaniem procesora, na przykład urządzenie będzie zakłócało mapowanie na procesor, wówczas wspomniane powiązanie spowoduje spadek wydajności.

O ryzyku dotyczącym powiązania z procesorem należy pamiętać szczególnie wtedy, gdy w systemie działają inne tenanty lub aplikacje. Ten problem można napotkać podczas przetwarzania w chmurze dla wirtualizacji systemu operacyjnego, kiedy aplikacja może zobaczyć wszystkie procesory i przyłączyć się do wybranego, jakby była jedyną aplikacją w serwerze. Jeżeli serwer jest współdzielony przez aplikacje różnych tenantów, które także stosują powiązanie z procesorem, to mamy do czynienia z konfliktami i opóźnieniami algorytmu szeregowania. Wynika to z faktu, że procesor jest zajęty obsługą zadań innych tenantów, nawet pomimo bezczynności pozostałych procesorów.

5.3. Języki programowania

Języki programowania mogą być kompilowane lub interpretowane, a na dodatek wykonywane za pomocą maszyny wirtualnej. Wiele języków wymienia „optymalizacje w zakresie wydajności" jako jedną z funkcji, ale szczerze mówiąc, to najczęściej są funkcje oprogramowania *wywołującego* dany język, a nie samego języka. Na przykład oprogramowanie Java HotSpot Virtual Machine zawiera kompilator JIT (ang. *Just-In-Time*) w celu dynamicznej poprawy wydajności.

Interpretery i maszyny wirtualne języków również zapewniają różne poziomy obsługi w zakresie monitorowania za pomocą własnych, specyficznych narzędzi. Analitykowi wydajności systemu przeprowadzenie prostego profilowania przy użyciu wspomnianych narzędzi może przynieść pewne korzyści. Na przykład wysoki poziom zużycia procesora można zidentyfikować jako wynik działania mechanizmu usuwania nieużytków (ang. *garbage collection*), a następnie usunąć problem za pomocą doskonale znanych technik. Obciążenie może być również skutkiem działania ścieżki kodu w wyniku znanego błędu w bazie danych, którego usunięcie wymaga po prostu uaktualnienia oprogramowania (taka sytuacja zdarza się dość często).

W kolejnych punktach przygotowano omówienie pewnej podstawowej charakterystyki wydajności w poszczególnych rodzajach języków programowania. Więcej informacji na temat wydajności tych języków programowania znajdziesz w poświęconych im książkach.

5.3.1. Języki kompilowane

Kompilacja to proces polegający na zamianie kodu źródłowego na instrukcje kodu maszynowego, umieszczane w plikach wykonywanych nazywanych **binarnymi**. Odbywa się to jeszcze przed uruchomieniem programu. Tak powstałe pliki binarne można uruchamiać później dowolnie, bez konieczności ponownej kompilacji. Do języków kompilowanych zaliczamy między innymi C i C++. Niektóre języki mogą być zarówno interpretowane, jak i kompilowane.

Ogólnie rzecz biorąc, kod skompilowany charakteryzuje się większą wydajnością działania i nie wymaga dalszej analizy przed jego wykonaniem przez procesor. Jądro systemu operacyjnego zostało utworzone prawie całkowicie w języku C, poza kilkoma komponentami o znaczeniu krytycznym, które przygotowano w asemblerze.

Analiza wydajności języków kompilowanych jest zwykle prosta, ponieważ wykonywany kod maszynowy najczęściej jest mapowany w pobliżu oryginalnego programu (to oczywiście zależy od optymalizacji zastosowanych w trakcie kompilacji). Podczas kompilacji może zostać wygenerowana tabela symboli przeznaczona do mapowania adresów na funkcje programu i nazwy obiektów. Przeprowadzane później profilowanie i monitorowanie działania procesora może być mapowane bezpośrednio na wspomniane nazwy w programie, co pozwala analitykowi analizować wykonywanie programu. Stos i zawarte w nim adresy liczbowe również mogą być mapowane i przekształcane na nazwy funkcji, aby zapewnić hierarchię ścieżki kodu.

Kompilator ma możliwość poprawienia wydajności na skutek zastosowania tak zwanej **optymalizacji kodu wynikowego**, czyli procedur optymalizujących wybór, i umieszczenie instrukcji procesora.

Optymalizacja kodu wynikowego

Kompilator `gcc` pozwala na wybór poziomu optymalizacji od 0 do 3, przy czym 3 oznacza największą liczbę optymalizacji. Istnieje możliwość sprawdzenia w `gcc`, jakie optymalizacje są stosowane na poszczególnych poziomach, na przykład:

```
$ gcc -Q -O3 --help=optimizers
The following options control optimizations:
  -O
  -Ofast
  -Os
  -falign-functions                     [enabled]
  -falign-jumps                         [enabled]
  -falign-labels                        [enabled]
  -falign-loops                         [enabled]
  -fasynchronous-unwind-tables          [enabled]
  -fbranch-count-reg                    [enabled]
  -fbranch-probabilities                [disabled]
  -fbranch-target-load-optimize         [disabled]
[...]
  -fomit-frame-pointer                  [disabled]
[...]
```

Pełna lista zawiera około 180 opcji, część z nich jest włączona nawet na poziomie 0 optymalizacji (`-O0`). Zobaczmy, jak wygląda działanie jednej z tych opcji, `-fomit-frame-pointer`; poniższy akapit pochodzi ze strony podręcznika `man` kompilatora `gcc`:

> Nie przechowuj ramki wskaźnika w rejestrze dla funkcji, które tego nie potrzebują. Dzięki temu można uniknąć konieczności zachowania instrukcji, konfiguracji i przywracania ramek wskaźników — ponadto dodatkowy rejestr pozostaje dostępny dla wielu funkcji. **W niektórych systemach to uniemożliwia debugowanie**.

Jest to przykład kompromisu: pominięcie ramki wskaźnika zwykle uniemożliwia przeprowadzenie operacji analizatora, który profiluje stos.

Biorąc pod uwagę użyteczność profilowania stosu, użycie omawianej opcji może oznaczać zbyć duże poświęcenie w kategoriach późniejszej wydajności, której nie będzie można łatwo poprawić. To zdecydowanie przewyższa zysk w zakresie wydajności, jaki początkowo można uzyskać dzięki użyciu tej opcji. W omawianym przypadku rozwiązaniem może być kompilacja wraz z opcją `-fno-omit-frame-pointer` w celu uniknięcia wymienionej wcześniej optymalizacji kodu.

Niebezpieczeństwo pojawienia się ewentualnych problemów dotyczących wydajności może skłaniać do przeprowadzenia ponownej kompilacji aplikacji i ustawienia mniejszego poziomu optymalizacji, na przykład wskutek użycia opcji `-O2` zamiast `-O3`, w nadziei, że zaspokoi to wszystkie wymagania w zakresie debugowania. Takie rozwiązanie jednak wcale nie jest proste: zmiany w danych wyjściowych kompilatora mogą być ogromne i ważne, a ponadto mogą mieć wpływ na zachowanie, które początkowo próbujesz poddać analizie.

5.3.2. Języki interpretowane

Języki interpretowane wykonują program na skutek jego zamiany na odpowiednie akcje w trakcie uruchomienia. Ten proces powoduje powstanie dodatkowego obciążenia podczas wykonywania programu. Od języków interpretowanych nie oczekuje się wysokiej wydajności. Są one używane w sytuacjach, gdy inne czynniki mają dużo większe znaczenie, na przykład łatwość programowania i debugowania. Przykładem użycia języka interpretowanego są **skrypty powłoki**.

Jeżeli nie zostaną dostarczone odpowiednie narzędzia monitorowania, analiza wydajności języków interpretowanych może być trudna. Profilowanie procesora pokaże działanie interpretera, między innymi przetwarzanie, translację i wykonywanie działań, ale na pewno nie pokaże oryginalnych nazw funkcji programu, czyli ważny kontekst programu pozostanie tajemnicą. Jednak analiza interpretera na pewno nie jest bezowocna, ponieważ problemy z wydajnością mogą dotyczyć samego interpretera, nawet jeśli wykonywany przez niego kod wydaje się doskonale przygotowany.

W zależności od interpretera kontekst programu może być łatwy do pośredniego uchwycenia, na przykład za pomocą monitorowania dynamicznego analizatora składni. Bardzo często wykonywana jest analiza programów w wyniku dodania poleceń wyświetlających informacje pomocnicze i znaczniki czasu. Rygorystyczniejsza analiza wydajności jest rzadziej spotykana, ponieważ języki interpretowane są znacznie rzadziej wybierane do utworzenia aplikacji, od których wymaga się dużej wydajności działania.

5.3.3. Maszyny wirtualne

Maszyna wirtualna języka (nazywana również **maszyną wirtualną procesu**) to oprogramowanie emulujące komputer. Kod utworzony w pewnych językach programowania, na przykład Java i Erlang, jest najczęściej uruchamiany w maszynach wirtualnych, które zapewniają im niezależne od platformy środowisko programistyczne. Aplikacja zostaje skompilowana na zestaw instrukcji maszyny wirtualnej (*kod bajtowy*), a następnie jest uruchamiana przez wspomnianą maszynę wirtualną. Takie rozwiązanie pozwala na zachowanie przenośności skompilowanych obiektów, oczywiście przy założeniu, że na platformie docelowej znajduje się maszyna wirtualna, która potrafi uruchamiać te skompilowane obiekty.

