

Nowe systemy plików dla Linuksa

Zwiększenie skalowalności

Autor: [Łukasz Spufiński](#)

W poprzednim artykule (PCkurier 5/2001) opisaliśmy architekturę kolejnych warstw obsługi plików w Linuksie oraz przebieg przykładowej operacji otwarcia pliku. Teraz omówimy możliwe scenariusze usunięcia niektórych problemów, które może powodować podstawowy system plików Linuksa. Złożoność obliczeniowa większości zaimplementowanych tam algorytmów ma postać liniową; ograniczenia narzuca również 32-bitowy charakter znacznej części struktur. Zmniejsza to zarówno wydajność, jak i skalowalność systemu.

Standardowy system plików Uniksa - UFS (Unix File System) oraz wywodzący się z niego Ext2 były projektowane dla pamięci masowych o pojemności nieprzekraczającej kilku gigabajtów. Pierwszy system plików Linuksa - Minix File System obsługiwał partycje o rozmiarze do 64 MB i nazwach plików o długości maks. 14 znaków. Jego następca - Ext - pozwalał pracować na przestrzeniach o zawrotnej (w 1992 r.) wielkości 2 GB. Dopiero zaprojektowany na bazie BSD FFS system plików Ext2 na dłuższy czas zaspokoił oczekiwania użytkowników. Wraz z rozwojem technologii również i ten system plików przestał wystarczać.

Firmy pozyskują, przechowują, przetwarzają i wytwarzają coraz większe ilości danych. Dostęp do danych powinien być możliwie najszybszy, a usuwanie ewentualnych usterek jak najkrótsze. Przy pojemnościach rzędu kilku terabajtów (to typowa wielkość baz danych nawet w polskich przedsiębiorstwach) trudno sobie wyobrazić każdorazową naprawę systemu plików przez jego pełne skanowanie. Dzieje się tak dlatego, że w klasycznych systemach o liniowej architekturze trwałoby to niemiłosiernie długo, a efekt wcale nie musiałby być zadowalający.

Nauka (nie) poszła w las...

Problemy wydajnościowe i spowodowały, że w ramach prac nad Linuksem zostało uruchomionych kilka projektów nowoczesnych systemów plików.

**System
plików
Projektami
kieruje
Maks. rozmiar
systemu plików (TB)
Rozmiar bloku (B)
Maks.rozmiar
pliku (TB)**

XFS
[SGI](#)
18 mld
512-65536
9 mld

JFS
[IBM \(Steve Best\)](#)
32 mln
512/1024/2048/4096
4 mln

ReiserFS
[Hans Reiser](#)
16
1024-65536
16

Aby lepiej zrozumieć zasady działania nowych systemów plików, trzeba przyjrzeć się dotychczas stosowanym architekturom i wynikającym z nich problemom oraz zastanowić się nad możliwymi rozwiązaniami.



W systemach UFS i Ext2 analiza dostępności poszczególnych bloków odbywa się sekwencyjnie. Wolne bloki można znaleźć, przeglądając po kolei zawartości map bitowych bloków z kolejnych grup bloków. W skrajnym przypadku może to oznaczać konieczność przejrzenia 131 071 grup bloków po 32 768 pozycji każda ($16\text{ TB} = 131\,072 \times 32\,768 \times 4\text{ kB} = 131\,072 \times 128\text{ MB}$). Proponowanym rozwiązaniem tego problemu jest zastąpienie sztywnej struktury map bitowych indeksami (położenia oraz liczby wolnych bloków) w formie B⁺-drzew, które są często stosowane w systemach baz danych.

Ponieważ indeksowanie obejmuje również rozmiar całych ciągłych sekwencji wolnych bloków (ekstentów, od ang. extents), korzyść jest podwójna - poza szybszym znalezieniem jakiegokolwiek pierwszego wolnego bloku, można stosunkowo szybko znaleźć wolny blok, który byłby jednocześnie najlepszym ze względu na przewidywany przyrost wielkości pliku. W przypadku dużej ilości dużych plików (np. bazy danych) ekstenty pozwalają zaoszczędzić miejsce na dysku. Zamiast zapisywania informacji o każdym z małych bloków, można zapisywać niewiele obszerniejsze (w porównaniu do informacji o bloku) informacje o znacznie większych jednostkach alokacji.

Testy zostały przeprowadzone na platformie serwerowej Intel ISP2150 wyposażonej w dwa procesory Intel Pentium III 800 MHz 256 kB cache i pamięć RAM 1 GB. Podsystem dyskowy oparto na kontrolerze SCSI Adaptec AIC-7896 Ultra-2 Wide i dyskach Quantum Atlas 10K 9SCA.

ISP2150 to serwer zbudowany na płycie IntelR L440GX+ "Lancewood" (Slot 1, FSB 100, maks. 2 GB SDRAM) ze zintegrowanym układem graficznym, sterownikiem sieciowym Ethernet 10/100 (IntelRPRO/100+ Fast Ethernet Server Controller 82559), dwukanałowym kontrolerem SCSI (wspomniany Adaptec AIC-7896), standardowymi portami szeregowymi i równoległym, portem myszki i klawiatury oraz kontrolerem i portami USB.

