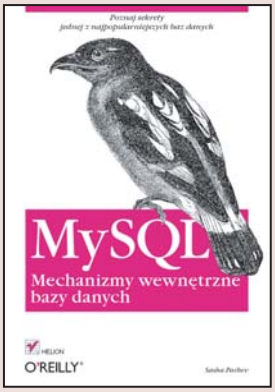
MySQL. Mechanizmy wewnętrzne bazy danych



Autor: Sasha Pachev
Tłumaczenie: Grzegorz Werner
ISBN: 978-83-246-1232-1
Tytuł oryginału: Understanding MySQL Internals
Format: B5, stron: 240

Poznaj sekrety jednej z najpopularniejszych baz danych

- Jak przechowywane są dane?
- Jak dodawać własne zmienne konfiguracyjne?
- Jak przebiega proces replikacji?

MySQL to obecnie jedna z najpopularniejszych baz danych. Jedną z jej największych zalet jest nieodpłatny dostęp zarówno do samego systemu, jak i do jego kodu źródłowego. Możliwość przeglądania kodu i – w razie potrzeby – samodzielnego modyfikowania go może okazać się przydatna programistom tworzącym aplikacje, które korzystają z MySQL jako zaplecza bazodanowego. Jednak samodzielne „przegryzanie się" przez setki tysięcy linii kodu i rozpracowywanie mechanizmów działania bazy danych może zająć mnóstwo czasu.

Dzięki tej książce poznasz kod źródłowy i sposób działania tego narzędzia. Autor, przez wiele lat pracujący w zespole tworzącym MySQL, przedstawia w niej tajniki systemu. Podczas czytania poznasz architekturę i wzajemne powiązania pomiędzy komponentami MySQL, strukturę kodu źródłowego oraz metody modyfikowania go przez kompilacją. Dowiesz się także, jak przebiega komunikacja pomiędzy klientem i serwerem bazy danych, jak realizowane są zapytania, w jaki sposób składowane są dane i jak implementowane są mechanizmy replikacji.

- Architektura MySQL
- Struktura kodu źródłowego
- Komunikacja pomiędzy klientem i serwerem
- Zmienne konfiguracyjne
- Obsługa żądań
- Parser i optymalizator zapytań
- Mechanizmy składowania danych
- Replikacja danych

Dzięki tej książce zrozumiesz budowę bazy danych MySQL i będziesz w stanie samodzielnie dostosować ją do każdego zadania

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 6.

Wątkowa obsługa żądań

Podczas pisania kodu serwera programista staje przed dylematem: czy obsługiwać żądania za pomocą wątków, czy procesów? Oba podejścia mają swoje zalety i wady. Od samego początku MySQL korzystał z wątków. W tym rozdziale uzasadnimy wątkową obsługę żądań w serwerze MySQL, a także omówimy jej wady i zalety oraz implementację.

Wątki kontra procesy

Być może najważniejszą różnicą między procesem a wątkiem jest to, że wątek potomny współdzieli stertę (globalne dane programu) z wątkiem macierzystym, a proces potomny — nie. Ma to pewne konsekwencje, które trzeba uwzględnić podczas wybierania jednego albo drugiego modelu.

Zalety wątków

Wątki są implementowane w bibliotekach programistycznych i systemach operacyjnych z następujących powodów:

- Zmniejszone wykorzystanie pamięci. Koszty pamięciowe związane z tworzeniem nowego wątku są ograniczone do stosu oraz do pamięci ewidencyjnej używanej przez menedżer wątków.
- Dostęp do globalnych danych serwera bez użycia zaawansowanych technik. Jeśli dane mogą zostać zmodyfikowane przez inny działający równolegle wątek, wystarczy chronić odpowiednią sekcję za pomocą blokady ze wzajemnym wykluczaniem, zwanej **muteksem** (opisywanej w dalszej części rozdziału). Jeśli nie ma takiej możliwości, dostęp do globalnych danych można uzyskiwać w taki sposób, jakby nie było żadnych wątków.
- Tworzenie wątku zajmuje znacznie mniej czasu niż tworzenie procesu, ponieważ nie trzeba kopiować segmentu sterty, który może być bardzo duży.
- Program szeregujący w jądrze szybciej przełącza konteksty między wątkami niż między procesami. Dzięki temu w mocno obciążonym serwerze procesor ma więcej czasu na wykonywanie rzeczywistej pracy.