Kod bajtowy jest *kompilowany* na podstawie kodu źródłowego programu, a następnie *interpretowany* przez język maszyny wirtualnej, który tłumaczy go na kod maszynowy. Maszyna wirtualna Java HotSpot obsługuje kompilację JIT, co powoduje kompilację kodu bajtowego na kod maszynowy przed czasem, a więc w trakcie jego wykonywania można analizować rodzimy kod maszynowy. W ten sposób można połączyć zalety kodu kompilowanego z przenośnością maszyny wirtualnej.

Maszyny wirtualne to z reguły najtrudniejsze do monitorowania rodzaje języków. Zanim program będzie wykonywany przez procesor, przeprowadzonych musi być wiele etapów kompilacji lub interpretacji kodu, a informacje o oryginalnym programie niekoniecznie będą łatwo dostępne. Analiza wydajności zwykle koncentruje się na zestawie narzędzi dostarczanych wraz z maszyną wirtualną języka. Wiele z nich oferuje sondy DTrace oraz narzędzia opracowane przez firmy trzecie.

5.3.4. Mechanizm usuwania nieużytków

Niektóre języki stosują automatyczne zarządzanie pamięcią, czyli zaalokowana pamięć nie musi być wyraźnie zwalniana. To zadanie jest pozostawione asynchronicznemu procesowi mechanizmu usuwania nieużytków. Wprawdzie takie rozwiązanie ułatwia programiście tworzenie aplikacji, ale jednocześnie wiąże się z pewnymi wadami, do których można zaliczyć:

- **Wzrost zużycia pamięci**. Mniejsza kontrola nad zużyciem pamięci przez aplikację oznacza większy poziom jej użycia, gdy nie wszystkie obiekty będą automatycznie identyfikowane jako możliwe do usunięcia z pamięci. Jeżeli aplikacja stanie się ogromna, to może osiągnąć własne ograniczenia lub doświadczyć stronicowania pamięci, co niezwykle negatywnie odbija się na wydajności.
- **Koszt związany z większym użyciem zasobów procesora**. Mechanizm usuwania nieużytków działa nieregularnie i obejmuje operację wyszukiwania obiektów w pamięci. To oczywiście oznacza zużywanie zasobów procesora, a więc zmniejszanie na krótki czas ilości zasobów dostępnych dla aplikacji. Kiedy wzrośnie ilość pamięci zużywanej przez aplikację, równocześnie może zwiększyć się zapotrzebowanie mechanizmu usuwania nieużytków na zasoby procesora. W pewnych przypadkach i implementacjach może dojść do sytuacji, że wymieniony mechanizm będzie nieustannie zużywać całą moc procesora.

- **Elementy odstające pod względem opóźnienia**. Wykonywanie aplikacji może być wstrzymane podczas działania mechanizmu usuwania nieużytków, co spowoduje chwilowe wydłużenie czasu reakcji aplikacji. Duże znaczenie ma tutaj typ implementowanego mechanizmu: *stop-the-world*, przyrostowy lub współbieżny.

Mechanizm usuwania nieużytków jest bardzo często poddawany modyfikacjom w celu zmniejszenia poziomu zużycia procesora oraz zmniejszenia opóźnienia elementów odstających. Na przykład maszyna wirtualna Javy oferuje możliwość zmiany wielu parametrów mechanizmu usuwania nieużytków, między innymi jego typu, liczby używanych wątków, maksymalnej wielkości sterty oraz współczynnika zwalniania sterty.

Jeżeli dostosowanie parametrów nie przyniesie oczekiwanych wyników, problem może polegać na tworzeniu przez aplikację zbyt dużej ilości nieużytków bądź na istnieniu wycieku pamięci. To są problemy, którymi powinien zająć się programista aplikacji.

5.4. Metodologia i analiza

W tym podrozdziale zostaną omówione metodologie wykorzystywane podczas analizy aplikacji i dostosowywania jej wydajności działania. Narzędzia używane podczas analizy zostaną wprowadzone tutaj lub w kolejnych rozdziałach. Temat podsumowano w tabeli 5.2.

Tabela 5.2. Metodologie analizy wydajności aplikacji

Metodologia	Rodzaj
Analiza stanu wątku	Analiza oparta na monitorowaniu
Profilowanie procesora	Analiza oparta na monitorowaniu
Analiza wywołań systemowych	Analiza oparta na monitorowaniu
Profilowanie operacji wejścia-wyjścia	Analiza oparta na monitorowaniu
Charakterystyka obciążenia	Analiza oparta na monitorowaniu, planowanie pojemności
Metoda USE	Analiza oparta na monitorowaniu
Analiza drążąca	Analiza oparta na monitorowaniu
Analiza blokad	Analiza oparta na monitorowaniu
Statyczne dostosowanie wydajności	Analiza oparta na monitorowaniu, dostrojenie

W rozdziale 2., noszącym tytuł „Metodologia”, możesz zapoznać się z dodatkowymi informacjami o ogólnych metodologiach oraz z omówieniem wybranych z nich. Ponadto w kolejnych rozdziałach znajdziesz przedstawienie analizy zasobów systemowych i wirtualizacji.

Wspomniane metodologie mogą być stosowane pojedynczo lub w połączeniu z innymi. Sugeruję wypróbowanie ich w kolejności przedstawionej w tabeli 5.2.

Oprócz wymienionych istnieją jeszcze inne techniki analizy stosowane w konkretnych aplikacjach i językach programowania, w których te aplikacje zostały utworzone. Techniki takie mogą uwzględniać zachowania logiczne aplikacji, między innymi znane problemy, i pozwalać na osiągnięcie pewnej poprawy wydajności.

5.4.1. Analiza stanu wątku

Celem jest ogólne ustalenie, gdzie wątki aplikacji przebywają większość czasu. To umożliwia praktycznie natychmiastowe rozwiązanie pewnych problemów i skierowanie analizy innych na odpowiednie tory. Operacja odbywa się przez podział czasu poszczególnych wątków aplikacji na pewną liczbę stanów.

Dwa stany

Absolutne minimum to zastosowanie dwóch stanów wątku:

- **W procesorze**. Wykonywanie wątku.
- **Poza procesorem**. Wątek czeka na swoją kolej dostępu do procesora, na zakończenie operacji wejścia-wyjścia, zwolnienie blokady, stronicowanie, wykonanie innego zadania itd.

Jeżeli wątek większość czasu przebywa w procesorze, to profilowanie procesora zwykle pozwoli na szybkie wyjaśnienie tego stanu rzeczy (więcej o tym znajdziesz w dalszej części rozdziału). Taka sytuacja zdarza się w przypadku wielu problemów związanych z wydajnością, nie trzeba więc poświęcać czasu na przeprowadzanie pomiarów innych stanów wątku.

Jeżeli wątek większość czasu przebywa poza procesorem, to można zastosować różne metodologie. Jednak bez dobrego punktu wyjścia zadanie będzie wymagało poświęcenia dużej ilości czasu.

Sześć stanów

W tym miejscu przedstawiono znacznie bardziej rozbudowaną listę, tym razem zawierającą sześć stanów wątku (i inny schemat nazewnictwa) — w ten sposób można otrzymać znacznie lepszy punkt wyjścia, gdy okaże się, że wątek większość czasu przebywa poza procesorem:

- **Wykonywanie**. W procesorze.
- **Działanie**. Oczekuje na swoją kolej dostępu do procesora.
- **Anonimowe stronicowanie**. Działa, ale jest zablokowany w oczekiwaniu na zakończenie anonimowego stronicowania.
- **Uśpienie**. Czeka na zakończenie operacji wejścia-wyjścia, na przykład sieciowej, na zwolnienie blokady lub przeniesienie danych bądź tekstu między pamięcią operacyjną i wirtualną.

- **Blokada**. Czeka na nałożenie blokady synchronizacji (oczekiwanie na inny element).
- **Bezczynność**. Oczekuje na zadanie do wykonania.

Powyższy zestaw został przygotowany w taki sposób, aby był minimalny i jednocześnie użyteczny. Oczywiście do listy możesz dodać kolejne stany. Na przykład stan wykonywania można podzielić jeszcze na wykonywanie po stronie użytkownika i jądra, natomiast stan uśpienia — podzielić według celu. (Osobiście radzę korzystać z przedstawionej wcześniej listy sześcioelementowej).

Poprawa wydajności nastąpi po zmniejszeniu czasu w pierwszych pięciu stanach, co spowoduje zwiększenie czasu bezczynności. Gdy pozostałe czynniki są takie same, oznacza to, że żądania generowane przez aplikację charakteryzują się mniejszym opóźnieniem, a tym samym może ona obsłużyć większe obciążenie.

Po ustaleniu, w którym z wymienionych pięciu stanów wątki przebywają najwięcej czasu, można przystąpić do dalszej analizy.