Platforma ma wbudowany układ kontrolowania i zarządzania serwerem, pozwalający na monitorowanie stanu procesorów i pamięci, temperatury w wielu punktach płyty głównej, pracy wentylatorów oraz napięć zasilania. Układ Watchdog umożliwia zdalny restart, wyłączenie lub włączenie serwera poprzez dedykowany port szeregowy (funkcjonalność zabezpieczana hasłem). Całość jest obsługiwana przez oprogramowanie Intel Server Control (support.intel.com/...).

Konsola BIOS platformy oraz konsola systemu operacyjnego może być sprzętowo przekierowana na port szeregowy, co w systemie operacyjnym Unix lub Linux pozwala na przeprowadzenie wstępnej konfiguracji sprzętowej i instalacji systemu bez konieczności podłączania monitora i klawiatury.

Obudowa ISP 2150 została zaprojektowana pod kątem firm typu ISP - montowana w standardowym racku 19", głębokość obudowy 24", wysokość 2 U. Na dyski hot-swap przeznaczone są cztery prowadnice ze złączem SCA. Backplane jest obsługiwany przez jeden kanał Ultra2 LVD zainstalowanego na płycie kontrolera SCSI.

www.intel.com/...

Alokacja i-węzłów to problem zbliżony do alokacji bloków. Oczywiście dotyczy on tylko tych systemów plików, które posługują się klasycznym rodzajem i-węzła, czyli wydzieloną fizycznie strukturą w systemie plików. Struktura grupy bloków narzuca ograniczenia dotyczące zarówno bloków (liniowa mapa bitowa bloków; stała, identyczna liczba bloków we wszystkich grupach; stały, taki sam rozmiar bloku we wszystkich grupach), jak i i-węzłów (liniowa mapa bitowa i-węzłów; stała, identyczna liczba i-węzłów we wszystkich grupach). Problem jest rozwiązywany podobnie jak dla bloków - przez zastąpienie sztywnych struktur B⁺-drzewami.

Problemem podczas pracy z Ext2 bywa również obsługa dużej ilości plików w ramach tego samego katalogu. W Ext2 katalog jest zaimplementowany w formie jednokierunkowej listy, a więc operacja wyszukiwania jakiegokolwiek pliku ma charakter liniowy. Przy dużych katalogach problem narasta. Tu także klasycznym rozwiązaniem są indeksy w postaci B⁺-drzew.

Poza technikami rozwiązującymi dotychczasowe problemy wydajnościowe i pojemnościowe nowe systemy plików implementują też szereg innych, interesujących mechanizmów. Jedną z opcji jest śledzenie wykorzystania przestrzeni dyskowej przez plik i alokacja tylko niezbędnych obszarów. Przykładowo ciągły zapis odpowiednio dużej (opłacalnej w odniesieniu do pojemności) ilości tych samych danych (zer, jedynek) do pliku może być zastąpiony zapisem specjalnego "skrót". Co prawda technika ta przypomina kompresję pliku, ale ma od niej znacznie mniejszą złożoność obliczeniową. W praktyce można nawet pominąć taki skrót, gdy założymy, że plik skracamy tylko o ciągi zer, a kolejne pozycje niezerowe mają wynikać ze zdefiniowanych pozostałych fragmentów pliku.

Taki scenariusz dotyczy nowych systemów plików dla Linuksa. Kolejne fragmenty plików są tam definiowane przez ekstenty (lub bloki - w przypadku nieco spóźnionego pod tym względem systemu ReiserFS), które stanowią nie tylko jednostki alokacji o zmiennej wielkości, ale ich deskryptory (zapisane w B⁺-drzewach) dają także informację o tym, jakiego fragmentu pliku dotyczą. Po prostu zerowe zapisy (ekstenty składające się z samych zer) nie są przenoszone na fizyczny nośnik i nie marnują zasobów dyskowych i czasu procesora.

System plików

Wykaz wolnych plików
Wykaz bloków zajętych przez plik
Wykaz i-węzłów
Struktura katalogów

XFS

B⁺-drzewo ekstentów

B⁺-drzewo ekstentów

B⁺-drzewo

B⁺-drzewo

JFS

podział mapy bitowej bloków na mniejsze fragmenty i indeksowanie drzewem binarnym (wg położenia)

B⁺-drzewo ekstentów

B⁺-drzewo z liśćmi o stałym rozmiarze 32 i-węzłów

B⁺-drzewo

ReiserFS

mapa bitowa

jedno B⁺-drzewo obejmujące cały system plików (adresowanie pojedynczymi blokami zamiast ekstentów)

jedno B⁺-drzewo obejmujące cały system plików

jedno B⁺-drzewo obejmujące cały system plików

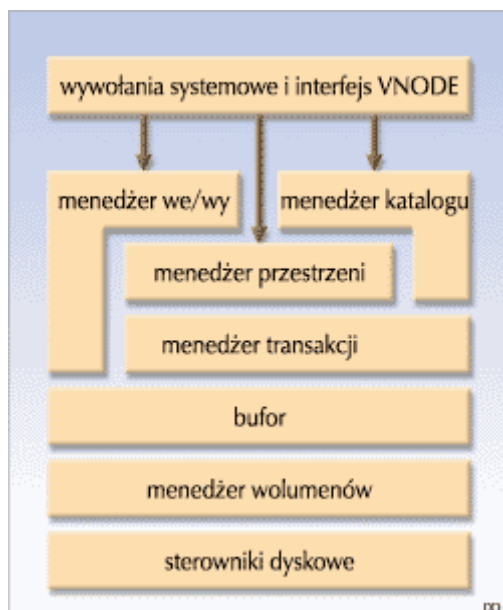
Projektem najbardziej skalowalnego systemu plików dla Linuksa jest próba adaptacji systemu XFS. Twórcą XFS jest znany producent superkomputerów oraz wydajnych, graficznych stacji roboczych - firma Silicon Graphics.