Wady wątków

Wątki odgrywają ważną rolę we współczesnych systemach komputerowych, ale mają również pewne wady:

- Pomyłki programistyczne są bardzo kosztowne. Awaria jednego wątku powoduje załamanie całego serwera. Jeden wyrodny wątek może uszkodzić globalne i zakłócić działanie innych wątków.
- Łatwo popełnić pomyłkę. Programista musi stale myśleć o problemach, jakie może spowodować jakiś inny wątek, oraz o tym, jak ich uniknąć. Niezbędna jest bardzo defensywna postawa.
- Wielowątkowe serwery są znane z usterek synchronizacyjnych, które są trudne do odtworzenia podczas testów, ale ujawniają się w bardzo złym momencie w środowiskach produkcyjnych. Wysokie prawdopodobieństwo występowania takich usterek jest następstwem współdzielenia przestrzeni adresowej, co znacznie zwiększa stopień interakcji między wątkami.
- W pewnych okolicznościach rywalizacja o blokady może wymknąć się spod kontroli. Jeśli zbyt wiele wątków próbuje jednocześnie pozyskać ten sam muteks, może to doprowadzić do nadmiernego przełączania kontekstów: procesor przez większość czasu zamiast użytecznej pracy wykonuje program szeregujący.
- W systemach 32-bitowych przestrzeń adresowa procesu jest ograniczona do 4 GB. Ponieważ wszystkie wątki współdzielą tę samą przestrzeń adresową, teoretycznie cały serwer ma do dyspozycji 4 GB pamięci RAM, nawet jeśli w komputerze zainstalowano znacznie więcej fizycznej pamięci. W praktyce przestrzeń adresowa robi się bardzo zatłoczona przy znacznie mniejszym limicie, gdzieś około 1,5 GB w Linuksie x86.
- Zatłoczona 32-bitowa przestrzeń adresowa stwarza jeszcze jeden problem: każdy wątek potrzebuje trochę miejsca na stos. Kiedy stos zostaje przydzielony, to nawet jeśli wątek korzysta z niego w minimalnym stopniu, konieczne jest zarezerwowanie części przestrzeni adresowej serwera. Każdy nowy wątek ogranicza miejsce, które można przeznaczyć na stertę. Jeśli więc nawet w komputerze jest dużo fizycznej pamięci, może się okazać, że nie da się jednocześnie zaalokować dużych buforów, uruchomić wiele współbieżnych wątków oraz zapewnić każdemu z nich dużo miejsca na stos.

Zalety rozwidlonych procesów

Wady wątków odpowiadają zaletom korzystania z oddzielnych procesów:

- Pomyłki programistyczne nie są tak katastrofalne. Choć niekontrolowany proces może zakłócić działanie całego serwera, jest to znacznie mniej prawdopodobne.
- Trudniej popełnić pomyłkę. Przez większość czasu programista może myśleć tylko o jednym wątku wykonania, nie martwiąc się o potencjalnych intruzów.
- Pojawia się znacznie mniej fantomowych usterek. Kiedy wystąpi jakaś usterka, zwykle można łatwo ją odtworzyć. Każdy rozwidlony proces ma własną przestrzeń adresową, więc stopień ich wzajemnej interakcji jest znacznie mniejszy.
- W systemach 32-bitowych ryzyko wyczerpania przestrzeni adresowej jest dużo mniejsze.

Wady rozwidlonych procesów

Aby podsumować nasz przegląd, wymienię wady rozwidlonych procesów, które są lustrzanym odbiciem zalet wątków:

- Wykorzystanie pamięci jest nieoptymalne. Podczas rozwidlania procesu potomnego kopiowane są duże segmenty pamięci.
- Współdzielenie danych między procesami wymaga użycia specjalnych technik. Utrudnia to dostęp do globalnych danych serwera.
- Tworzenie procesu wiąże się z większymi kosztami na poziomie jądra. Konieczność kopiowania segmentu danych procesu macierzystego znacznie obniża wydajność. Linux jednak odrobinę tu oszukuje, stosując technikę zwaną **kopiowaniem przy zapisie**. Rzeczywiste kopiowanie strony procesu macierzystego zachodzi dopiero wtedy, gdy zostanie ona zmodyfikowana przez proces macierzysty lub potomny. Do tego momentu oba procesy używają jednej strony.
- Przełączanie kontekstów między procesami jest bardziej czasochłonne, ponieważ jądro musi przełączyć strony, tabele deskryptorów plików oraz inne dodatkowe informacje kontekstowe. Serwer ma mniej czasu na wykonywanie rzeczywistej pracy.

Ogólnie rzecz biorąc, serwer wątkowy jest idealny wtedy, gdy programy obsługi połączeń muszą współdzielić wiele danych, a programiście nie brakuje umiejętności. Kiedy trzeba było wybrać model odpowiedni dla MySQL, wybór był prosty. Serwer baz danych musi mieć wiele współużytkowanych buforów oraz innych współdzielonych danych.

Jeśli chodzi o umiejętności programistyczne, tych również nie brakowało. Podobnie jak dobry jeździec staje się jednością ze swoim koniem, tak Monty stał się jednością z komputerem. Bolało go, kiedy widział marnotrawienie zasobów systemowych. Był pewien, że potrafi napisać kod praktycznie pozbawiony usterek, poradzić sobie z problemami współbieżności powodowanymi przez wątki, a nawet pracować z małym stosem. Co za ekscytujące wyzwanie! Oczywiście, wybrał wątki.