- **Wykonywanie**. Za pomocą profilowania sprawdź tryb działania wątku (użytkownika lub jądra), a także powód zużycia procesora. Profilowanie pomaga w ustaleniu, które ścieżki kodu są odpowiedzialne za zużycie procesora oraz przez jaki czas, obejmujący także blokady. Zapoznaj się z punktem 5.4.2. „Profilowanie procesora”.
- **Działanie**. Spędzanie czasu w tym stanie oznacza, że aplikacja potrzebuje więcej zasobów procesora. Przeanalizuj obciążenie procesora w całym systemie oraz wszelkie ograniczenia procesora dla danej aplikacji (na przykład nakładane przez mechanizm kontroli zasobów).
- **Anonimowe stronicowanie**. Brak pamięci operacyjnej dla aplikacji może być powodem powstania anonimowego stronicowania i opóźnień. Przeanalizuj zużycie pamięci w całym systemie oraz wszystkie ograniczenia pamięci dla danej aplikacji. Więcej informacji na ten temat znajdziesz w rozdziale 7. „Pamięć”.
- **Uśpienie**. Przeanalizuj zasoby, w których aplikacja jest zablokowana. Zapoznaj się z punktami 5.4.3 „Analiza wywołań systemowych” i 5.4.4 „Profilowanie operacji wejścia-wyjścia”.
- **Blokada**. Zidentyfikuj blokadę, nakładający ją wątek oraz powód, dla którego blokada trwa długo. Wspomnianym powodem może być oczekiwanie przez nakładającego na zwolnienie innej blokady, co oznacza konieczność kolejnej analizy. To jest zaawansowane działanie, zwykle przeprowadzane przez programistę aplikacji, który ma wystarczająco dużą wiedzę o aplikacji i stosowanej w niej hierarchii blokad.

Z powodu sposobu, w jaki aplikacje zwykle oczekują na zadania, bardzo często będziesz się przekonywał, że czas w stanach uśpienia i blokady to w rzeczywistości czas bezczynności. Wątek roboczy aplikacji może oczekiwać na zakończenie działania zmiennej warunkowej (stan blokady) lub sieciowej operacji wejścia-wyjścia (stan uśpienia). Kiedy spotykasz się z długimi czasami uśpienia i blokady, pamiętaj, aby drążyć dalej ten temat i sprawdzić, czy czas ten nie będzie rzeczywiście czasem bezczynności.

Dalsze podsumowanie pokazuje, jak wspomniane stany wątków można zmierzyć w systemach Linux i Solaris. Narzędzia i technologie użyte w trakcie pomiarów będą dokładniej omówione w innych miejscach książki. Sprawdź je zwłaszcza pod kątem nowych narzędzi i ich opcji, które mogą ułatwić wykonanie zadania pomiaru.

Linux

Czas poświęcony na wykonywanie nie jest trudny do ustalenia, ponieważ polecenie `top` podaje go w kolumnie `%CPU` danych wyjściowych. Pomiar czasu w pozostałych stanach będzie wymagał dodatkowych kroków, które przedstawiono poniżej.

Czas działania jest mierzony przez funkcję *schedstats* jądra i udostępniony przez pliki */proc/*/schedstats*. Wywołanie polecenia `perf sched` również może dostarczyć metryk pozwalających na ustalenie czasu poświęconego na działanie i oczekiwanie.

Czas oczekiwania na anonimowe stronicowanie (w systemie Linux to **wymiana**) można zmierzyć za pomocą funkcji **zliczania opóźnienia**, przy założeniu, że została ona włączona. Wymieniona funkcja przekazuje informacje o oddzielnych stanach wymiany i blokady podczas odzyskiwania pamięci (a także informacje dotyczące nacisku wywieranego na pamięć). Nie ma powszechnie używanego narzędzia do udostępniania informacji o wspomnianych stanach. Jednak w dokumentacji jądra znajduje się przykładowy program służący do tego celu: *getdelays.c*, który zademonstrowano w rozdziale 4. „Narzędzia monitorowania”. Inne podejście polega na użyciu narzędzi monitorowania, takich jak DTrace lub SystemTap.

Czas zablokowania w wątku uśpienia można z grubsza oszacować, stosując inne narzędzia, na przykład polecenie `pidstat -d`, i ustalić, czy proces przeprowadza dyskową operację wejścia-wyjścia, a tym samym, czy prawdopodobnie znajduje się w stanie uśpienia. O ile opóźnienia i inne funkcje sprawdzania operacji wejścia-wyjścia zostały włączone, podają czas oczekiwania na zakończenie operacji wejścia-wyjścia, który można sprawdzić za pomocą polecenia `iotop`. Inne powody blokad można sprawdzić, korzystając z narzędzi monitorowania, takich jak DTrace i SystemTap. Sama aplikacja może udostępniać odpowiednie instrumenty, ewentualnie instrumenty takie można dodać i tym samym monitorować czas przeprowadzania operacji wejścia-wyjścia (zarówno dyskowej, jak i sieciowej).

Jeżeli aplikacja znajduje się w stanie uśpienia przez bardzo długi czas (liczony w sekundach), to powód takiego zachowania można spróbować ustalić za pomocą polecenia `pstack`. Wymienione polecenie pobiera pojedynczą migawkę wątku i informacje o użyciu stosu użytkownika, które powinny zawierać uśpiony wątek i powód jego uśpienia. Pamiętaj, działanie polecenia `pstack` może wstrzymać działanie analizowanej aplikacji i dlatego używaj go ostrożnie.

Czas blokady można sprawdzić za pomocą narzędzi monitorowania.

Solaris

W systemach Solaris dane statystyczne dotyczące *zliczania mikrostanów* (wprowadzonego w rozdziale 4. „Narzędzia monitorowania”) bezpośrednio dostarczają informacji o większości stanów wątku. Wspomniane dane można wyświetlić, stosując polecenie `prstat`:

```
$ prstat -mLcp 4937 1
Please wait...
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
  4937 root     7.4 7.9 0.0 0.0  15 0.0  69 0.2 239  31  3K   0 redis-server/1
  4937 root     0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 redis-server/3
  4937 root     0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 redis-server/2
Total: 1 processes, 3 lwps, load averages: 5.28, 5.36, 5.36
[...]
```

Osiem kolumn, począwszy od `USR` do `LAT`, zawiera informacje o wszystkich mikrostanach wątku i dzieli czas wątku na procenty. Suma wartości wymienionych kolumn wynosi 100%. Poniżej przedstawiono mapowanie stanów na odpowiadające im kolumny.

- **wykonywanie** — `USR + SYS`,
- **działanie** — `LAT`,
- **anonimowe stronicowanie** — `DFL`,
- **uśpienie** — `SLP`,
- **blokada** — `LCK`,
- **bezczynność** — również w `SLP + LCK`.

Wprawdzie to nie jest doskonałe odwzorowanie, ale mimo wszystko dostarczone informacje mają ogromną wartość. Czas bezczynności można sprawdzić za pomocą narzędzia DTrace, analizując stos, gdy wątek opuszcza procesor, i tym samym przekonać się, jaki jest powód oczekiwania. Jeżeli wątek pozostaje w stanie uśpienia przez bardzo długi czas (liczony w sekundach), to wypróbuj polecenie `pstack`. Pamiętaj, działanie polecenia `pstack` może wstrzymać działanie analizowanej aplikacji i dlatego używaj go ostrożnie.

Więcej informacji na temat polecenia `prstat` i wyświetlanych przez nie kolumn znajdziesz w rozdziale 6., zatytułowanym „Procesory”.

5.4.2. Profilowanie procesora

Profilowanie procesora zostanie dokładnie omówione w punkcie 6.5.4 w rozdziale 6. „Procesory”. Znajdziesz tam również szczegółowe przykłady użycia narzędzi DTrace i `perf`. Profilowanie to bardzo ważna operacja i dlatego została tutaj podsumowana z perspektywy aplikacji.

Celem profilowania jest ustalenie, dlaczego aplikacja zużywa zasoby procesora. Efektywną techniką jest próbkowanie w procesorze stosu na poziomie użytkownika i łączenie wyników. Ślad stosu pokazuje zastosowaną ścieżkę kodu, co może wskazać powody, dla których aplikacja zużywa zasoby procesora.

Próbkowanie stosu może wygenerować tysiące wierszy danych wyjściowych, nawet podczas tworzenia podsumowania i wyświetlania jedynie unikalnych stosów. Jednym ze sposobów szybkiego poznania profilu jest użycie wykresów, co również zostanie przedstawione w rozdziale 6.

Oprócz stosu próbkować można również aktualnie wykonywaną funkcję. W takich przypadkach wystarczające jest ustalenie, dlaczego aplikacja używa procesora, i wygenerowanie znacznie mniejszej ilości danych wyjściowych, co ułatwia odczyt i poznanie profilu. Prezentowany przykład pochodzi z rozdziału 6. „Procesory" i wykorzystuje narzędzie DTrace:

```
# dtrace -n 'profile-997 /arg1 && execname == "beam.smp"/ {
    @[ufunc(arg1)] = count(); } tick-10s { exit(0); }'
[...]
innostore_drv.so`os_aio_array_get_nth_slot                    80
  beam.smp`process_main                                       127
  libc.so.1`mutex_trylock_adaptive                            140
  innostore_drv.so`os_aio_simulated_handle                    158
  beam.smp`sched_sys_wait                                     202
  libc.so.1`memcpy                                            258
  innostore_drv.so`ut_fold_binary                            1800
  innostore_drv.so`ut_fold_ulint_pair                        4039
```

W pokazanym przykładzie, w trakcie większości operacji próbkowania, w procesorze była wykonywana funkcja `ut_fold_ulint_pair()`.

Użyteczne może być również przeanalizowanie komponentu wywołującego aktualnie wykonywaną funkcję, co można łatwo zrobić za pomocą niektórych programów służących do profilowania (między innymi DTrace). Jeżeli w poprzednim przykładzie okazałoby się, że w procesorze najwięcej czasu przebywa funkcja `malloc()`, taka informacja nie dostarczy nam zbyt wielu danych. Komponent wywołujący funkcję `malloc()` stanie się znacznie ciekawszym elementem do profilowania, a ponadto nie będzie wymagane przechwytywanie stosu.