Prace nad XFS rozpoczęto w 1993 roku. Główną przyczyną był wzrost wydajności i skalowalności pamięci masowych, z czym nie radził sobie system plików EFS dotychczas stosowany w systemie operacyjnym IRIX (odmiana Uniksa dla maszyn SGI). System EFS wywodził się z BSD FFS, obsługiwał partycje o rozmiarze do 8 GB oraz miał klasyczne ograniczenie 32-bitowego systemu plików - maksymalny rozmiar pliku wynosił 2 GB.

Pierwsza wersja nowego systemu plików dla IRIX-a ukazała się w grudniu 1994 roku. W roku 1996 XFS stał się domyślnym systemem, wypierając poprzednika. Pierwotnie XFS był projektowany pod kątem systemu operacyjnego IRIX. Przeniesienie go na inną platformę wymagało poniesienia znacznych nakładów pracy, a zwłaszcza stworzenia nowych styków systemu plików z resztą systemu operacyjnego. Architektura XFS stawia specyficzne wymagania dla warstwy wirtualnego systemu plików oraz warstwy buforów dyskowych.

W projekcie zdecydowano się nie tworzyć nowych styków, lecz dodatkowy kod pośredniczący, który mapowałby wywołania funkcji zgodnych z Linuksem na wywołania zgodne z IRIX-em. Pierwsza z warstw - linvfs - odpowiada za współpracę z linuksowym VFS, druga - pagebuf - za styk z buforem blokowym. Tego typu podejście ma swoje wady i zalety. Wadą są dodatkowe opóźnienia w porównaniu z implementacją dla "rodzimego" systemu. Zaletą - wspólna linia rozwojowa samego XFS dla obydwu platform.

Podstawowa architektura



Architektura XFS

XFS ma architekturę modułową (patrz schemat obok). Najważniejszym modułem jest menedżer przestrzeni (Space Manager), którego głównym zadaniem jest obsługa wszystkich obiektów w systemie plików, a zwłaszcza zarządzanie i-węzłami oraz ekstentami. Modułami pośredniczącymi pomiędzy VFS IRIX-a a menedżerem przestrzeni są menedżer wejścia/wyjścia (I/O Manager) oraz menedżer katalogów (Directory Manager).

Pierwszy z nich zajmuje się całościową obsługą plików, a drugi - jak sama nazwa wskazuje - katalogów. Menedżer transakcji zarządza aktualizacją danych. Odpowiada również za operacje księgowania (journaling), które pozwalają na szybką naprawę systemu po awarii. Księgowanie polega na transakcyjności zmian w systemie plików oraz ich logowaniu. W XFS obejmuje ono tylko metadane, takie jak drzewa i-węzłów, i-węzły, zawartość katalogów, drzewa ekstentów zawartości plików, drzewa wolnych ekstentów, bloki nagłówkowe grup alokacji oraz superbloki.

Podobnie jak i inne systemy plików, XFS operuje na blokach o stałym rozmiarze. Rozmiar ten możemy wybrać przy tworzeniu systemu plików, a zakres dopuszczalnych wartości mieści się w przedziale od 0,5 kB do 64 kB. Poza tym XFS dzieli partycję na kilka jeszcze większych części. Pierwsze z nich to tzw. grupy alokacji (allocation groups), czyli porcje przestrzeni o rozmiarze od 16 MB do 4 GB przeznaczone na właściwy system plików. Grupy alokacji ułatwiają wielowątkowe zarządzanie systemem plików. Oczywiście przy tworzeniu XFS-a należy wziąć pod uwagę ograniczenia narzucone przez maksymalny rozmiar grupy alokacji oraz liczbę procesorów w naszym systemie. Większa liczba procesorów zwiększa wydajność XFS, ale jednocześnie większa liczba grup alokacji przy mniejszej liczbie procesorów zmniejsza wydajność systemu plików.

Na początku każdej grupy alokacji znajduje się superblok, czyli - tak jak w Ext2 - [blok opisujący cały system plików](#). Za superblokiem umiejscowiony jest nagłówek grupy alokacji. W jego skład wchodzi trzy struktury:

[Drzewo wolnych ekstentów](#)

W ramach AG (grup alokacji) dostępna wolna przestrzeń całego systemu plików jest indeksowana w formie pary B⁺-drzew. Pierwsze drzewo to indeks położenia wolnych ekstentów, drugie - ich rozmiarów. Złożoność obliczeniowa operacji wyszukiwania w drzewie B⁺ jest rzędu O(log n), a więc znacznie mniejsza od złożoności przeszukiwania map bitowych, która w skrajnym wariantcie (jak np. w Ext2) może mieć postać O(n), gdzie n - liczba jednostek alokacji (i-węzłów, bloków czy ekstentów).

