

Wykład trzynasty

Warstwa operacji blokowych w Linuksie

Blokowe urządzenia wejścia – wyjścia są bardziej skomplikowane w obsłudze niż urządzenia znakowe. W przeciwieństwie do tych ostatnich pozwalają one bowiem na swobodny dostęp do zgromadzonych w nich danych. Oznacza to, że umożliwiają one wyszukiwanie pozycji, gdzie są zgromadzone interesujące nas dane, lub gdzie jest miejsce, w którym te dane chcemy zapisać. Musi więc istnieć jakiś mechanizm pozwalający na dwukierunkową zmianę położenia wskaźnika danych względem bieżącej pozycji. Wszystkie urządzenia blokowe są wyposażone w system plików określonego typu. Najczęściej spotykanymi urządzeniami blokowymi są oczywiście dyski twarde, ale istnieją również inne urządzenia, które zaliczamy do tej kategorii (CD, DVD, pamięci flash). Czas dostępu do tych urządzeń (w szczególności do dysku twardego) jest jednym z najbardziej znaczących czynników mających wpływ na wydajność całego systemu komputerowego. Z tego względu, a także z uwagi na skomplikowanie zagadnienia, w jądrze systemu Linux wyodrębniono osobny podsystem, zajmujący się obsługą takich urządzeń, który nazywa się warstwą blokowych operacji wejścia – wyjścia (*ang. block IO layer*).

Urządzenia blokowe przechowują dane w sektorach, które najczęściej mają wielkość 512 bajtów (choć nie jest to regułą). Sektor jest równocześnie najmniejszą jednostką pamięci urządzenia blokowego, którą można zaadresować. Pojedyncza operacja wejścia – wyjścia może obejmować jeden lub większą liczbę sektorów. Większość systemów operacyjnych nie posługuje się bezpośrednio sektorami, ale łączy je w zazwyczaj większe jednostki zwane blokami. Blok może mieć taki sam rozmiar jak sektor, lub jego rozmiar może być wielokrotnością rozmiaru sektora. W systemie Linux przyjęto, celem uproszczenia kodu jądra, że bloki będą miały wielkość mniejszą lub równą jednej stronie, choć to ograniczenie w przyszłych wersjach systemu może zniknąć. Bloki na dane pochodzące z odczytu lub zawierające dane do zapisu do urządzenia blokowego są umieszczone w pamięci operacyjnej, w buforach. Każdy z takich buforów wyposażony jest w nagłówek, określony strukturą typu *struct buffer_head*, przechowujący dane niezbędne do prawidłowego zarządzania takim buforem. Do tych danych należą między innymi: stan bufora, który jest przechowywany w polu *b_state* tej struktury. Stan ten może być określony jednym lub kilkoma znacznikami należącymi do wyliczenia *bh_state_bits*. Znacznik *BH_Uptodate* oznacza, że bufor zawiera zsynchronizowane z nośnikiem dane, *BH_Dirty* – zawartość bufora została zmodyfikowana i powinna zostać zapisana na nośniku, *BH_Lock* – bufor jest chroniony przed dostępem współbieżnym na czas realizowanej właśnie operacji wejścia – wyjścia, *BH_Req* – bufor jest używany w realizowanym zleceniu, *BH_Update_Lock* – używany do oznaczenia pierwszego bufora ze wszystkich znajdujących się na stronie, który wraz z nimi podlega ochronie przed współbieżnym dostępem na czas realizacji bieżącej operacji wejścia-wyjścia, *BH_Mapped* – bufor jest poprawnym buforem powiązany z blokiem urządzenia, *BH_New* – bufor został przydzielony, ale jeszcze nie był wykorzystywany, *BH_Async_Read* – bufor jest używany w operacji asynchronicznego odczytu, *BH_Async_Write* – bufor jest używany w operacji asynchronicznego zapisu, *BH_Delay* – bufor nie został jeszcze skojarzony z blokiem urządzenia, *BH_Boundary* – bufor związany z blokiem graniczny ciągłego obszaru bloków na nośniku, następny blok nie należy już do tego obszaru, *BH_Write_EIO* – wystąpił błąd podczas zapisu bufora na nośnik, *BH_Unwritten* – zostało przydzielone miejsce na nośniku dla bufora, ale dane z niego nie zostały jeszcze w tym miejscu zapisane, *BH_Quiet* – błędy operacji na buforze nie będą zgłaszane, *BH_Meta* – bufor zawiera metadane, *BH_Prio* – bufor używany w priorytetowej operacji, *BH_Defer_Completion* – zakończenie operacji, w której bufor bierze udział jest opóźniane przy użyciu kolejki prac, jest to operacja asynchroniczna. Wyliczenie *bh_state_bits* zawiera również dodatkowy znacznik *BH_PrivateStart*, który informuje, że kolejne, starsze od niego bity pola *b_state* są wykorzystywane przez sterownik urządzenia blokowego do własnych celów. Kolejne pole tej struktury o nazwie *b_count* jest licznikiem odwołań do bufora. Jego wartość jest zwiększana przy pomocy funkcji *put_bh()*, a zmniejszana przy pomocy *put_bh()*. Obie funkcje są funkcjami *inline*. Licznik odwołań powinien być zwiększany przed wykonaniem każdej z operacji dotyczącej danego bufora, gdyż zapobiega to jego wcześniejszemu zwolnieniu. Pole *b_dev* zawiera wskaźnik na strukturę opisującą urządzenie fizyczne, na którym znajduje się blok skojarzony z buforem, a pole *b_blocknr* zawiera numer tego bloku. Strona, na której znajduje się bufor jest określona wartością pola *b_page*. Adres, od którego zaczyna się obszar bufora na tej stronie jest umieszczony w polu *b_data*. Rozmiar tego bufora jest określony zawartością pola *b_size*. Istnieją również inne pola nagłówka, które nie będą tutaj opisywane.