Przeanalizowanie zużycia procesora przez interpretowany język programowania i maszynę wirtualną może okazać się trudne. Nie istnieje łatwy sposób mapowania wykonywanego oprogramowania na oryginalny program. Konkretne rozwiązanie zależy od środowiska języka: może oferować funkcje debugowania pozwalające na wykonanie wymienionego zadania lub do tego celu mogą być dostępne narzędzia opracowane przez firmy trzecie.

Na przykład narzędzie DTrace używa tak zwanych **programów pomocniczych ustack** do sprawdzania zawartości maszyny wirtualnej i konwersji stosów z powrotem na oryginalny program. Wspomniane programy pomocnicze istnieją dla Javy, Pythona i Node.js.

To jest przykład próbkowania wykonywanego przez procesor kodu Java za pomocą `jstack()` narzędzia DTrace:

```
# dtrace -n 'profile-97 /pid == 1742/ { @[jstack(100)] = count(); }'
dtrace: description 'profile-97 ' matched 1 probe
^C
[...]
      libc.so.1`_so_send+0x7
      libjvm.so`__1cDhpiEsend6Fipcii_i_+0xac
      libjvm.so`JVM_Send+0x31
      libnet.so`Java_java_net_SocketOutputStream_socketWrite0+0x100
```

```
java/net/SocketOutputStream.socketWrite0
java/net/SocketOutputStream.socketWrite
java/net/SocketOutputStream.write
java/io/DataOutputStream.write
TransThread.TransTCP
TransThread.run
StubRoutines (1)
libjvm.so`__1cJJavaCallsLcall_helper6FpnJJavaValue_pnMmethodHandle_pnRJ...
libjvm.so`__1cCosUos_exception_wrapper6FpFpnJJavaValue_pnMmethodHandle_...
libjvm.so`__1cJJavaCallsEcall6FpnJJavaValue_nMmethodHandle_pnRJavaCallA...
libjvm.so`__1cJJavaCallsMcall_virtual6FpnJJavaValue_nLKlassHandle_nMsym...
libjvm.so`__1cJJavaCallsMcall_virtual6FpnJJavaValue_nGHandle_nLKlassHan...
libjvm.so`__1cMthread_entry6FpnKJavaThread_pnGThread__v_+0xd0
libjvm.so`__1cKJavaThreadRthread_main_inner6M_v_+0x51
libjvm.so`__1cKJavaThreadDrun6M_v_+0x105
libjvm.so`__1cG_start6Fpv_0_+0xd2
libc.so.1`_thr_setup+0x4e
libc.so.1`_lwp_start
 10
```

Dane wyjściowe zostały skrócone, tak że przedstawiają jedynie najczęściej występujące stosy, które były próbkowane dziesięciokrotnie. Stosy pokazują wewnętrzne komponenty wirtualnej maszyny Javy (`libjvm`), a poszczególne funkcje są wyświetlone jako sygnatury C++. Stos Javy został przekształcony z wirtualnej maszyny Javy (pogrubione wiersze danych wyjściowych) i pokazuje klasy oraz metody odpowiedzialne za zużycie procesora w trakcie danej ścieżki kodu. Dla omawianego stosu było to `java/io/DataOutputStream.write`.

W rozdziale 6. „Procesory” przygotowano omówienie innych metodologii i narzędzi, a także różnych sposobów analizowania poziomu zużycia procesora przez aplikację.

5.4.3. Analiza wywołań systemowych

Metodologia analizy stanu wątku rozpoczyna się od opisania dwóch stanów do sprawdzenia: w procesorze i poza procesorem. Będzie użyteczne, a czasami nawet praktyczne, przeanalizowanie wymienionych stanów na podstawie wykonywania wywołań systemowych:

- **wykonywania** — w procesorze (tryb użytkownika),
- **wywołania systemowego** — czas wywołania systemowego (oczekiwanie lub działanie w trybie jądra).

Czas wywołania systemowego obejmuje operacje wejścia-wyjścia, blokady i inne rodzaje wywołań systemowych. W pozostałych stanach wątku, na przykład działania (i oczekiwania na procesor) oraz anonimowego stronicowania, nie ma miejsca na takie uproszczenie. Jeżeli mamy do czynienia z nasyceniem procesora lub pamięci, to takie sytuacje można zidentyfikować w systemie za pomocą metody USE.

Stan wykonywania jest możliwy do przeanalizowania przy użyciu wspomnianej wcześniej metody profilowania procesora.

Z kolei wywołania systemowe można analizować na wiele sposobów. Celem jest ustalenie, gdzie wywołanie systemowe przebywa najwięcej czasu, typu wywołania systemowego i powodu jego wywołania.

Monitorowanie punktu kontrolnego

Tradycyjny styl monitorowania wywołania systemowego oznacza ustawienie punktu kontrolnego dla wywołania systemowego i jego zakończenia. To jest technika inwazyjna, w przypadku aplikacji wykonujących dużą liczbę wywołań systemowych oznacza również wręcz ogromny spadek wydajności.

W zależności od stawianych aplikacji wymagań w zakresie wydajności można zaakceptować stosowanie tego rodzaju stylu monitorowania jedynie w krótkim czasie, aby ustalić typy wywołań systemowych.

strace

W systemie **Linux** można wykorzystać polecenie `strace`, na przykład:

```
$ strace -ttt -T -p 1884
1356982510.395542 close(3)              = 0 <0.000267>
1356982510.396064 close(4)              = 0 <0.000293>
1356982510.396617 ioctl(255, TIOCGPGRP, [1975]) = 0 <0.000019>
1356982510.396980 rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0 <0.000024>
1356982510.397288 rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0 <0.000014>
1356982510.397365 wait4(-1, [{WIFEXITED(s) && WEXITSTATUS(s) == 0}],
WSTOPPED|WCONTINUED, NULL) = 1975 <0.018187>
1356982510.415710 rt_sigprocmask(SIG_BLOCK, [CHLD TSTP TTIN TTOU], [CHLD], 8) = 0
<0.000018>
1356982510.416047 ioctl(255, SNDRV_TIMER_IOCTL_SELECT or TIOCSPGRP, [1884]) = 0
<0.000016>
1356982510.416118 rt_sigprocmask(SIG_SETMASK, [CHLD], NULL, 8) = 0 <0.000154>
[...]
```

W przedstawionym przykładzie użyto następujących opcji (omówienie wszystkich znajdziesz na stronie podręcznika `man`):

- **`-ttt`**. Wyświetla pierwszą kolumnę jako czas od początku epoki. Czas będzie podany w sekundach z dokładnością do mikrosekund.
- **`-T`**. Wyświetla ostatnią kolumnę jako <*czas*>. Kolumna ta wskazuje długość trwania wywołania systemowego. Czas będzie podany w sekundach z dokładnością do mikrosekund.
- **`-p PID`**. Monitorowanie procesu o podanym identyfikatorze. Istnieje również możliwość podania polecenia, co spowoduje jego uruchomienie przez `strace` i monitorowanie.

Funkcje oferowane przez polecenie strace można zobaczyć w danych wyjściowych — przekształcenie argumentów wywołania systemowego ma postać czytelną dla człowieka. To będzie szczególnie użyteczne podczas próby wykrycia użycia ioctl().

Omówiona postać strace wyświetla wiersz danych wyjściowych dla każdego wywołania systemowego. Użycie opcji -c powoduje przygotowanie podsumowania dotyczącego funkcjonowania wywołania systemowego:

```
$ strace -c -p 1884
Process 1884 attached - interrupt to quit
^CProcess 1884 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 83.29    0.007994           9       911       455 wait4
 14.41    0.001383           3       455           clone
  0.85    0.000082           0      2275           ioctl
  0.68    0.000065           0       910           close
  0.63    0.000060           0      4551           rt_sigprocmask
  0.15    0.000014           0       455           setpgid
  0.00    0.000000           0       455           rt_sigreturn
  0.00    0.000000           0       455           pipe
------ ----------- ----------- --------- --------- ----------------
100.00    0.009598                 10467       455 total
```

Przedstawione dane wyjściowe zawierają następujące kolumny:

- **time** — wyrażona w procentach wartość pokazująca, na co został poświęcony czas procesora,
- **seconds** — wyrażony w sekundach całkowity czas procesora,
- **usecs/call** — wyrażony w mikrosekundach średni czas procesora dla wywołania systemowego,
- **calls** — liczba wywołań systemowych w trakcie sesji strace,
- **syscall** — nazwa wywołania systemowego.

Jest to doskonałe rozwiązanie w sytuacji, gdy wiążące się z nim duże obciążenie nie stanowi problemu.

Aby zilustrować problem, polecenie dd wykorzystamy do przeprowadzenia 5 milionów operacji transferu danych o wielkości 1 KB. Sprawdzimy wykonanie zadania bez użycia strace oraz z jego użyciem. W tym pierwszym przypadku polecenie jest wykonywane następująco:

```
$ dd if=/dev/zero of=/dev/null bs=1k count=5000k
5120000+0 records in
5120000+0 records out
5242880000 bytes (5.2 GB) copied, 1.91247 s, 2.7 GB/s
```

Dane wyjściowe polecenia dd podają czas trwania operacji oraz przepustowość. W omawianym przypadku wykonanie zadania zabrało około dwóch sekund.