[Drzewo i-węzłów](#)

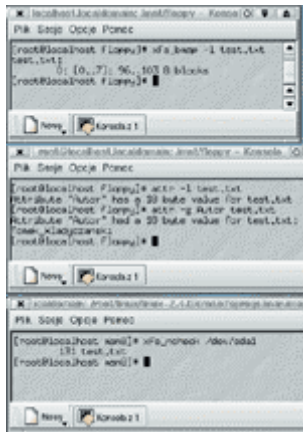
W grupie alokacji, tak jak wolne ciągłości, i-węzły są zapamiętywane w specjalnych B⁺-drzewach. i-węzły XFS nie różnią się wiele od klasycznych uniksowych i-węzłów. Również mają swój niepowtarzalny numer, według którego są indeksowane w B⁺drzewach.

Wykaz bloków zajmowanych przez drzewo wolnych ekstentów

[Struktura pomocnicza przyspieszająca znajdowanie wolnej przestrzeni.](#)

Rejestracja transakcji





Narzędziownia XFS - tworzenie systemu plików (mkfs.xfs) i wydruk logów (xfs_logprint)
 Narzędziownia XFS - mapa ekstentów (xfs_bmap), dodatkowe atrybuty (attr) oraz sprawdzanie i-węzła (xfs_ncheck) pliku

Tuż za grupami alokacji znajduje się obszar logów, czyli informacji o ostatnio wykonywanych transakcjach w ramach systemu plików. XFS rejestruje wszystkie zmiany dokonywane na metadanych, tzn. w superblokach, nagłówkach grup alokacji, drzewach wolnych ekstentów, drzewach i-węzłów, w samych i-węzłach oraz zawartościach katalogów. Operacje tego typu są opakowywane w transakcje, będące ciągiem zmian w blokach systemu plików. Dzięki temu późniejsza naprawa uszkodzonego systemu plików nie polega na naprawie poszczególnych, złożonych struktur, lecz na dokończeniu ostatnio zaplanowanych zmian w blokach.

Domyślnie obsługa transakcji w XFS jest asynchroniczna, tzn. dostęp do modyfikowanych bloków systemu plików jest blokowany dla innych żądań dopiero na etapie zatwierdzania całej transakcji. Wyjątkiem są zasoby udostępniane przez sieciowy system plików NFS, dla których obowiązuje synchroniczny tryb postępowania (blokowanie dostępu przy każdej nowej modyfikacji metadanych w ramach transakcji). XFS umożliwia tworzenie logów na innym urządzeniu niż logowany system plików, co zwłaszcza przy synchronicznym trybie pracy może istotnie zwiększyć wydajność.

Każdy z logów XFS ma swój **nagłówek**. Nazwy plików i katalogów mogą mieć długość do 255 znaków i są indeksowane poprzez funkcję skrótu, która generuje odpowiadające tym nazwom 32-bitowe klucze.

W ramach XFS alokacja ekstentów oraz zapis pliku nie odbywa się od razu. Najpierw na dysku rezerwowane są bloki na możliwie największy ekstent. Następnie w pamięci tworzony jest specjalny bufor ekstentu, który w przypadku implementacji linuksowej jest zbiorem buforów blokowych. Dopiero gdy zaistnieje potrzeba zwolnienia pamięci lub zostanie przekroczony rozmiar zarezerwowanych bloków, ekstent jest kopiowany na fizyczny system plików, a bufor usuwany. Niewątpliwą zaletą tego typu buforów plikowych/ekstentów jest zmniejszenie fragmentacji zewnętrznej systemu plików, liczby operacji na metadanych oraz szybka obsługa plików tymczasowych (w skrajnym przypadku wcale nie muszą być zapisywane na urządzeniu blokowym).

XFS daje aplikacjom również możliwość pomijania buforów (w tym blokowych). Inną interesującą właściwością jest bardziej elastyczny mechanizm zarządzania równoczesnym dostępem do tego samego pliku. W tradycyjnych uniksowych systemach plików (np. Ext2) obowiązuje zasada - jedno zadanie zapisuje, wiele zadań odczytuje. W XFS zaimplementowano dodatkowe funkcje dla programów użytkownika, które dzięki mechanizmowi kolejkwania żądań zapisu pozbawione są tego ograniczenia.

Z powrotem na drzewa

Podstawowe obiekty systemu plików - wolne ekstenty oraz i-węzły - są zorganizowane w B⁺-drzewa. Struktura takich drzew jest zdefiniowana w pliku źródłowym [fs/xfs/xfs_btree.h](#). Do [pojedynczego i-węzła](#), odnoszą się liście w drzewach i-węzłów.

B⁺-drzewa ekstentów zdefiniowane są w pliku źródłowym [fs/xfs/xfs_bmap_btree.h](#). Cechy ekstentu, czyli elementarnej składowej zawartości pliku, to: 54-bitowy numer bloku w ramach pliku, 52-bitowe położenie (numer bloku) w ramach systemu plików oraz 21-bitowy rozmiar (wyrażony liczbą bloków). W XFS zawartość katalogów to także [B⁺-drzewa](#).