Nagłówek bufora w wersjach jądra systemu wcześniejszych niż 2.6 przechowywał również informacje dotyczące operacji I/O, w których uczestniczył bufor. Taka sytuacja powodowała niską efektywność takich operacji, gdyż pojedynczy zapis lub odczyt z urządzenia wymagał posłużenia się kilkoma takimi nagłówkami, dodatkowo rozmiar nagłówka był porównywalny z rozmiarem bufora, który opisywał. W nowszych wersjach jądra postanowiono więc „odchudzić” nagłówek bufora i stworzyć nową strukturę, o nazwie *bio*, która osobno przechowuje dane związane z operacjami wejścia – wyjścia. Reprezentuje ona takie operacje w trakcie ich trwania za pomocą listy *segmentów*¹. Segment w tym przypadku jest definiowany jako ciągły fragment bufora. Bufory, których segmenty są zgromadzone na liście nie muszą tworzyć ciągłego obszaru w pamięci operacyjnej. Dodatkowo, dzięki tej strukturze można realizować kilka operacji wejścia – wyjścia na jednym buforze współbieżnie. Najważniejsze pola struktury *bio* to *bi_io_vec*, *bi_vcnt* oraz *bi_iter*, które samo jest strukturą zawierającą pole *bi_idx*. Pierwsze z nich zawiera adres tablicy struktur *bio_vec*, która jest wykorzystywana jako lista poszczególnych segmentów używanych w danej operacji wejścia – wyjścia. Każdy z elementów tej tablicy jest opisany trójką: *<bv_page, bv_offset, bv_len>*, (strona, przemieszczenie, długość). Cała tablica opisuje więc sumaryczną przestrzeń w pamięci, stworzoną z segmentów buforów i wyznaczoną dla operacji. Drugie pole struktury *bio* określa ile elementów z opisywanej tablicy bierze udział w operacji. Bieżącą pozycję w tej tablicy reprezentuje pole *bi_idx*, którego zawartość jest aktualizowana na bieżąco. Użycie tego pola pozwala na podział struktury *bio*, co ma znaczenie w przypadku takich sterowników, jak sterowniki macierzy RAID. Podział polega na kilkukrotnym skopiowaniu tej struktury i ustawieniu dla każdej z tych kopii innej wartości pola indeksującego w polu *bi_iter*. Podobnie jak nagłówek bufora, również struktura *bio* posiada licznik odwołań. Jego wartość jest zwiększana przy pomocy funkcji *bio_get()*, a zmniejszana przy pomocy *bio_put()*. Pole *bi_private* może być wykorzystywane dla danych twórcy struktury *bio*. Zastosowanie struktury *bio* przyniosło następujące korzyści:

- blokowe operacje wejścia – wyjścia mogą w prosty sposób korzystać z wysokiej pamięci, gdyż struktura *bio* posługuje się strukturami typu *struct page*,
- struktura *bio* może reprezentować zarówno zwykłe operacje I/O, jak i również operacje bezpośrednie, które nie korzystają z buforów jądra,
- ułatwiona jest realizacja operacji wejścia – wyjścia, w których dane pochodzą z wielu rozłącznych stron pamięci (tzw. operacje z rozproszonym źródłem),
- obsługa takiej struktury jest mniej skomplikowana niż obsługa nagłówków buforów.

Większość sterowników urządzeń blokowych utrzymuje struktury przechowujące zlecenia operacji wejścia – wyjścia przeznaczone dla obsługiwanych przez nie urządzeń. Te struktury są nazywane kolejkami zleceń lub kolejkami zadań. Są one reprezentowane strukturą *request_queue_t* i zawierają dwukierunkową listę zleceń i skojarzonych z nimi informacji sterujących. Każdy z elementów tych kolejek jest opisany strukturą *struct request* i reprezentuje pojedyncze zlecenie. Jeśli kolejka nie jest pusta, to pierwsze zlecenie znajdujące się na niej jest przekazywane przez sterownik do urządzenia, które je realizuje. Każde zlecenie może zawierać wiele struktur *bio*, które opisują segmenty zaangażowane w daną operację.

Za szeregowanie zleceń w opisywanej kolejce odpowiedzialny jest planista operacji wejścia – wyjścia (*ang. I/O scheduler*). Jego zadaniem jest zminimalizowanie liczby przestawień mechanizmu służącego do odczytywania i zapisywania danych w urządzeniu blokowym (np. głowicy w dysku twardym), co pozwala na osiągnięcie maksymalnej średniej przepustowości oraz unikanie zagłodzenia zadań. Planista dokonuje tego wykonując operacje skalania i sortowania². Kiedy nowe zadanie trafia do kolejki, wówczas planista stara się je scalić z zadaniami, które dotyczą przyległych sektorów. Jeśli takowych zadań nie ma, to planista stara się umieścić je pośród zadań, które dotyczą sektorów leżących w pobliżu, dzięki czemu nie będzie konieczna częsta zmiana kierunku ruchu głowicy³. W jądrach serii 2.6 jest używany jeden z czterech⁴ algorytmów szeregowania zadań. Dwa z nich są wzorowane na algorytmie, który był wykorzystywany w wersji 2.4, więc on jako pierwszy zostanie opisany. Planista I/O w jądrach wersji 2.4 działał w oparciu o algorytm nazwany Windą Linusa (*ang. Linus Elevator*). Algorytm ten stosuje skalanie obustronne. Oznacza to, że nowe zlecenie może być umieszczone przed istniejącymi już zleceniami lub za, jeśli tylko będą one dotyczyły spójnego obszaru sektorów. Pierwszy rodzaj skalania nazywany jest skalaniem frontowym, drugi skalaniem tylnym. Zazwyczaj ten drugi rodzaj występuje częściej. Jeśli nowego zlecenia nie da się scalić z innymi, które są obecne w kolejce, to następuje etap sortowania. W tym etapie planista stara się znaleźć miejsce dla kolejki dla nowego zlecenia, takie że otaczające je inne zlecenia będą dotyczyły sektorów znajdujących się w pobliżu. Jeśli nie znajdzie takiego miejsca, to umieszcza dane zlecenie na końcu kolejki. Może tak postąpić jeszcze w jednym przypadku, kiedy podczas przeszukiwania kolejki znajdzie przeterminowane zlecenie. Takie postępowanie ma na celu wyeliminowanie głodzenia zadań, ale niestety nie jest skuteczne i algorytm Windy Linusa może doprowadzić do tego zjawiska.