W tym miejscu przedstawiono wykonanie tego samego zadania, ale z wykorzystaniem `strace` do podsumowania użytych wywołań systemowych:

```
$ strace -c dd if=/dev/zero of=/dev/null bs=1k count=5000k
5120000+0 records in
5120000+0 records out
5242880000 bytes (5.2 GB) copied, 140.722 s, 37.3 MB/s
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 51.46    0.008030           0   5120005           read
 48.54    0.007574           0   5120003           write
  0.00    0.000000           0        20        13 open
  0.00    0.000000           0        10           close
  0.00    0.000000           0         5           fstat
  0.00    0.000000           0         1           lseek
  0.00    0.000000           0        14           mmap
  0.00    0.000000           0         8           mprotect
  0.00    0.000000           0         2           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         6           rt_sigaction
  0.00    0.000000           0         1           rt_sigprocmask
  0.00    0.000000           0         5         5 access
  0.00    0.000000           0         2           dup2
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           getrlimit
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         2         1 futex
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0         1           set_robust_list
------ ----------- ----------- --------- --------- ----------------
100.00    0.015604              10240092        19 total
```

Jak możesz się przekonać, czas trwania operacji wydłużył się 73 razy, podobny spadek zanotowano w przepustowości. Mamy tutaj do czynienia z poważnym problemem, który wynika z tego, że polecenie `dd` wykonuje ogromną ilość wywołań systemowych.

truss

W systemach **Solaris** istnieje polecenie `truss` przeznaczone do monitorowania wywołań systemowych, na przykład:

```
$ truss -dE -p 81573
Base time stamp:  1356985396.2469  [ Mon Dec 31 20:23:16 UTC 2012 ]
 0.0016  0.0000 waitid(P_ALL, 0, 0x08047A80, WEXITED|WTRAPPED|WSTOPPED|WCONTINUED) =
0
 0.0018  0.0000 lwp_sigmask(SIG_SETMASK, 0x06820000, 0x00000000, 0x00000000,
0x00000000) = 0xFFBFFEFF [0xFFFFFFFF]
 0.0019  0.0000 ioctl(255, TIOCGSID, 0x08047AEC)                 = 0
 0.0019  0.0000 getsid(0)                                        = 81573
 0.0020  0.0000 ioctl(255, TIOCSPGRP, 0x08047B24)                = 0
```

```
 0.0021  0.0000 lwp_sigmask(SIG_SETMASK, 0x00020000, 0x00000000, 0x00000000,
0x00000000) = 0xFFBFFEFF [0xFFFFFFFF]
 0.0022  0.0000 ioctl(255, TCGETS, 0x0811D640)                   = 0
 0.0023  0.0000 ioctl(255, TIOCGWINSZ, 0x08047B48)               = 0
[...]
```

W zaprezentowanym przykładzie użyto następujących opcji (omówienie wszystkich znajdziesz na stronie podręcznika `man`):

- **`-d`**. Wyświetla pierwszą kolumnę jako czas od początku epoki.
- **`-E`**. Wyświetla drugą kolumnę jako znacznik czasu pokazujący wyrażony w sekundach czas wykonywania wywołania systemowego.
- **`-p PID`**. Monitorowanie procesu o podanym identyfikatorze. Istnieje również możliwość podania polecenia, co spowoduje jego uruchomienie przez `truss` i monitorowanie.

Dane wyjściowe zawierają po jednym wierszu dla wywołania systemowego i są użyteczne podczas przekształcania argumentów na postać czytelną dla człowieka. Znaczniki czasu mają dokładność jedynie 0,1 ms, co nieco ogranicza ich użyteczność.

Polecenie `truss` obsługuje również tryb podsumowania, dostępny za pomocą opcji `-c`:

```
$ truss -c dd if=/dev/zero of=/dev/null bs=1k count=10k
10240+0 records in
10240+0 records out

syscall               seconds   calls  errors
_exit                    .000       1
read                     .075   10252
write                    .073   10246
open                     .000       9       1
close                    .000      10
brk                      .000       6
getpid                   .000       1
fstat                    .000       6
sysi86                   .000       1
ioctl                    .000       1       1
execve                   .000       1
sigaction                .000       2
getcontext               .000       1
setustack                .000       1
mmap                     .000       8
mmapobj                  .000       1
getrlimit                .000       1
memcntl                  .000       3
sysconfig                .000       3
sysinfo                  .000       1
lwp_private              .000       1
llseek                   .000       3
schedctl                 .000       1
```

```
resolvepath               .000       3
stat64                    .000       2
fstat64                   .000       4
open64                    .000       2
                        --------  ------   ----
sys totals:               .150    20571      2
usr time:                 .029
elapsed:                  .880
```

Kolumna `seconds` pokazuje czas procesora poświęcony na wykonanie wywołania systemowego. Z kolei kolumna `calls` podaje liczbę wywołań.

Dzięki użyciu opcji `-u` polecenie `truss` może również przeprowadzać swego rodzaju *monitorowanie dynamiczne* wywołań funkcji na poziomie użytkownika. Poniżej pokazano na przykład monitorowanie wywołań funkcji `printf()`:

```
$ truss -u 'libc:*printf*' uptime
/1:     open("/usr/lib/locale/en_US.UTF-8/LC_MESSAGES/SUNW_OST_OSCMD.mo", O_RDONLY)
Err#2 ENOENT
/1:     -> libc:printf(0x403363, 0x0, 0x0, 0x0, 0x0, 0xfffffd7fffdfeab0)
/1:     <- libc:printf() = 4
/1:     -> libc:printf(0x403368, 0x58, 0x0, 0x0, 0x0, 0x10)
/1:     <- libc:printf() = 11
[...]
```

Podobnie jak w przypadku polecenia `strace`, obciążenie związane z wykonywaniem wielu wywołań systemowych lub funkcji będzie znaczne, co praktycznie uniemożliwia zastosowanie polecenia `truss` w środowisku produkcyjnym.

Monitorowanie buforowane

W przypadku **monitorowania buforowanego** dane instrumentów mogą być buforowane w jądrze, podczas gdy analizowany program kontynuuje działanie. Takie rozwiązanie różni się od monitorowania punktów kontrolnych, w których następowało przerywanie działania analizowanego programu.

Narzędzie DTrace pozwala na stosowanie monitorowania buforowanego i agregacji, aby tym samym zmniejszyć obciążenie związane z operacją monitorowania i umożliwić programom rejestrację wywołań systemowych do późniejszej analizy. Pewne przykłady takiego rozwiązania zostaną przedstawione w tym fragmencie książki. W systemie Linux, działającym pod kontrolą jądra w wersji 3.7, do polecenia `perf` dodano podpolecenie `trace`, które przeprowadza buforowane monitorowanie wywołań systemowych (i nie tylko).

Przedstawione dalej jednowierszowe wywołania narzędzia DTrace obrazują pewne podstawy dotyczące analizy wywołań systemowych i są przeznaczone zarówno do systemów Linux, jak i Solaris (przykłady dotyczą tego drugiego). Znacznie więcej przykładów jednowierszowych wywołań narzędzia DTrace znajdziesz w dodatku D.

Pokazane tutaj przykłady przetwarzają sygnały (za pomocą wywołania systemowego `kill()`), zawierają identyfikator procesu, jego nazwę oraz docelowy identyfikator procesu i numer sygnału.

```
# dtrace -qn 'syscall::kill:entry {
    printf("%Y: %s (PID %d) sent a SIG %d to PID %d\n",
    walltimestamp, execname, pid, arg1, arg0); }'
2013 Apr 17 00:27:37: bash (PID 2583) sent a SIG 9 to PID 2638
2013 Apr 17 00:27:51: postgres (PID 25906) sent a SIG 16 to PID 25896
2013 Apr 17 00:27:51: postgres (PID 2676) sent a SIG 17 to PID 25906
2013 Apr 17 00:27:51: postgres (PID 2676) sent a SIG 17 to PID 25906
```

Podczas monitorowania przechwycono, jak proces `bash` wysłał sygnał –9 (SIGKILL) do procesu o identyfikatorze 2638, a także pewne sygnały z procesu `postgres` (baza danych PostgreSQL). Uwzględnienie znaczników czasu może być pomocne podczas korelacji z innymi operacjami.

Przedstawione jednowierszowe polecenie powoduje zliczanie wywołań systemowych (za pomocą agregacji) dla procesów o nazwie `postgres` (baza danych PostgreSQL):

```
# dtrace -n 'syscall:::entry /execname == "postgres"/ { @[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 233 probes
^C
  setitimer                                                         4
  semsys                                                           22
  open64                                                           35
  kill                                                             79
  lwp_sigmask                                                      79
  setcontext                                                       79
  write                                                           126
  fcntl                                                           252
  pollsys                                                        2498
  read                                                           2750
  send                                                           9542
  recv                                                          12096
  llseek                                                        27925
```

W trakcie monitorowania najczęściej — 27 925 razy — było wykonywane wywołanie systemowe `llseek()`.

Kolejne polecenie powoduje zmierzenie czasu trwania (nazywanego również **opóźnieniem**) wywołań systemowych `read()`, wykonywanych przez PostgreSQL:

```
# dtrace -n 'syscall::read:entry /execname == "postgres"/ {
    self->ts = timestamp; } syscall::read:return /self->ts/ { @["ns"] =
    quantize(timestamp - self->ts); self->ts = 0; }'
dtrace: description 'syscall::read:entry ' matched 2 probes
^C
  ns
           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@                                       1124
            1024 |@@@@@@@                                  5108
            2048 |@@@@@@@@@@@@@@@@@@@@@@@@@@               15427
            4096 |@@@@@@                                   4391
            8192 |@                                        777
           16384 |@                                        425
```

```
 32768 |                                         114
 65536 |                                         5
131072 |                                         4
262144 |                                         4
524288 |                                         0
```

Podczas przedstawionej sesji monitorowania większość wywołań systemowych `read()` trwała od 1 µs do 8 µs (od 1024 ns do 8191 ns). Wywołanie systemowe `read()` operuje na deskryptorze pliku, który może być obiektem systemu plików lub gniazdem sieciowym. Identyfikacja wywołań zostanie przedstawiona w odpowiednich rozdziałach — odbywa się za pomocą tablicy `fds[]` narzędzia DTrace, używanej do mapowania deskryptorów plików na ich typy systemów plików.