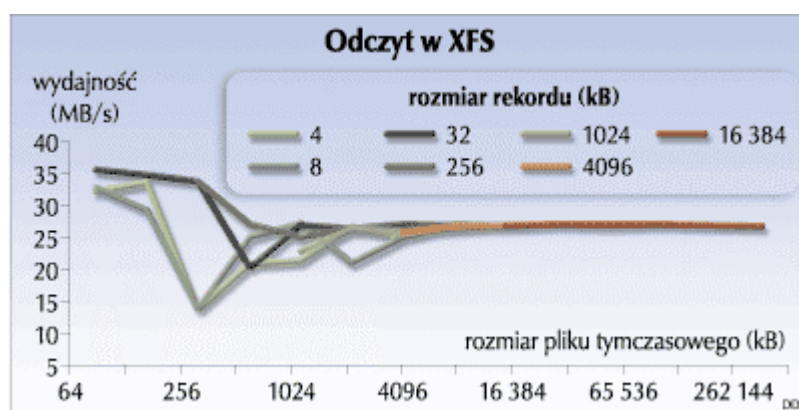
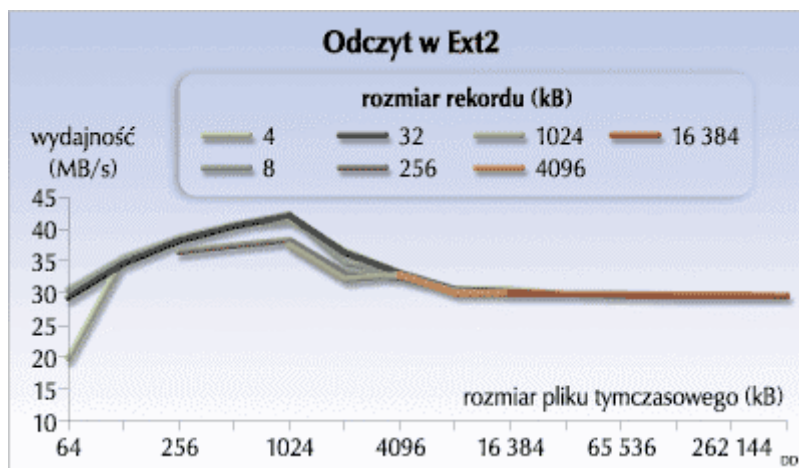
Testy IOZone

W celu porównania wydajności obydwu systemów plików - podstawowego (Ext2) oraz XFS - wykonałem kilka

testów programem [IOZone](#). Testy polegały na jednorazowych odczytach oraz zapisach do plików testowych o wielkości od 64 kB do 512 MB. Rekordy miały rozmiar od 4 kB (rozmiar pojedynczego bloku w systemie plików) do 16 MB. Udział buforów dyskowych został zminimalizowany przez odmontowywanie i ponowne montowanie systemu plików przed wykonaniem każdego z testów.

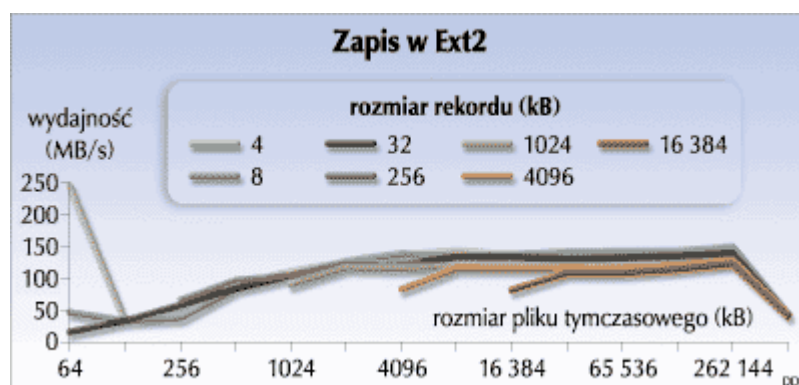
Test odczytu

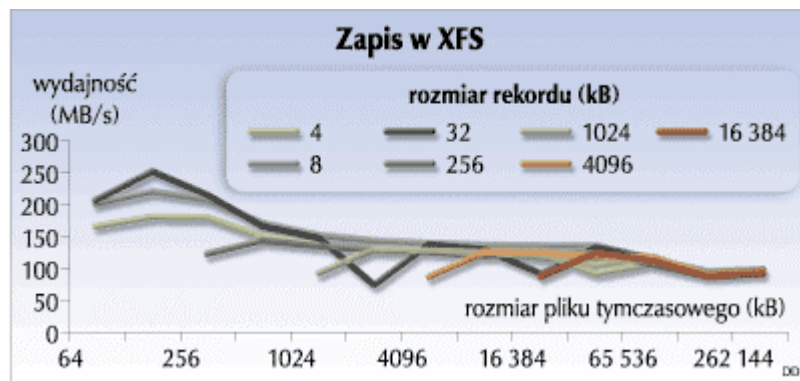
polegał na jednorazowym, sekwencyjnym odczycie pliku. Dla małych plików wydajność XFS była gorsza od wydajności Ext2. W miarę wzrostu objętości wydajność obydwu systemów plików okazała się zbliżona.



Test zapisu

polegał na sekwencyjnym zapisie rekordów do pliku. Wyniki testu wskazują na znaczny wpływ bufora dyskowego (256 MB), który maleje wraz z jego przepelnianiem się. Dla niewielkich plików (kilka standardowych ekstentów - 64 kB) wydajność XFS była do pięciu razy większa niż Ext2. Dla plików średniej wielkości (ale mieszczących się w buforze dyskowym) wydajność Ext2 była podobna do wydajności XFS. Po przekroczeniu pojemności bufora wydajność Ext2 drastycznie spadła. Wydajność XFS okazała się wtedy nawet dwa razy większa - taki wynik to niewątpliwie również rezultat lepszej wielowątkowości systemu





Zajrzyj: archiwum.pckurier.pl/art.asp?id=4609

Publikacja jest fragmentem pracy dyplomowej wykonanej w Instytucie Informatyki Politechniki Białostockiej.