W jądrze 2.6 postanowiono więc go zastąpić czterema innymi rozwiązaniami. Pierwszym z nich jest planista terminowy (*ang. deadline I/O scheduler*). Zapobiega on głodzeniu zadań oraz faworyzuje operacje odczytu przed operacjami zapisu. Okazuje się bowiem, że opóźnienia odczytu mają większy wpływ na wydajność systemu niż

1 Które nie mają nic wspólnego z segmentami, którymi posługują się procesory Intel'a i pokrewne do adresowania pamięci operacyjnej.

2 W tym wypadku chodzi tu o dwie odrębne operacje, a nie o operację sortowania przez skalanie.

3 Podobnie jak w metodzie LOOK omawianej w poprzednim semestrze.

4 Liczba ta ulega co jakiś czas zmianie. W wersjach jądra 3.0 i nowszych są dostępne tylko trzy takie algorytmy.

opóźnienia operacji zapisu. Planista terminowy stosuje cztery struktury danych: główną kolejkę zleceń, kolejkę zleceń odczytu, kolejkę zleceń zapisu i kolejkę rozdziału. Mechanizm ten przydziela każdemu zleceniu termin realizacji. Domyślnie wynosi on 500 ms dla operacji odczytu i 5 s dla operacji zapisu. Nowe zlecenia są wstawiane równocześnie do kolejki głównej, gdzie realizowane są operacje scalania i sortowania, oraz w zależności od rodzaju zlecenia do kolejki zapisu lub odczytu⁵. Te dwie ostatnie kolejki są kolejkami FIFO. Planista terminowy pracuje w dwóch trybach: w trybie normalnym pobiera pierwsze żądanie z kolejki głównej i wstawia je do kolejki rozdziału, z której trafi ono później bezpośrednio do urządzenia. Planista przelacza się w drugi tryb jeśli zbliża się termin realizacji operacji z kolejki FIFO. Wówczas do kolejki rozdziału trafia żądanie z którejś z tych kolejek.

Drugim planistą stosowanym w jądrach serii 2.6 jest planista przewidujący⁶. W przeciwieństwie do planisty terminowego pozwala on uniknąć sytuacji, kiedy ciągi operacji zapisu są przerywane przez pojedyncze żądania operacji odczytu. W działaniu jest on bardzo podobny do planisty terminowego, ale stosuje tzw. heurystykę przewidziania. W momencie przekazania zlecenia odczytu do kolejki rozdziału planista nie wraca od razu do realizacji kolejnych zleceń, lecz wstrzymuje swe działanie na 6 ms⁷. Jeśli po tym czasie aplikacja wygeneruje żądanie odczytu dotyczące obszaru leżącego w pobliżu tego, którego dotyczyło poprzednie żądanie, to jest ono realizowane natychmiast. Aby ten czas oczekiwania nie był czasem straconym, sytuacje takie, jak opisywana powyżej powinny mieć często miejsce. Planista przewidujący stara się określić możliwość wystąpienia takiej sytuacji prowadząc statystykę działań aplikacji i stosując heurystyki. Planistę przewidującego usunięto z jądra systemu w wersji 2.6.33.