W omawianym przykładzie, jeśli wbudowana wartość `timestamp` zostanie zmieniona na `vtimestamp`, to będzie oznaczać, że czas procesora jest mierzony tylko podczas wywołania systemowego. Takie rozwiązanie może być używane do porównania z czasem wykonywania, co pozwoli przekonać się, czy wywołanie systemowe przebywa więcej czasu w kodzie jądra czy zablokowane w oczekiwaniu na operację wejścia-wyjścia.

Istnieje możliwość tworzenia znacznie bardziej skomplikowanych skryptów DTrace w celu wyrażania na wiele różnych sposobów czasu dotyczącego wywołania systemowego. Przykłady mogą obejmować między innymi następujące skrypty (pochodzą z zestawu *DTraceToolkit*, patrz odwołanie [3] na końcu rozdziału):

- **dtruss**. Wersja `truss`, taka jak DTrace; działa na poziomie systemu.
- **execsnoop**. Monitorowanie wykonywania nowych procesów, odbywa się za pomocą wywołania systemowego `exec()`.
- **opensnoop**. Monitorowanie wywołań systemowych `open()` wraz z różnymi detalami.
- **procsystime**. Podsumowanie na wiele sposobów czasu dotyczącego wywołań systemowych.

Przedstawione rozwiązania pozwalają rozwikłać wiele kwestii związanych z wydajnością. Odbywa się to przez ogólną identyfikację aktywności procesu, którą można dostroić lub zupełnie wyeliminować. Stanowi to pewien rodzaj charakterystyki obciążenia: obciążeniem są wywołania systemowe wykonywane przez aplikację.

Na przykład przedstawione polecenie `execsnoop` wraz z opcją `-v` wyświetla znaczniki czasu w systemie działającym w chmurze:

```
# execsnoop -v
STRTIME                UID    PID   PPID ARGS
2013 Jan 12 22:10:05     0  15044  14378 /usr/bin/date +%M
2013 Jan 12 22:10:05     0  15039  15038 /opt/mon/bin/rrdtool graph /opt/mo...
2013 Jan 12 22:10:05     0  15037  15036 /opt/mon/bin/rrdtool update /opt/m...
2013 Jan 12 22:10:05     0  15041  15040 /opt/mon/bin/rrdtool graph /opt/mo...
2013 Jan 12 22:10:05     0  15043  15042 /opt/mon/bin/rrdtool graph /opt/mo...
2013 Jan 12 22:10:06     0  15046  15045 /usr/bin/echo
2013 Jan 12 22:10:06     0  15048  15045 /usr/bin/tail -200
2013 Jan 12 22:10:06     0  15049  15048 /usr/bin/cat -sv /var/adm/messages...
```

```
2013 Jan 12 22:10:06     0  15050  15049 /usr/bin/ls -tr1 /var/adm/messages...
2013 Jan 12 22:10:06     0  15045  14377 /usr/bin/sh /usr/bin/dmesg
[...]
```

Znaczniki czasu wskazują, że wszystkie procesy są wykonywane w czasie około 2 s. Wysoka liczba krótkotrwałych procesów może zużywać zasoby procesora i negatywnie wpływać na inne aplikacje, co wynika z konieczności wykonywania wywołań między procesorami (przełączania kontekstu MMU podczas kończenia działania procesu).

5.4.4. Profilowanie operacji wejścia-wyjścia

Profilowanie operacji wejścia-wyjścia pełni podobną rolę jak profilowanie procesora, ale pomaga także w ustaleniu, dlaczego i jak wykonywane są wywołania systemowe związane z operacjami wejścia-wyjścia. Do tego celu używa się narzędzia DTrace i analizuje stosy na poziomie użytkownika dla wywołań systemowych.

Na przykład przedstawione tutaj polecenie powoduje monitorowanie wywołań systemowych `read()` wykonywanych przez PostgreSQL, zebranie informacji dotyczących stosu na poziomie użytkownika oraz agregację zebranych danych:

```
# dtrace -n 'syscall::read:entry /execname == "postgres"/ {
    @[ustack()] = count(); }'
dtrace: description 'syscall::read:entry ' matched 1 probe
^C
[...]
              libc.so.1`__read+0x15
              postgres`XLogRead+0xb7
              postgres`XLogSend+0x115
              postgres`WalSenderMain+0x10c6
              postgres`PostgresMain+0x1aa
              postgres`ServerLoop+0x6fe
              postgres`PostmasterMain+0x7e2
              postgres`main+0x412
              postgres`_start+0x83
              210
              libc.so.1`__read+0x15
              postgres`WaitLatchOrSocket+0xb1
              postgres`PgstatCollectorMain.isra.21+0x2ed
              postgres`pgstat_start+0x68
              postgres`reaper+0x5bd
              libc.so.1`__sighndlr+0x15
              libc.so.1`call_user_handler+0x292
              libc.so.1`sigacthandler+0x77
              libc.so.1`syscall+0x13
              libc.so.1`thr_sigsetmask+0x1c2
              libc.so.1`sigprocmask+0x52
              postgres`ServerLoop+0xb7
              postgres`PostmasterMain+0x7e2
              postgres`main+0x412
              postgres`_start+0x83
            10723
```

Dane wyjściowe (skrócone) pokazują stosy na poziomie użytkownika oraz zliczone liczby ich wystąpień. Wspomniane stosy zawierają nazwy wewnętrznych funkcji aplikacji. Prawdopodobnie nie będziesz w stanie zrozumieć tych danych wyjściowych, o ile nie przeprowadzisz analizy kodu źródłowego. Jednak zdołasz zebrać wystarczająco dużo użytecznych informacji na podstawie nazw funkcji. Stos pierwszy zawiera `XLogRead:`; może być powiązany z rodzajem dziennika zdarzeń w bazie danych. Stos drugi zawiera `PgstatCollectorMain.isra`; nazwa wskazuje na pewne działania z zakresu monitorowania.

Stosy dostarczają informacji o tym, *dlaczego* wykonywane są dane wywołania systemowe. Użyteczne może być przeanalizowanie dodatkowych atrybutów metodologii charakterystyki obciążenia, o czym wspomniano już w rozdziale 2. „Metodologia":

- **Kto?** Identyfikator procesu, nazwa użytkownika.
- **Co?** Komponent wykonujący dane wywołanie systemowe (na przykład system plików lub gniazdo), wielkość operacji wejścia-wyjścia, wartość IOPS, przepustowość (wyrażona w bajtach na sekundę), inne atrybuty.
- **Jak?** Zmiana wartości IOPS na przestrzeni czasu.

Oprócz zastosowanego obciążenia można przeanalizować jeszcze wydajność wynikową — opóźnienie wywołania systemowego — jak wspomniano przy okazji poprzedniej metodologii.

5.4.5. Charakterystyka obciążenia

Aplikacja zleca wykonywanie zadań zasobom systemu — procesorom, pamięci, systemowi plików, dyskom i sieci — przez użycie wywołań systemowych, podobnie jak ma to miejsce w przypadku systemu operacyjnego. Te wszystkie działania można przeanalizować za pomocą metodologii charakterystyki obciążenia, którą przedstawiono w rozdziale 2. „Metodologia" oraz omówiono w kolejnych rozdziałach.

Ponadto przeanalizować można obciążenie aplikacji. Tutaj należy się skoncentrować na operacjach wykonywanych przez aplikację i na ich atrybutach. To może być kluczowa metryka w monitorowaniu wydajności i podczas planowania pojemności.

5.4.6. Metoda USE

Metodę USE wprowadzono w rozdziale 2. „Metodologia" i zastosowano w kolejnych rozdziałach. Pozwala ona na sprawdzenie poziomu wykorzystania, nasycenia i błędów we wszystkich zasobach sprzętowych. Wiele problemów związanych z wydajnością można usunąć dzięki użyciu metody USE, która wskaże zasób będący wąskim gardłem.

Metodę USE można zastosować także względem zasobów w postaci oprogramowania, choć ta możliwość zależy od danej aplikacji. Jeżeli jesteś w stanie zdobyć wykres funkcjonalny pokazujący wewnętrzne komponenty aplikacji, przeanalizuj poziomy wykorzystania, nasycenia i błędów dla każdego zasobu oprogramowania i przekonaj się, co ma sens.

Na przykład aplikacja może używać puli wątków roboczych w celu przetwarzania żądań oraz kolejki przeznaczonej dla żądań oczekujących na swoją kolej. Traktując wspomniane trzy metryki jako zasób, można je zdefiniować następująco:

- **Poziom wykorzystania**. Średnia liczba wątków zajętych przetwarzaniem żądań w pewnym przedziale czasu, podana jako procent wszystkich wątków. Na przykład wartość 50% oznacza, że średnio połowa wątków była zajęta przetwarzaniem żądań.
- **Poziom nasycenia**. Średnia długość kolejki żądań w pewnym przedziale czasu. Ta wartość określa, ile żądań czeka na przetworzenie przez wątek roboczy.
- **Błędy**. Żądania odrzucone lub zakończone niepowodzeniem z jakiegokolwiek powodu.