==

Blok opisujący cały system plików (plik źródłowy fs/xfs/xfs_sb.h)

```
#define XFS_SB_MAGIC          0x58465342 /* 'XFSB' */
#define XFS_SB_VERSION_1    1 /* 5.3, 6.0.1, 6.1 */
#define XFS_SB_VERSION_2    2 /* 6.2 - dodatkowe atrybuty dla plików */
#define XFS_SB_VERSION_3    3 /* 6.2 - nowa wersja i-w•z•ów */
#define XFS_SB_VERSION_4    4 /* 6.2+ - nowa wersja ci•g•o•ci */

typedef struct xfs_sb
{
    __uint32_t      sb_magicnum;          /* wersja superbloku == XFS_SB_MAGIC */
    __uint32_t      sb_blocksize;         /* rozmiar bloku w bajtach */
    xfs_drfsbno_t   sb_dblocks;           /* liczba bloków danych */
    ...
    uuid_t          sb_uuid;              /* unikalny numer systemu plików */
    xfs_ufsbn_t     sb_logstart;           /* numer pierwszego bloku logów */
    xfs_ino_t        sb_rootino;           /* numer i-w•z•a b•d•cego korzeniem
                                           drzewa katalogowego */
    ...
    xfs_agblock_t   sb_agblocks;           /* rozmiar grupy alokacji w blokach */
    xfs_agnumber_t  sb_agcount;           /* liczba grup alokacji */
    ...
    xfs_extlen_t    sb_logblocks;          /* rozmiar logu w blokach */
    __uint16_t      sb_versionnum;         /* wersja superbloku XFS == XFS_SB_VERSION */
    __uint8_t       sb_imax_pct;           /* maksymalny procentowy udział
                                           i-w•z•ów w przestrzeni systemu plików */
    ...
    __uint64_t      sb_icount;             /* liczba zaj•tych i-w•z•ów */
    __uint64_t      sb_ifree;             /* liczba wolnych i-w•z•ów */
    __uint64_t      sb_fdblocks;          /* liczba wolnych bloków */
    ...
    xfs_ino_t       sb_uquotino;          /* numer i-w•z•a dla kwot u•ytkowników */
    ...
} xfs_sb_t;
```

gdzie:

- w pliku źródłowym fs/xfs/xfs_types.h:

```
typedef __uint64_t xfs_ufsbn_t;           /* numer bloku w grupie alokacji */
typedef __uint64_t xfs_drfsbno_t;         /* numer bloku w systemie plików */
typedef __uint32_t xfs_agblock_t;         /* liczba bloków w grupie alokacji */
typedef __uint32_t xfs_extlen_t;          /* rozmiar ci•g•o•ci w blokach */
typedef __uint32_t xfs_agnumber_t;        /* numer grupy alokacji */
```

- w pliku źródłowym fs/xfs/support/types.h:

```
typedef __u64      xfs_ino_t;             /* numer i-w•z•a */
```

==

Opis drzewa wolnych ekstentów

```

#define XFS_AGF_MAGIC 0x58414746 /* 'XAGF' */
#define XFS_AGI_MAGIC 0x58414749 /* 'XAGI' */
#define XFS_AGF_VERSION 1
#define XFS_AGI_VERSION 1
/*
 * Drzewo btree numer 0 indeksuje pozycje, a numer 1 - rozmiar.
 */
#define XFS_BTNUM_AGF ((int)XFS_BTNUM_CNTI + 1)

typedef struct xfs_agf
{
    /*
     * Nagłówek informacyjny grupy alokacji
     */
    __uint32_t agf_magicnum; /* symbol nagłówka == XFS_AGF_MAGIC */
    __uint32_t agf_versionnum; /* wersja nagłówka == XFS_AGF_VERSION */
    xfs_agnumber_t agf_segno; /* numer grupy alokacji */
    xfs_agblock_t agf_length; /* rozmiar grupy alokacji w blokach */
    /*
     * Informacje o wolnej przestrzeni
     */
    xfs_agblock_t agf_roots[XFS_BTNUM_AGF]; /* bloki, w których znajdują się
                                                korzenie pary drzew wolnych ekstentów */
    __uint32_t agf_spare0;
    __uint32_t agf_levels[XFS_BTNUM_AGF]; /* wartośći kluczy określające
                                                poziom drzew wolnych ekstentów w odniesieniu
                                                do pozostałych grup alokacji */
    __uint32_t agf_spare1;
    __uint32_t agf_flfirst; /* indeks pierwszego wolnego ekstentu */
    __uint32_t agf_fllast; /* indeks ostatniego wolnego ekstentu */
    __uint32_t agf_flcount; /* liczba bloków wszystkich wolnych ciągów */
    xfs_extlen_t agf_freeblks; /* liczba wszystkich wolnych bloków */
    xfs_extlen_t agf_longest; /* rozmiar największego wolnego ekstentu */
} xfs_agf_t;

```

gdzie:

- w pliku źródłowym fs/xfs/xfs types.h:

```
typedef enum {
    XFS_BTNUM_BNOi, XFS_BTNUM_CNTi, XFS_BTNUM_BMAPI,
    XFS_BTNUM_INOi, XFS_BTNUM_MAX
} xfs_btnum_t;
```

- w pliku źródłowym fs/xfs/xfs_inum.h:

```
typedef __uint32_t xfs_agino_t; /* numer i-w•z•a w grupie alokacji */
```

Opis drzewa i-węzłów

[illegible]


```

    * tabela skrótów do li•ci ze zwolnionymi i-w•z•ami
    */
    xfs_agino_t      agi_unlinked[XFS_AGI_UNLINKED_BUCKETS];
} xfs_agi_t;

```

==

Wykaz bloków zajmowanych przez drzewo wolnych ekstentów

Struktura pomocnicza przyspieszająca znajdowanie wolnej przestrzeni:

```

#define XFS_AGFL_SIZE      (BBSIZE / sizeof(xfs_agblock_t))
typedef struct xfs_agfl
{
    xfs_agblock_t agfl_bno[XFS_AGFL_SIZE];
} xfs_agfl_t;

```

gdzie:

- w pliku źródłowym include/linux/xfs_fs.h:

```

#define BBSHIFT 9
#define BBSIZE (1<<BBSHIFT)

```

==

Nagłówek logu XFS (plik źródłowy fs/xfs/xfs_log_priv.h)

```

#define XLOG_MAX_RECORD_BSIZE (32*1024)

typedef struct xlog_rec_header
{
    uint h_magicno; /* symbol logu */
    uint h_cycle; /* stan zapisu logu */
    int h_version; /* wersja logu */
    int h_len; /* rozmiar w bajtach */
    xfs_lsn_t h_lsn; /* numer sekwencyjny logu */
    xfs_lsn_t h_tail_lsn; /* numer sekwencyjny wcześniejszego,
                           niezatwierdzonego logu */

    uint h_chksum;
    int h_prev_block; /* wzgl.dny numer bloku poprzedniego logu */
    int h_num_logops; /* liczba logowanych operacji w ramach tego logu */
    uint h_cycle_data[XLOG_MAX_RECORD_BSIZE / BBSIZE];
    int h_fmt; /* format logu */
    uuid_t h_fs_uuid; /* UUID systemu plików */
} xlog_rec_header_t;

```

gdzie:

- w pliku źródłowym fs/xfs/xfs_types.h

```

typedef __int64_t xfs_lsn_t; /* numer sekwencyjny logu */

```

==

Struktura B⁺- drzew (plik źródłowy fs/xfs/xfs_btree.h)

```

/*
 * Nagłówek i blok drzewa B+.
 */
typedef struct xfs_btree_hdr
{
    __uint32_t bb_magic; /* rodzaj */
    __uint16_t bb_level; /* warto•• 0 oznacza li•• */
    __uint16_t bb_numrecs; /* liczba danych */
} xfs_btree_hdr_t;

typedef struct xfs_btree_block
{
    xfs_btree_hdr_t bb_h; /* nagłówek zdefiniowany powyżej */
    union {
        struct {
            xfs_agblock_t bb_leftsib;
            xfs_agblock_t bb_rightsib;
        } s; /* wskazania do sąsiednich li•ci w grupie alokacji */
    };
} xfs_btree_block_t;

```

```

    struct {
        xfs_dfsbno_t bb_leftsib;
        xfs_dfsbno_t bb_rightsib;
    } l; /* wskazania do s•siednich li•ci w systemie plików */
} bb_u;
} xfs_btree_block_t;

```

==

Pojedynczy i-węzeł opisany strukturami (plik źródłowy fs/xfs/xfs_dinode.h)

```

#define XFS_DINODE_MAGIC 0x494e /* 'IN' */

typedef struct xfs_timestamp {
    __int32_t t_sec; /* sekundy */
    __int32_t t_nsec; /* nanosekundy */
} xfs_timestamp_t;

typedef struct xfs_dinode_core
{
    __uint16_t di_magic; /* symbol i-w•z•a = XFS_DINODE_MAGIC */
    __uint16_t di_mode; /* rodzaj pliku */
    __int8_t di_version; /* wersja i-w•z•a */
    ...
    __uint32_t di_uid; /* UID w•a•ciciela */
    __uint32_t di_gid; /* GID w•a•ciciela */
    __uint32_t di_nlink; /* liczba dowi•za• do pliku */
    ...
    xfs_timestamp_t di_atime; /* czas ostatniego dost•pu */
    xfs_timestamp_t di_mtime; /* czas ostatniej modyfikacji */
    xfs_timestamp_t di_ctime; /* czas utworzenia */
    xfs_fsize_t di_size; /* rozmiar pliku w bajtach */
    xfs_dfsbno_t di_nblocks; /* liczba bezpo•rednich i po•rednich
                             bloków danych */
    xfs_extlen_t di_extsize; /* rozmiar najmniejszego ekstentu
                             w ramach pliku */
    xfs_extnum_t di_nextents; /* liczba ekstentów w ramach pliku */
    ...
} xfs_dinode_core_t;

/* struktura i-w•z•a zapisywanego na dysku */
typedef struct xfs_dinode
{
    xfs_dinode_core_t di_core;
    ...
    union {
        xfs_bmdr_block_t di_bmbt; /* blok korzenia B+-drzewa ekstentów */
        xfs_bmbt_rec_32_t di_bmx[1]; /* blok tabeli ekstentów */
        xfs_dir_shortform_t di_dirsf; /* skrócony opis katalogu */
        xfs_dir2_sf_t di_dir2sf; /* skrócony opis katalogu, wersja 2 */
        ...
        xfs_dev_t di_dev; /* urz•dzenie - je•eli plik
                           jest urz•dzeniem znakowym lub blokowym */
        uuid_t di_muuid; /* warto•• UUID systemu plików */
        char di_symlink[1]; /* warto•• dowi•zania symbolicznego */
    } di_u;
    union {
        xfs_bmdr_block_t di_abmbt; /* blok korzenia B+-drzewa ekstentów */
        xfs_bmbt_rec_32_t di_abmx[1]; /* blok tabeli ekstentów */
        xfs_attr_shortform_t di_attrsf; /* skrócona lista atrybutów pliku */
    } di_a;
} xfs_dinode_t;