Planista przewidujący był domyślnym planistą wejścia-wyjścia do czasu wydania jądra w wersji 2.6.18 (choć w niektórych dystrybucjach przestał nim być już wcześniej). Zastąpił go planista CFQ (ang. *Complete Fair Queuing*), który po raz pierwszy pojawił się w wersji 2.6.6 jądra. Jego działanie można krótko scharakteryzować jako połączenie planowania z użyciem kolejek wielopoziomowych, algorytmu rotacyjnego i przewidziania. Ten planista wprowadza również nową cechę procesów użytkownika: priorytet wejścia-wyjścia. Każdemu z procesów, który wykonuje operacje blokowe przydzielana jest dynamicznie kolejka na zlecenia synchronicznych operacji wejścia-wyjścia. Zlecenia operacji asynchronicznych trafiają do wspólnych kolejek, których zazwyczaj jest mniej niż kolejek żądań operacji synchronicznych. Planista przegląda kolejki procesów poczynając od kolejek o najwyższym priorytecie, a skończywszy na kolejkach o najniższym. Z każdej z tych kolejek zdejmuje tyle zleceń, ile może zrealizować w ciągu określonego dla kolejki przedziału czasu. Kwant czasu i liczba zleceń dla kolejki są zdeterminowane przez priorytet wejścia-wyjścia jej procesu. Dla tych kolejek planista CFQ realizuje również opcję przewidziania, czyli po opróżnieniu danej kolejki zatrzymuje się na krótki czas, sprawdzając, czy nie pojawią się w niej nowe zlecenia. Jeśli tak się stanie, to są one realizowane natychmiast. Po obsłużeniu kolejek operacji synchronicznych planista przechodzi do kolejek operacji asynchronicznych, ale w ich przypadku nie stosuje opcji przewidziania. Wszystkie wymienione kolejki są zaimplementowane przy użyciu drzew czerwno-czarnych.

Czwarty algorytm szeregowania żądań I/O jest bardzo prosty w działaniu – realizuje wyłącznie operację scalania. Ten algorytm nosi nazwę *noop*.

Domyślnego planistę operacji wejścia-wyjścia można wybrać na etapie kompilacji, spośród czterech (trzech od wersji jądra 2.6.33) opisanych wyżej. Można także zmienić planistę I/O dla konkretnego urządzenia w czasie wykonania dokonując odpowiednich wpisów do plików w katalogu `/sys`. W przypadku urządzeń blokowych o prawdziwie swobodnym dostępie (np.: pamięci flash) najlepszym planistą jest *noop*.

Ostatnia większa zmiana w warstwie operacji blokowych nastąpiła w wersji 3.13 jądra. Podyktowana ona była upowszechnieniem urządzeń SSD, które w kwestii wydajności znacznie przewyższają tradycyjne urządzenia pamięci masowej, dla których ta warstwa oryginalnie była zaprojektowana (setki operacji na sekundę wobec milionów operacji na sekundę). Niezmodyfikowana warstwa operacji blokowych stanowiła wąskie gardło, szczególnie dla urządzeń SSD zainstalowanych w wieloprocesorowych systemach. Aby pozbyć się tego problemu wprowadzono do niej trzeci (obok kolejki żądań i braku kolejki żądań) tryb pracy. Jest on na tyle różny od pozostałych, że programiści jądra określają go nową warstwą operacji blokowych. Polega on na tym, że każdy procesor (lub węzeł w przypadku komputerów o architekturze NUMA) otrzymuje własną, lokalną programową kolejkę żądań (ang. *software request queue*), która nie wymaga synchronizacji w postaci rygła pętlowego. Kolejka ta nie podlega także żadnemu z wcześniej opisanych mechanizmów szeregowania, a jedynie realizuje operacje scalania przyległych żądań pochodzących od procesora, który obsługuje. Z tych kolejek żądania trafiają do sterownika urządzenia SSD, który może być wyposażony w kilka⁸ sprzętowych kolejek żądań (ang. *hardware request queue*). Liczba takich kolejek jest określana przez sterownik podczas jego inicjacji i wynika z możliwości równoległego obsługiwanie przez urządzenie SSD żądań wejścia-wyjścia. Żądania zdejmuwane z tych kolejek są przekazywane do pamięci SSD. Prawdopodobnie, kiedy urządzenia SSD wyeliminują z użycia tradycyjne dyski talerzowe, to tryb z pojedynczą kolejką żądań zostanie zastąpiony przez tryb z wieloma kolejkami.