Twoje zadanie polega na znalezieniu sposobu pomiaru wymienionych metryk. Odpowiednie wartości mogą być dostarczane przez aplikację lub trzeba będzie je dodać bądź zmierzyć za pomocą innych narzędzi, na przykład oferujących możliwość monitorowania dynamicznego.

Systemy kolejkowe, jak wymieniony w przykładzie, można przeanalizować za pomocą teorii kolejek (patrz rozdział 2. „Metodologia").

Spójrzmy teraz na inny przykład, obejmujący deskryptory plików. System może nakładać pewne ograniczenia, ponieważ ilość zasobów nie jest nieskończona. Trzy wspomniane wcześniej metryki można więc zdefiniować następująco:

- **Poziom wykorzystania**. Liczba używanych deskryptorów plików, podana jako wartość procentowa dozwolonego limitu.
- **Poziom nasycenia**. Zależy od zachowania systemu operacyjnego. Jeżeli podczas oczekiwania na alokację deskryptora pliku wątek pozostaje zablokowany, wówczas to może być liczba zablokowanych wątków oczekujących na dany zasób.
- **Błędy**. Błędy alokacji, na przykład typu EFILE lub „Zbyt wiele otwartych plików".

Przedstawione ćwiczenie powtórz dla wszystkich komponentów aplikacji i pomiń te metryki, które nie mają sensu.

Taki proces może pomóc w przygotowaniu krótkiej listy rzeczy do sprawdzenia w aplikacji przed przejściem do innych metodologii, takich jak analiza drążąca.

5.4.7. Analiza drążąca

W przypadku aplikacji analiza drążąca może się rozpoczynać od sprawdzenia operacji wykonywanych przez dany program, by następnie przejść w głąb aplikacji i zobaczyć, jak są one przeprowadzane. Dla operacji wejścia-wyjścia tego rodzaju analiza drążąca może obejmować biblioteki systemowe, wywołania systemowe oraz jądro.

To zdecydowanie jest zaawansowane zadanie, które bardzo szybko doprowadzi analityka do wewnętrznych komponentów aplikacji. W idealnej sytuacji powinny być one typu open source, co pozwoli na ich analizę. Narzędzia oferujące funkcję monitorowania dynamicznego (na przykład DTrace, SystemTap lub `perf`) mają instrumenty

przeznaczone dla wspomnianych komponentów wewnętrznych i mogą prowadzić monitorowanie w pewnych językach łatwiej niż inne narzędzia. Sprawdź, czy używany język oferuje własny zestaw narzędzi przeznaczonych do analizy, których użycie może okazać się lepszym rozwiązaniem.

Istnieją również określone narzędzia przeznaczone do analizy wywołań bibliotek: `ltrace` w systemie Linux i `apptrace` w Solarisie (choć jego użycie skłania do DTrace).

5.4.8. Analiza blokad

W aplikacjach wielowątkowych blokady mogą stać się wąskim gardłem, negatywnie wpływającym na współbieżność i skalowalność. Analizę można przeprowadzić pod kątem:

- sprawdzenia, czy występuje zjawisko rywalizacji;
- sprawdzenia, czy blokady są nakładane na długi czas.

Punkt pierwszy pozwala ustalić, czy problem *już* występuje. Długi czas blokady niekoniecznie jest problemem, ale może się nim stać w przyszłości, gdy wzrośnie poziom jednoczesnego obciążenia. W przypadku obu punktów warto ustalić nazwę blokady (o ile istnieje) i ścieżkę kodu prowadzącą do jej nałożenia.

Wprawdzie istnieją specjalne narzędzia przeznaczone do analizy blokad, ale problem można czasami rozwiązać, stosując jedynie profilowanie procesora. W *blokadach wirujących* stan rywalizacji przejawia się w postaci zużycia procesora i można go łatwo wychwycić podczas profilowania procesora. Z kolei w *adaptacyjnych blokadach muteksu* stan rywalizacji często oznacza istnienie pewnych blokad, co również można wychwycić za pomocą profilowania procesora. Warto jednak pamiętać, że profilowanie procesora nie daje pełnego obrazu sytuacji, ponieważ wątki mogą być blokowane i uśpione podczas oczekiwania na zniesienie blokad. Zapoznaj się z punktem 5.4.2, zatytułowanym „Profilowanie procesora”.

Oto przykłady specjalnych narzędzi w systemie Solaris przeznaczonych do analizy blokad:

- **`plockstat`** — analiza blokad na poziomie użytkownika,
- **`lockstat`** — analiza blokad na poziomie jądra.

Zachowanie wymienionych poleceń jest podobne. Zostały zaimplementowane za pomocą narzędzia DTrace, które można wykorzystać do bezpośredniego przeprowadzenia znacznie dokładniejszej analizy blokad.

To jest przykład użycia polecenia `lockstat`:

```
# lockstat -n 1000000 -C -s5 sleep 5 > lockstat.txt
# more lockstat.txt
Adaptive mutex spin: 134438 events in 5.058 seconds (26577 events/sec)
-------------------------------------------------------------------------------
Count indv cuml rcnt     nsec Lock                   Caller
14144  11%  11% 0.00     1787 0xffffff0d71404348     zfs_range_unlock+0x2a
      nsec ------ Time Distribution ------ count     Stack
```

```
       256 |@                                902        zfs_read+0x239
       512 |@@@@                             1948       fop_read+0x8b
      1024 |@@@@@@                           3033       read+0x2a7
      2048 |@@@@@@@@@@                       4286       read32+0x1e
      4096 |@@@@@@                           3143
      8192 |@                                656
     16384 |                                 148
     32768 |                                 24
     65536 |                                 2
    131072 |                                 0
    262144 |                                 0
    524288 |                                 0
   1048576 |                                 2
---------------------------------------------------------------------------
Count indv cuml rcnt     nsec Lock                   Caller
13701  10%  21% 0.00     1769 0xffffff0d71404348     zfs_range_lock+0x86
      nsec ------ Time Distribution ------ count     Stack
       256 |@@                               1119       zfs_read+0x101
       512 |@@@@                             1970       fop_read+0x8b
      1024 |@@@                              1492       read+0x2a7
      2048 |@@@@@@@@@@                       4976       read32+0x1e
      4096 |@@@@@@@                          3469
      8192 |@                                520
     16384 |                                 124
     32768 |                                 24
     65536 |                                 7
---------------------------------------------------------------------------
[...]
Adaptive mutex block: 399 events in 5.058 seconds (79 events/sec)
---------------------------------------------------------------------------
Count indv cuml rcnt     nsec Lock                   Caller
   21   5%   5% 0.00    21053 0xffffff0d71404348     zfs_range_unlock+0x2a
      nsec ------ Time Distribution ------ count     Stack
      8192 |@@@@@                            4          zfs_read+0x239
     16384 |@@@@                             3          fop_read+0x8b
     32768 |@@@@@@@@@@@@@@@                  11         read+0x2a7
     65536 |@@@@                             3          read32+0x1e
---------------------------------------------------------------------------
Count indv cuml rcnt     nsec Lock                   Caller
   20   5%  10% 0.00    15107 0xffffff0d71404348     zfs_range_lock+0x86
      nsec ------ Time Distribution ------ count     Stack
      8192 |@@@@                             3          zfs_read+0x101
     16384 |@@@@@@@@@@@@                     8          fop_read+0x8b
     32768 |@@@@@@@@@@@@                     9          read+0x2a7
                                                      read32+0x1e
[...]
Spin lock spin: 2174 events in 5.058 seconds (430 events/sec)
---------------------------------------------------------------------------
Count indv cuml rcnt     nsec Lock                   Caller
  589  27%  27% 0.00    73972 cp_default            disp_lock_enter+0x26
      nsec ------ Time Distribution ------ count     Stack
       512 |@@@                              65         disp_getbest+0x28
```

```
      1024 |@@@@@@@@@@@@@            239       disp_getwork+0x37
      2048 |@                        23        idle+0x55
      4096 |                         6         thread_start+0x8
      8192 |@                        39
     16384 |@                        37
     32768 |@@                       53
     65536 |@@                       43
    131072 |@@                       45
    262144 |                         18
    524288 |                         3
   1048576 |                         7
   2097152 |                         4
   4194304 |                         7
[...]
R/W reader blocked by writer: 1 events in 5.058 seconds (0 events/sec)
-------------------------------------------------------------------------------
Count indv cuml rcnt     nsec Lock                   Caller
    1 100% 100% 0.00 259688397 0xffffff0d6f0cf898     as_fault+0x2a9
      nsec ------ Time Distribution ------ count     Stack
 268435456 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1       pagefault+0x96
                                                     trap+0x2c7
                                                     cmntrap+0xe6
-------------------------------------------------------------------------------
```

W przedstawionym przykładzie polecenie `lockstat` monitorowało zdarzenia rywalizacji (`-C`) na pięciu poziomach stosu (`-s5`) i zostało uruchomione wraz z koprocesem (`sleep(1)`) w celu dostarczenia czasu utraty ważności wynoszącego 5 s. Dane zostały przekierowane do pliku, aby można była łatwiej je przeglądać (to ponad 100 000 wierszy).

Dane wyjściowe zaczynają się od przedstawienia czasu blokad adaptacyjnych i rozkładu punktów pokazujących czas każdego zdarzenia rywalizacji wraz z nazwą blokady i stosem. Najczęściej występowała blokada `zfs_range_unlock`, która miała 14 144 wystąpienia i średni czas trwania wynoszący 1,787 ns. Rozkład pokazuje, że istnieją dwa wystąpienia o czasie trwania przekraczającym 1 048 576 ns (w przedziale od 1 ms do 2 ms). Liczbę tych blokad można sprawdzić w danych wyjściowych dotyczących adaptacyjnych blokad muteksu.