Implementacja obsługi żądań

Serwer oczekuje na połączenia w swoim głównym wątku. Po odebraniu połączenia przydziela wątek do jego obsługi. W zależności od konfiguracji i bieżącego stanu serwera, wątek może zostać utworzony od zera albo przydzielony z pamięci podręcznej (puli) wątków. Klient przesyła żądania, a serwer je realizuje, dopóki klient nie wyda polecenia kończącego sesję (`COM_QUIT`) albo sesja nie zostanie nieoczekiwanie przerwana. Po zakończeniu sesji, w zależności od konfiguracji i bieżącego stanu serwera, wątek może zostać zakończony albo zwrócony do puli wątków w oczekiwaniu na następne połączenie.

Struktury, zmienne, klasy i interfejsy API

Jeśli chodzi o obsługę wątków, prawdopodobnie najważniejsza jest klasa `THD`, która reprezentuje deskryptor wątku. Niemal każda z funkcji parsera i optymalizatora przyjmuje obiekt `THD` jako argument, najczęściej pierwszy na liście argumentów. Klasę `THD` opisano szczegółowo w rozdziale 3.

Podczas tworzenia wątku deskryptor jest umieszczany na globalnej liście wątków `I_List<THD> threads` (`I_List<>` to szablonowa klasa połączonej listy; zob. *sql/sql_list.h* oraz *sql/sql_list.c*). Listy tej używa się do trzech podstawowych celów:

- dostarczanie danych na użytek polecenia `SHOW PROCESSLIST`;
- lokalizowanie docelowego wątku podczas wykonywania polecenia `KILL`;
- sygnalizowanie wszystkim wątkom, aby przerwały pracę, kiedy serwer jest zamykany.

Ważną rolę odgrywa inna lista `I_List<THD>`: `thread_cache`. Jest ona używana w dość nieoczekiwany sposób: jako metoda na przekazywanie obiektu `THD` utworzonego przez wątek główny do wątku oczekującego w puli, który został wyznaczony do obsługi bieżącego żądania. Więcej informacji można znaleźć w funkcjach `create_new_thread()`, `start_cached_thread()` oraz `end_thread()` w pliku *sql/mysqld.cc*.

Wszystkie operacje związane z tworzeniem, kończeniem i śledzeniem wątków są chronione przez muteks `LOCK_thread_count`. Do obsługi wątków używa się trzech zmiennych warunku POSIX. `CONT_thread_count` pomaga w synchronizacji podczas zamykania serwera, gwarantując, że wszystkie wątki dokończą swoją pracę przed zatrzymaniem wątku głównego. Warunek `COND_thread_cache` jest rozgłaszany, kiedy wątek główny postanawia obudzić buforowany wątek i skierować go do obsługi bieżącej sesji z klientem. Warunek `COND_flush_thread_cache` jest używany przez buforowane wątki do sygnalizowania, że zaraz zakończą pracę (podczas zamykania serwera albo przetwarzania sygnału `SIGHUP`).

Ponadto do obsługi wątków używa się kilku globalnych zmiennych stanu. Są one opisane w tabeli 6.1.

Wykonywanie żądań krok po kroku

Pętla realizacji standardowych żądań `select()/accept()` znajduje się w funkcji `handle_connections_sockets()` w pliku *sql/mysqld.cc*. Po dość skomplikowanej serii testów, które sprawdzają ewentualne błędy wywołania `accept()` na różnych platformach, docieramy do poniższego fragmentu kodu:

```
if (!(thd= new THD))
{
  (void) shutdown(new_sock,2);
  VOID(closesocket(new_sock));
  continue;
}
```

Tworzy on instancję `THD`. Po pewnych dodatkowych operacjach na obiekcie `THD` wykonanie przenosi się do funkcji `create_new_thread()` w tym samym pliku *sql/mysqld.cc*. Po przeprowadzeniu kilku kolejnych testów oraz inicjalizacji dochodzimy to instrukcji warunkowej, która ustala, jak zostanie uzyskany wątek obsługi żądania. Istnieją dwie możliwości: użyć buforowanego wątku albo utworzyć nowy.

Kiedy buforowanie wątków jest włączone, stary wątek po obsłużeniu żądania klienta nie kończy działania, lecz usypia. Gdy nowy klient nawiązuje połączenie, serwer nie tworzy od razu nowego wątku, lecz sprawdza, czy ma jakieś uśpione wątki w pamięci podręcznej. Jeśli tak, to budzi jeden z nich, przekazując mu instancję `THD` jako argument.