Mimo, że pierwotnie programiści jądra zakładali, że w programowych kolejkach wejścia-wyjścia nie będzie konieczne szeregowanie, to okazało się, że może ono poprawić wydajność wolniejszych urządzeń SSD oraz może umożliwić obsługę priorytetów żądań pochodzących z różnych aplikacji. Wstępnie w wersji 4.11 dostosowano planistę terminowego do obsługi takich kolejek. W wersji 4.12 pojawiło się dwóch planistów przeznaczonych dla programowych kolejek żądań. Są to BFQ i Kyber. Ten ostatni jest prostszy w działaniu oraz budowie, więc zostanie opisany jako pierwszy. Celem tego planisty jest osiągnięcie jak najmniejszego opóźnienia realizacji operacji wejścia-wyjścia. W tym celu dzieli on każdą programową kolejkę żądań na dwie. Jedna jest przeznaczona dla operacji synchronicznych, druga dla asynchronicznych. Na realizację operacji z pierwszej kolejki domyślnie przeznaczono są 2 ms, a na realizację operacji asynchronicznych 10 ms. Kyber wprowadza żądania z tych kolejek do kolejek sprzętowych w taki sposób, aby te były możliwie jak najkrótsze (zawierały jak najmniej zleceń). To zapewnia krótki czas ich realizacji. Maksymalna liczb żądań w kolejkę sprzętowej jest wyznaczana na podstawie czasu realizacji poprzednich operacji I/O.

Planista BFQ miał być stosowany w trybie z jedną kolejką, ale ostatecznie trafił do nowej warstwy operacji blokowych. Jego działanie jest podobne do planisty CFQ, ale wykorzystuje również rozwiązania znane z planisty procesora CFS. BFQ przypisuje każdemu procesowi liczbę sektorów („budżet wejścia-wyjścia”), które może on przesłać lub pobrać z urządzenia, kiedy planista wyznaczy go do realizacji operacji wejścia-wyjścia. Obliczenia tego budżetu są skomplikowane, ale bazują na dwóch danych wejściowych: wadze wejścia-wyjścia procesu oraz jego zachowaniu podczas wykonywania poprzednich operacji wejścia-wyjścia. Wielkość budżetu nie może przekraczać ustalonego w systemie maksimum. Budżet procesu oznacza jego udział w przepustowości (ang. *bandwidth*) urządzenia wejścia-wyjścia, która wyznaczana jest heurystycznie. Żądania procesów o mniejszym budżecie są szeregowane jako pierwsze, przed żadaniami procesów o większych budżetach. Każdy proces otrzymuje porcję czasu, w ramach której musi zrealizować swój budżet. Jeśli proces wykona to zadanie przed upływem zadanego czasu, a ostatnią wykonaną przez niego operacją wejścia-wyjścia będzie operacja synchroniczna, to planista wstrzyma swoje działanie, tak jak planiści CFQ i przewidujący. Aby usprawnić szeregowanie wykonywane przez tego planistę stosowany jest szereg heurystyk, które nie będą na tym wykładzie opisywane. Można się z nimi zapoznać czytając artykuł Jonathana Corbeta "The BFQ I/O scheduler" na stronie: <https://lwn.net/Articles/601799/>.

5 Dokładniej: do tych kolejek zapisywany jest wskaźnik na to zlecenie.

6 Można go też „ładnie” nazwać planistą antycypującym.

7 Czas ten można konfigurować.

8 Co najmniej jedną, ale najczęściej więcej niż jedną.