Monitorowanie blokad na poziomie jądra lub użytkownika wiąże się z pewnym obciążeniem. Wymienione tutaj narzędzia są oparte na DTrace, co w maksymalnym stopniu ogranicza wspomniane obciążenie. Jak wcześniej wspomniano, alternatywne rozwiązanie polega na profilowaniu procesora na stałej częstotliwości (na przykład 97 Hz) — pozwoli to wykryć wiele problemów związanych z blokadami (choć nie wszystkie) i nie spowoduje przy tym obciążenia nieodłącznie towarzyszącego monitorowaniu.

5.4.9. Statyczne dostosowanie wydajności

Statyczne dostosowanie wydajności koncentruje się na problemach występujących w skonfigurowanym środowisku. W przypadku wydajności aplikacji należy przeanalizować wymienione aspekty konfiguracji statycznej:

- Jaka jest wersja uruchomionej aplikacji? Czy są dostępne jej nowsze wersje? Czy w informacjach dotyczących nowych wydań pojawiają się jakiekolwiek wzmianki o poprawie wydajności?
- Jakie są znane problemy z wydajnością w danej aplikacji? Czy dla aplikacji istnieje baza danych pozwalająca na przeglądanie zgłoszonych błędów?
- W jaki sposób aplikacja została skonfigurowana?
- Jeżeli została skonfigurowana lub dostrojona odmiennie od jej ustawień domyślnych, to jaki był powód zastosowania takiej konfiguracji? Czy konfigurację przygotowano na podstawie pomiarów i analizy, czy po prostu losowo dobierając wartości?
- Czy aplikacja wykorzystuje buforowanie obiektów? Jaka jest wielkość bufora?
- Czy aplikacja pozwala na współbieżność? Jak to zostało skonfigurowane (na przykład czy zastosowano pulę wątków)?
- Czy aplikacja działa w trybie specjalnym? (Na przykład mógł zostać włączony tryb debugowania, który powoduje spadek wydajności działania aplikacji).
- Z jakich bibliotek systemowych korzysta aplikacja? W jakich są one wersjach?
- Jaki rodzaj alokacji pamięci jest stosowany w aplikacji?
- Czy aplikacja została skonfigurowana do użycia ogromnych stron dla sterty?
- Czy aplikacja została skompilowana? Jakiej wersji kompilatora użyto? Jakie zastosowano opcje i optymalizacje kompilatora? Czy aplikacja jest w wersji 64-bitowej?
- Czy w aplikacji wystąpił jakikolwiek błąd i teraz działa ona w trybie zdegradowanym?
- Czy w systemie zastosowano jakiekolwiek mechanizmy kontroli zasobów dla procesora, pamięci, systemu plików, dysku lub sieci? (Taka sytuacja najczęściej występuje w przetwarzaniu w chmurze).

Udzielenie odpowiedzi na wymienione pytania może wskazać różne opcje konfiguracyjne, które wcześniej zostały przeoczone.

5.5. Ćwiczenia

1. Odpowiedz na następujące pytania związane z terminologią:
   - Co to jest bufor?
   - Co to jest bufor cykliczny?
   - Co to jest bufor wirujący?
   - Co to jest adaptacyjna blokada muteksu?
   - Jaka jest różnica między współbieżnością i równoległością?
   - Co oznacza powiązanie z procesorem?

2. Odpowiedz na pytania i wykonaj polecenia dotyczące koncepcji:
   - Jakie są ogólne wady i zalety użycia ogromnych operacji wejścia-wyjścia?
   - Do czego jest używana tabela hash blokad?
   - Przedstaw ogólną charakterystykę wydajności środowiska języków kompilowanych, języków interpretowanych i tych, które używają maszyn wirtualnych.
   - Wyjaśnij rolę mechanizmu usuwania nieużytków i jego wpływ na wydajność.
3. Wybierz aplikację i odpowiedz na podstawowe pytania na jej temat:
   - Jaka jest rola tej aplikacji?
   - Jakie odmienne operacje są przez nią wykonywane?
   - W jakim trybie działa: użytkownika czy jądra?
   - W jaki sposób została skonfigurowana? Jakie są kluczowe opcje związane z jej wydajnością?
   - Jakie metryki wydajności są przez nią dostarczane?
   - Jakie dzienniki zdarzeń są tworzone przez aplikację? Czy znajdują się w nich jakiekolwiek informacje dotyczące wydajności?
   - Czy w najnowszych jej wersjach zostały usunięte problemy związane z wydajnością?
   - Czy w aplikacji istnieją znane błędy wpływające na jej wydajność?
   - Czy aplikacja ma swoją społeczność (na przykład poświęcony jej kanał IRC, spotkania użytkowników)? Czy to jest społeczność koncentrująca się na wydajności aplikacji?
   - Czy aplikacji poświęcono jakiekolwiek książki? Czy to są książki dotyczące wydajności?
   - Czy istnieją jacyś uznani eksperci w zakresie danej aplikacji? Kim oni są?
4. Wybierz aplikację znajdującą się pod obciążeniem, a następnie wykonaj następujące zadania (wiele z nich będzie wymagało zastosowania monitorowania dynamicznego):
   - Przed wykonaniem jakichkolwiek pomiarów odpowiedz na pytanie: Jakiego rodzaju ograniczeń oczekujesz — związanych z procesorem czy operacjami wejścia-wyjścia? Wyjaśnij powody.
   - Za pomocą narzędzi monitorowania ustal, co jest czynnikiem ograniczającym wydajność działania aplikacji: procesor czy operacje wejścia-wyjścia.
   - Scharakteryzuj wielkość przeprowadzanych operacji wejścia-wyjścia, na przykład odczyt i zapis systemu plików, otrzymywanie i wysyłanie danych przez sieć itd.
   - Czy aplikacja korzysta z bufora? Ustal jego wielkość i współczynnik trafności.
   - Dokonaj pomiaru opóźnienia (czasu udzielania odpowiedzi) dla operacji wykonywanych przez aplikację. Określ wartość średnią, minimalną, maksymalną oraz pełny rozkład.

- Przeprowadź analizę drążącą operacji, sprawdź źródło większości opóźnień.
- Scharakteryzuj obciążenie stosowane w aplikacji, w szczególności odpowiedz na pytania: kto i dlaczego.
- Przygotuj listę statycznego dostrojenia wydajności aplikacji.
- Czy aplikacja pozwala na współbieżność? Sprawdź jej możliwości w zakresie stosowania synchronizacji podstawowej.

5. (Opcjonalne, zaawansowane) Opracuj dla systemu Linux narzędzie o nazwie `tsastat`, którego zadaniem jest wyświetlenie kolumn dla wszystkich sześciu stanów stosowanych podczas analizy wątku oraz czasu spędzonego w poszczególnych stanach. Działanie narzędzia będzie zbliżone do `pidstat`, podobnie jak generowane dane wyjściowe.

5.6. Odwołania

[Knuth 76] D. Knuth, *Big Omicron and Big Omega and Big Theta*, „ACM SIGACT News", 1976.

[Knuth 97] D. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, wyd. 3, Addison-Wesley, 1997.

[1] *http://lwn.net/Articles/314512/*

[2] *http://nodejs.org/*

[3] *http://www.brendangregg.com/dtrace.html#DTraceToolkit*

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z

Ź

Ż

Zbuduj wydajny system komputerowy na miarę Twoich potrzeb!

Wszędzie tam, gdzie przetwarzane są gigantyczne ilości danych, wydajność systemu komputerowego ma fundamentalne znaczenie. W takim środowisku nawet minimalne opóźnienie pomnożone przez liczbę operacji może skutkować ogromnym wzrostem kosztów działania. Zastanawiasz się, jak zapewnić najwyższą wydajność systemów posiadanych fizycznie lub tych uruchomionych w chmurze? Jesteś inżynierem odpowiedzialnym za wydajność systemu komputerowego? A może pasjonujesz się systemami operacyjnymi i ich wydajnością? Jeżeli odpowiedziałeś twierdząco na którekolwiek z tych pytań, trafiłeś na idealny podręcznik!

W trakcie lektury przekonasz się, jak kontrolować oraz poprawiać wydajność Twoich systemów komputerowych. Poznasz popularne metodologie badań wydajności, ich zalety oraz wady. Ponadto nauczysz się planować pojemność systemu oraz go monitorować. Po tym wstępie przyjdzie czas na system Linux w detalach. Odkryjesz specyfikę jądra systemu Linux, sposób zarządzania procesami oraz pamięcią. Dzięki lekturze kolejnych rozdziałów zdobędziesz szczegółową wiedzę na temat procesorów, systemów plików, dysków oraz sieci. Każdy z tych elementów ma kluczowe znaczenie dla wydajności konfigurowanego systemu. Książka ta jest wyjątkowym, kompletnym kompendium wiedzy na temat wydajności systemów — zajrzyj koniecznie!

DZIĘKI TEJ KSIĄŻCE:

- zdobędziesz szczegółową wiedzę na temat systemu Linux
- nauczysz się korzystać z narzędzi do badania wydajności systemu
- przeprowadzisz wiarygodne testy wydajności
- wybierzesz odpowiedni sprzęt dla Twojego systemu komputerowego
- porównasz możliwości systemów działających w chmurze i systemów lokalnych
- zrozumiesz wpływ poszczególnych elementów systemu na wydajność

ISBN 978-83-246-9157-9
9 788324 691579
cena: 99,00 zł