Tabela 6.1. Zmienne globalne związane z wątkami

Definicja zmiennej	Opis
`int abort_loop`	Znacznik, który sygnalizuje wątkom, że czas po sobie posprzątać i zakończyć pracę. Serwer nigdy nie wymusza przerwania wątku, ponieważ mogłoby to doprowadzić do poważnego uszkodzenia danych. Każdy wątek jest napisany w taki sposób, aby monitorował swoje środowisko i kończył działanie, kiedy serwer tego zażąda.
`int cached_thread_count`	Zmienna stanu śledząca liczbę wątków, które zakończyły działanie i oczekują na przydzielenie do obsługi innego żądania. Można ją obejrzeć w wynikach polecenia `SHOW STATUS` pod nagłówkiem `Threads_connected`.
`int kill_cached_thread`	Znacznik, który sygnalizuje wszystkim buforowanym wątkom, że powinny zakończyć działanie. Buforowane wątki czekają na warunek `COND_thread_cache` w funkcji `end_thread()`. Przerywają pracę, kiedy wykrywają, że ten znacznik jest ustawiony.
`int max_connections`	Zmienna konfiguracyjna serwera określająca maksymalną liczbę połączeń nieadministracyjnych, które serwer może przyjąć. Po osiągnięciu tego limitu administrator bazy danych może nawiązać jedno dodatkowe połączenie administracyjne, aby jakoś rozwiązać kryzys. Dzięki temu limitowi serwer może „wyhamować", zanim sparaliżuje system przez nadmierne wykorzystanie zasobów. Limit ten jest kontrolowany przez zmienną konfiguracyjną `max_connections` o domyślnej wartości 100.
`int max_used_connections`	Zmienna stanu śledząca maksymalną liczbę jednoczesnych połączeń odnotowaną od czasu uruchomienia serwera. Jej wartość można obejrzeć w wynikach polecenia `SHOW STATUS` pod nagłówkiem `Max_used_connections`.
`int query_id`	Zmienna używana do generowania unikatowych identyfikatorów zapytań. Każdemu zapytaniu przesłanemu do serwera przypisuje się bieżącą wartość tej zmiennej, która następnie jest zwiększana o 1.
`int thread_cache_size`	Zmienna konfiguracyjna serwera określająca maksymalną liczbę wątków w pamięci podręcznej wątków.
`int thread_count`	Zmienna stanu śledząca bieżącą liczbę wątków. Jej wartość można obejrzeć w wynikach polecenia `SHOW STATUS` pod nagłówkiem `Threads_cached`.
`int thread_created`	Zmienna stanu śledząca liczbę wątków utworzonych od momentu uruchomienia serwera. Jej wartość można obejrzeć w wynikach polecenia `SHOW STATUS` pod nagłówkiem `Threads_created`.
`int thread_id`	Zmienna używana do generowania unikatowych identyfikatorów wątków. Każdemu nowo utworzonemu wątkowi przypisuje się bieżącą wartość tej zmiennej, która następnie jest zwiększana o 1. Można ją obejrzeć w wynikach polecenia `SHOW STATUS` pod nagłówkiem `Connections`.
`int thread_running`	Zmienna stanu śledząca liczbę wątków, które obecnie odpowiadają na zapytanie. Zwiększana o 1 na początku funkcji `dispatch_command()` w pliku *sql/sql_parse.cc* i zmniejszana o jeden na końcu tej funkcji. Można ją obejrzeć w wynikach polecenia `SHOW STATUS` pod nagłówkiem `Threads_running`.

Choć buforowanie wątków może znacznie zwiększyć wydajność mocno obciążonego systemu, funkcję tę pierwotnie dodano w celu rozwiązania pewnych problemów synchronizacji w Linuksie na platformach Alpha.

Jeśli buforowanie wątków jest wyłączone albo żaden buforowany wątek nie jest dostępny, w celu obsłużenia żądania trzeba utworzyć nowy wątek.

Decyzja jest podejmowana w następującym bloku:

```
if (cached_thread_count > wake_thread)
{
   start_cached_thread(thd);
}
```

Funkcja `start_cached_thread()` z pliku *sql/mysqld.cc* budzi wątek, który obecnie nie obsługuje żądania, jeśli taki wątek istnieje. Warunek `cached_thread_count > wake_thread` gwarantuje istnienie uśpionego wątku, więc funkcja nigdy nie jest wywoływana, jeśli nie ma żadnych buforowanych wątków. Dotyczy to również sytuacji, w której pamięć podręczna wątków jest wyłączona.

Jeśli test dostępności buforowanych wątków zakończy się niepowodzeniem, kod przechodzi do bloku `else`, gdzie zadanie utworzenia nowego wątku przypada poniższemu wierszowi:

```
if ((error=pthread_create(&thd->real_id, &connection_attrib,
                          handle_one_connection,
                          (void*) thd)))
```

Nowy wątek zaczyna się od funkcji `handle_one_connection()` w pliku *sql/sql_parse.cc*.

Funkcja `handle_one_connection()` po kilku testach i inicjalizacjach bierze się do roboty:

```
while (!net->error && net->vio != 0 && !thd->killed)
{
  if (do_command(thd))
      break;
}
```

Polecenia są akceptowane i przetwarzane dopóty, dopóki nie wystąpi warunek zakończenia pętli. Oto możliwe warunki wyjścia:

- Błąd sieciowy.
- Wątek zostaje usunięty poleceniem `KILL` przez administratora bazy danych albo przez zamykany serwer.
- Klient wysyła żądanie `COM_QUIT`, informując serwer, że chce zakończyć sesję. W takim przypadku funkcja `do_command()` z pliku *sql/sql_parse.cc* zwraca wartość niezerową.
- Funkcja `do_command()` zwraca wartość niezerową z jakiejś innej przyczyny. Obecnie jedyną inną możliwością jest to, że nadrzędny serwer replikacji postanawia przerwać przesyłanie strumienia aktualizacji, którego serwer podrzędny (albo klient podszywający się pod serwer podrzędny) zażądał poleceniem `COM_BINLOG_DUMP`.

Następnie funkcja `handle_one_connection()` przechodzi do fazy kończenia wątku i porządkowania. Kluczowym elementem tego segmentu kodu jest wywołanie funkcji `end_thread()` z pliku *sql/mysqld.cc*.

Funkcja `end_thread()` zaczyna od pewnych dodatkowych czynności porządkowych, a następnie dociera do interesującego punktu: możliwości umieszczenia obecnie wykonywanego wątku w pamięci podręcznej. Decyzja jest podejmowana przez następującą instrukcję warunkową:

```
if (put_in_cache && cached_thread_count < thread_cache_size &&
      ! abort_loop && !kill_cached_threads)
```

Jeśli funkcja `end_thread()` postanowi zbuforować wątek, wykonywana jest poniższa pętla:

```
while (!abort_loop && ! wake_thread && ! kill_cached_threads)
      (void) pthread_cond_wait(&COND_thread_cache, &LOCK_thread_count);
```

Pętla czeka, aż wątek zostanie obudzony przez wywołanie `start_cached_thread()`, procedurę obsługi sygnału `SIGHUP` albo procedurę zamykania serwera. Jeśli sygnał budzenia pochodzi od funkcji `start_cached_thread()`, parametr `wake_thread` ma wartość niezerową. W takim przypadku kod pobiera obiekt `THD` przekazany przez `start_cached_thread()` z listy `thread_cache`, a następnie wraca (zwróćmy uwagę na makro `DBUG_VOID_RETURN`) do funkcji `handle_one_connection()`, aby zacząć obsługiwanie nowego klienta.

Jeśli wątek nie zostanie przeniesiony do pamięci podręcznej, ostatecznie kończy działanie przez wywołanie `pthread_exit()`.

Problemy programowania wątkowego

W MySQL występują podobne komplikacje co w innych programach, które używają wątków.

Wywołania standardowej biblioteki C

Podczas pisania kodu, który może być wykonywany przez kilka wątków jednocześnie, trzeba zachować szczególną ostrożność, jeśli chodzi o wywoływanie funkcji z zewnętrznych bibliotek. Zawsze istnieje pewne prawdopodobieństwo, że wywołany kod używa zmiennej globalnej, pisze we współdzielonym deskryptorze pliku albo używa jakiegoś innego wspólnego zasobu, nie gwarantując wzajemnego wykluczania. W takim przypadku trzeba zabezpieczyć wywołanie za pomocą muteksu.

Trzeba zachować ostrożność, a jednocześnie unikać nadmiernej ochrony, która może spowodować spadek wydajności. Na przykład można oczekiwać, że wywołanie `malloc()` jest bezpieczne dla wątków. Inne funkcje, takie jak `gethostbyname()`, często mają odpowiedniki bezpieczne dla wątków. Skrypty konfigurujące kompilację MySQL sprawdzają, czy są one dostępne i używają ich, kiedy tylko jest to możliwe. Jeśli odpowiednik bezpieczny dla wątków nie zostanie wykryty, w ostateczności włączany jest ochronny muteks.

Ogólnie rzecz biorąc, MySQL oszczędza sobie wielu zmartwień związanych z bezpieczeństwem wątków, implementując odpowiedniki wywołań standardowej biblioteki C w warstwie przenośności w *mysys* oraz w bibliotece łańcuchów w *strings*. Nawet jeśli ostatecznie wywoływana jest biblioteka C, to w większości przypadków odbywa się to za pośrednictwem nakładki. Jeśli w jakimś systemie okazuje się, że wywołanie nie jest bezpieczne dla wątków, można łatwo rozwiązać problem przez dodanie muteksu do nakładki.

Blokady z wzajemnym wykluczaniem (muteksy)

W serwerze wątkowym kilka wątków może mieć dostęp do współdzielonych danych. W takim przypadku każdy wątek musi zagwarantować, że będzie miał dostęp na wyłączność. W tym celu stosuje się blokady z wzajemnym wykluczaniem, zwane też muteksami.

W miarę jak zwiększa się złożoność aplikacji, trzeba zdecydować, ilu muteksów użyć i jakie dane powinny być chronione przez każdy z nich. Jedną skrajnością jest utworzenie oddzielnego muteksu dla każdej zmiennej. Ma to tę zaletę, że rywalizacja o muteksy jest ograniczona do minimum. Są również pewne wady: co się stanie, jeśli trzeba będzie uzyskać dostęp do grupy zmiennych w sposób atomowy? Konieczne będzie oddzielne pozyskanie każdego muteksu.

W takim przypadku trzeba zawsze pozyskiwać je w tej samej kolejności, aby uniknąć zakleszczeń. Częste wywołania funkcji `pthread_mutex_lock()` i `pthread_mutex_unlock()` doprowadzą do spadku wydajności, a programista prędzej czy później pomyli kolejność wywołań i spowoduje zakleszczenie.

Na drugim końcu spektrum znajduje się jeden muteks dla wszystkich zmiennych. Upraszcza to pracę programisty — wystarczy założyć blokadę podczas dostępu do zmiennej globalnej, a później ją zwolnić. Niestety, ma to bardzo negatywny wpływ na wydajność. Wiele wątków musi niepotrzebnie czekać, kiedy jeden z nich uzyskuje dostęp do zmiennej, która nie musi być chroniona przed innymi.

Rozwiązaniem jest odpowiednie pogrupowanie zmiennych globalnych i utworzenie muteksu dla każdej grupy. Właśnie w ten sposób postąpili twórcy MySQL.

W tabeli 6.2 znajduje się lista globalnych muteksów MySQL wraz z opisami grup zmiennych, które są przez nie chronione.

Tabela 6.2. Globalne muteksy

Nazwa muteksu	Opis muteksu
`LOCK_Acl`	Inicjalizowany, ale obecnie nieużywany w kodzie. W przyszłości może zostać usunięty.
`LOCK_active_mi`	Chroni wskaźnik `active_mi`, który wskazuje deskryptor aktywnego podrzędnego serwera replikacji. W tym momencie ochrona jest zbędna, ponieważ wartość `active_mi` nigdy nie jest zmieniana współbieżnie. Ochrona stanie się jednak konieczna, kiedy do serwera zostanie dodana obsługa wielu serwerów nadrzędnych.
`LOCK_bytes_received`	Chroni zmienną stanu `bytes_received`, która śledzi liczbę bajtów odebranych od wszystkich klientów od momentu uruchomienia serwera. Nieużywana w wersji 5.0 i nowszych.
`LOCK_bytes_sent`	Chroni zmienną stanu `bytes_sent`, która śledzi liczbę bajtów wysłanych do wszystkich klientów od momentu uruchomienia serwera. Nieużywana w wersji 5.0 i nowszych.
`LOCK_crypt`	Chroni wywołanie uniksowej biblioteki C `crypt()`, które nie jest bezpieczne dla wątków.
`LOCK_delayed_create`	Chroni zmienne i struktury zaangażowane w tworzenie wątku do obsługi opóźnionego wstawiania. Opóźnione operacje wstawiania natychmiast wracają do klienta, nawet jeśli tablica jest zablokowana — w takim przypadku są przetwarzane w tle przez wątek opóźnionego wstawiania.
`LOCK_delayed_insert`	Chroni listę wątków opóźnionego wstawiania `I_List<delayed_insert> delayed_threads`.
`LOCK_delayed_status`	Chroni zmienne stanu śledzące operacje opóźnionego wstawiania.
`LOCK_error_log`	Chroni zapisy w dzienniku błędów.
`LOCK_gethostbyname_r`	Chroni wywołanie `gethostbyname()` w funkcji `my_gethostbyname_r()` w pliku *mysys/my_gethostbyname.c* w systemach, w których biblioteka C nie oferuje wywołania `gethostbyname_r()`.
`LOCK_global_system_variables`	Chroni operacje modyfikujące globalne zmienne konfiguracyjne z poziomu wątku klienckiego.
`LOCK_localtime_r`	Chroni wywołanie `localtime()` w funkcji `my_localtime_r()` w pliku *mysys/my_pthread.c* w systemach, w których biblioteka C nie oferuje wywołania `localtime_r()`.
`LOCK_manager`	Chroni struktury danych używane przez wątek menedżera, który obecnie jest odpowiedzialny za okresowe wymuszanie zapisu tabel na dysku (jeśli ustawienie `flush_time` jest niezerowe) oraz za porządkowanie dzienników Berkeley DB.

Tabela 6.2. Globalne muteksy — ciąg dalszy

Nazwa muteksu	Opis muteksu
`LOCK_mapped_file`	Chroni struktury danych i zmienne używane do operacji na plikach odwzorowanych w pamięci. Obecnie funkcja ta jest wewnętrznie obsługiwana, ale nie jest używana w żadnej części kodu.
`LOCK_open`	Chroni struktury danych i zmienne związane z pamięcią podręczną tabel oraz z otwieraniem i zamykaniem tabel.
`LOCK_rpl_status`	Chroni zmienną `rpl_status`, która miała być używana do bezpiecznej replikacji z automatycznym przywracaniem danych. Obecnie jest to martwy kod.
`LOCK_status`	Chroni zmienne wyświetlane w wynikach polecenia `SHOW STATUS`.
`LOCK_thread_count`	Chroni zmienne i struktury danych zaangażowane w tworzenie lub niszczenie wątków.
`LOCK_uuid_generator`	Chroni zmienne i struktury danych używane przez funkcję SQL `UUID()`.
`THR_LOCK_charset`	Chroni zmienne i struktury danych związane z operacjami na zestawie znaków.
`THR_LOCK_heap`	Chroni zmienne i struktury danych związane z pamięciowym mechanizmem składowania (MEMORY).
`THR_LOCK_isam`	Chroni zmienne i struktury danych związane z mechanizmem składowania ISAM.
`THR_LOCK_lock`	Chroni zmienne i struktury danych związane z menedżerem blokad tabel.
`THR_LOCK_malloc`	Chroni zmienne i struktury danych związane z nakładkami na rodzinę wywołań `malloc()`. Używany głównie w wersji `malloc()` przeznaczonej do debugowania (zob. *mysys/safemalloc.c*).
`THR_LOCK_myisam`	Chroni zmienne i struktury danych związane z mechanizmem składowania MyISAM.
`THR_LOCK_net`	Obecnie używany do ochrony wywołania `inet_ntoa()` w funkcji `my_inet_ntoa()` w pliku *mysys/my_net.c*
`THR_LOCK_open`	Chroni zmienne i struktury danych, które śledzą otwarte pliki.

Oprócz muteksów globalnych istnieje kilka muteksów osadzonych w strukturach lub klasach, które służą do ochrony części danej struktury lub klasy. Istnieje wreszcie kilka muteksów globalnych o zasięgu plikowym (`static`) w bibliotece *mysys*.

Blokady odczytu-zapisu

Blokada na wyłączność nie zawsze jest najlepszym rozwiązaniem ochrony operacji współbieżnych. Wyobraźmy sobie sytuację, w której pewna zmienna rzadko jest modyfikowana tylko przez jeden wątek, natomiast często czytana przez wiele innych. Gdybyśmy użyli muteksu, zwykle jeden wątek czytający musiałby czekać, aż inny zakończy czytanie, choć mogłyby one wykonywać się współbieżnie.

W takich sytuacjach lepiej sprawdza się inny typ blokady: **blokada odczytu-zapisu**. Blokady odczytu mogą być współdzielone, a blokady zapisu wzajemnie się wykluczają. Zatem wiele wątków czytających może działać współbieżnie, pod warunkiem że nie ma wątku piszącego.

Jak widać, blokada odczytu-zapisu może robić to samo co muteks i więcej. Czemu więc nie używać tylko blokad odczytu-zapisu? Jak mówi przysłowie, nie ma nic za darmo. Dodatkowe funkcje wymagają bardziej złożonej implementacji. W rezultacie blokady odczytu-zapisu zajmują więcej cykli procesora, nawet gdy blokada zostanie pozyskana natychmiast.

Kiedy więc wybieramy typ blokady, musimy oszacować prawdopodobieństwo, że nie uda się jej uzyskać za pierwszym razem, i rozważyć, w jakim stopniu możemy je zmniejszyć przez zastąpienie muteksu blokadą zapisu-odczytu. Jeśli na przykład w typowych okolicznościach niepowodzeniem kończy się 1 na 1000 prób, to blokada zapisu-odczytu pomaga tylko co 999. raz, a w innych przypadkach marnuje czas procesora. Jeśli nawet przejście na blokadę zapisu-odczytu miałoby zmniejszyć prawdopodobieństwo niepowodzenia praktycznie do zera, to i tak nie jest tego warte.

Jeśli jednak prawdopodobieństwo niepowodzenia pierwszej próby wynosi 1:10, być może dodatkowe cykle procesora, dziewięciokrotnie poświęcone na testowanie blokady odczytu-zapisu, zostaną zrównoważone tym, że za 10. razem rzeczywiście uzyskamy blokadę i nie będziemy musieli czekać tak długo jak w przypadku muteksu. Z drugiej strony, jeśli zastosowanie blokady odczytu-zapisu w tej konkretnej sytuacji nie zmniejsza w znaczący sposób prawdopodobieństwa niepowodzenia pierwszej próby, to poświęcanie dodatkowych cykli procesora i tak może być nieopłacalne.

Regiony krytyczne w MySQL są zwykle dość krótkie, co przekłada się na niskie prawdopodobieństwo niepowodzenia pierwszej próby. Zatem w większości przypadków muteks okazuje się lepszy niż blokada odczytu-zapisu. W tabeli 6.3 wymienione są blokady odczytu-zapisu używane przez MySQL.

Tabela 6.3. Blokady odczytu-zapisu używane przez MySQL

Nazwa blokady odczytu-zapisu	Opis blokady odczytu-zapisu
`LOCK_grant`	Chroni zmienne i struktury danych związane z kontrolą dostępu.
`LOCK_sys_init_connect`	Chroni deskryptor zmiennej systemowej `sys_init_connect` przed modyfikacjami, kiedy wykonywane są zapisane w niej polecenia. Zmienna systemowa `sys_init_connect` przechowuje polecenia, które są wykonywane za każdym razem, kiedy z serwerem łączy się nowy klient. Polecenia te określa się za pomocą zmiennej konfiguracyjnej `init-connect`.
`LOCK_sys_init_slave`	Chroni deskryptor zmiennej systemowej `sys_init_slave` przed modyfikacjami, kiedy wykonywane są zapisane w niej polecenia. Zmienna systemowa `sys_init_slave` przechowuje polecenia, które są wykonywane przez serwer nadrzędny za każdym razem, kiedy łączy się z nim serwer podrzędny. Polecenia te określa się za pomocą zmiennej konfiguracyjnej `init-slave`.

Synchronizacja

W aplikacjach wątkowych często pojawia się problem synchronizacji wątków. Jeden wątek musi dowiedzieć się, że inny osiągnął pewien stan. Biblioteka POSIX Threads oferuje przeznaczony do tego mechanizm: **zmienne warunku**. Wątek czekający na warunek może wywołać `pthread_cond_wait()`, przekazując jako argument zmienną warunku oraz muteks używany w danym kontekście. Wywołanie również musi być chronione przez ten sam muteks. Wątek, który osiągnie określony stan, może zasygnalizować to czekającemu wątkowi przez wywołanie `pthread_cond_signal()` albo rozgłosić to za pomocą wywołania `pthread_cond_broadcast()`. Sygnał albo rozgłoszenie muszą również być chronione przez ten sam muteks, którego wątek oczekujący użył w wywołaniu `pthread_cond_wait()`. Warunek sygnałowy budzi tylko jeden wątek, który na niego oczekuje, podczas gdy rozgłoszenie budzi wszystkie czekające wątki.

MySQL używa kilku zmiennych warunku POSIX. Są one opisane w tabeli 6.4.

Tabela 6.4. Zmienne warunku używane przez MySQL

Nazwa zmiennej warunku	Opis zmiennej warunku
`COND_flush_thread_cache`	Sygnalizowana przez `end_thread()` w pliku *sql/mysqld.cc* podczas opróżniania pamięci podręcznej wątków, aby poinformować funkcję `flush_thread_cache()` (również w pliku *sql/mysqld.cc*), że wątek zakończył działanie. Dzięki temu `flush_thread_cache()` może obudzić się i sprawdzić, czy są jeszcze jakieś inne wątki do zakończenia. Używana z muteksem `LOCK_thread_count`.
`COND_manager`	Nakazuje wątkowi menedżera (zob. *sql/sql_manager.cc*) obudzić się i przeprowadzić zaplanowany zbiór zadań konserwacyjnych. Obecnie są tylko dwa takie zadania: porządkowanie dzienników Berkeley DB oraz wymuszanie zapisu tabel. Używana z muteksem `LOCK_manager`.
`COND_refresh`	Sygnalizowana, kiedy dane w pamięci podręcznej tabel zostaną zaktualizowane. Używana z muteksem `LOCK_open`.
`COND_thread_count`	Sygnalizowana, kiedy wątek jest tworzony lub niszczony. Używana z muteksem `LOCK_thread_count`.
`COND_thread_cache`	Sygnalizowana w celu obudzenia wątku czekającego w pamięci podręcznej. Używana z muteksem `LOCK_thread_count`.

Oprócz tych zmiennych warunku kilka struktur i klas używa lokalnych warunków do synchronizowania operacji na danej strukturze lub klasie. Istnieje wreszcie kilka globalnych zmiennych warunku o zasięgu plikowym (`static`) w bibliotece *mysys*.

Wywłaszczanie

Termin **wywłaszczanie** oznacza przerywanie wątku w celu przydzielenia czasu procesora innemu zadaniu. Ogólnie rzecz biorąc, MySQL stosuje podejście „odpowiedzialnego obywatela". Wątek wywłaszczający ustawia odpowiednie znaczniki, informując wątek wywłaszczany, że powinien po sobie posprzątać i zakończyć działanie albo ustąpić pola. Wątek wywłaszczany jest odpowiedzialny za wykrycie komunikatu i zastosowanie się do niego.

W większości sytuacji takie podejście się sprawdza, ale istnieje jeden wyjątek. Jeśli wywłaszczany wątek jest zablokowany na operacji wejścia-wyjścia, nie ma okazji, żeby sprawdzić znaczniki komunikatu wywłaszczającego. Aby rozwiązać ten problem, MySQL używa techniki zwanej żargonowo **alarmem wątków**.

Wątek, który ma rozpocząć blokującą się operację wejścia-wyjścia, za pomocą wywołania `thr_alarm()` zgłasza żądanie otrzymania sygnału alarmowego po wyczerpaniu się limitu czasu. Jeśli operacja wejścia-wyjścia zakończy się wcześniej, alarm jest anulowany za pomocą wywołania `end_thr_alarm()`. W większości systemów sygnał alarmu przerywa zablokowaną operację wejścia-wyjścia, dzięki czemu potencjalnie wywłaszczany wątek może sprawdzić znaczniki oraz kod błędu wejścia-wyjścia i rozpocząć odpowiednie działania. Zwykle polega to na wykonaniu czynności porządkowych i wyjściu z pętli wejścia-wyjścia, ewentualnie na próbie ponownego wykonania operacji wejścia-wyjścia.

Zarówno funkcja `thr_alarm()`, jak i `end_thr_alarm()` przyjmują argument w postaci deskryptora alarmu, który przed pierwszym użyciem musi być zainicjalizowany przez wywołanie `init_thr_alarm()`. Procedury alarmu wątków są zaimplementowane w pliku *mysys/thr_alarm.c*.