```

gdzie:

- w pliku źródłowym fs/xfs/xfs_types.h:

```

typedef __int32_t xfs_extnum_t; /* liczba ekstentów w ramach pliku */
typedef __uint32_t xfs_extlen_t; /* rozmiar ekstentu w blokach */
typedef __int64_t xfs_fsize_t; /* rozmiar pliku w bajtach */

```

- w pliku źródłowym fs/xfs/xfs_attr_sf.h:

```

/* skrócona lista atrybutów pliku */
typedef struct xfs_attr_shortform {
    struct xfs_attr_sf_hdr {
        __uint16_t totsize; /* ca•kowity rozmiar nag•ówka */
        __uint8_t count; /* liczba atrybutów */
    } hdr;
    struct xfs_attr_sf_entry {
        __uint8_t namelen; /* rozmiar nazwy atrybutu */
        __uint8_t valuelen; /* rozmiar warto•ci atrybutu */
        __uint8_t flags; /* znaczniki atrybutu */
        __uint8_t nameval[1]; /* nazwa i warto•• atrybutu */
    } ent;
} xfs_attr_sf_t;

```

```

    } list[1]; /* tabela o zmiennej liczbie elementów */
} xfs_attr_shortform_t;

- w pliku źródłowym fs/xfs/xfs_dir_sf.h

typedef struct { __uint8_t i[sizeof(xfs_ino_t)]; } xfs_dir_ino_t;

/* skrócony opis katalogu */
typedef struct xfs_dir_shortform {
    struct xfs_dir_sf_hdr { /* nagłówek */
        xfs_dir_ino_t parent; /* numer i-w.z.a katalogu macierzystego */
        __uint8_t count; /* liczba wpisów w skróconym opisie */
    } hdr;
    struct xfs_dir_sf_entry {
        xfs_dir_ino_t inumber; /* numer i-w.z.a wpisu w katalogu */
        __uint8_t namelen; /* rozmiar nazwy */
        __uint8_t name[1]; /* nazwa */
    } list[1]; /* tabela o zmiennej liczbie elementów */
} xfs_dir_shortform_t;

```

- w pliku źródłowym fs/xfs/xfs_dir2_sf.h:

```

/* nagłówek skróconego opisu katalogu, wersja 2. */
typedef struct xfs_dir2_sf_hdr {
    __uint8_t count; /* liczba wpisów w skróconym opisie */
    __uint8_t i8count;
    xfs_dir2_inou_t parent; /* numer i-w.z.a katalogu macierzystego */
} xfs_dir2_sf_hdr_t;

/* pozycja w skróconym opisie katalogu, wersja 2. */
typedef struct xfs_dir2_sf_entry {
    __uint8_t namelen; /* rozmiar nazwy */
    xfs_dir2_sf_off_t offset; /* wzgl. dny numer dla nast. pnej pozycji */
    __uint8_t name[1]; /* nazwa */
    xfs_dir2_inou_t inumber; /* numer i-w.z.a */
} xfs_dir2_sf_entry_t;

/* skrócony opis katalogu, wersja 2. */
typedef struct xfs_dir2_sf {
    xfs_dir2_sf_hdr_t hdr; /* nagłówek opisu */
    xfs_dir2_sf_entry_t list[1]; /* skrócone wpisy */
} xfs_dir2_sf_t;

```

==

B⁺-drzewa ekstentów (plik źródłowy fs/xfs/xfs_bmap_btree.h)

```

/*
 * Nagłówek
 */
typedef struct xfs_bmdr_block
{
    __uint16_t bb_level; /* je.li 0, to li.. */
    __uint16_t bb_numrecs /* liczba danych */
} xfs_bmdr_block_t;

/*
 * Warto.. deskryptora ekstentu.
 * W 32-bitowych systemach,
 * 10:31 - znacznik ekstentu (warto.. 1 oznacza b..d).
 * 10:0-30 i 11:9-31 - numer bloku w ramach pliku.
 * 11:0-8, 12:0-31 i 13:21-31 - nr pierwszego bloku ekstentu.
 * 13:0-20 - liczba bloków ekstentu.
 * W 64-bitowych systemach,
 * 10:63 - znacznik ekstentu (warto.. 1 oznacza b..d).
 * 10:9-62 - numer bloku w ramach pliku.
 * 10:0-8 i 11:21-63 - numer pierwszego bloku ekstentu.
 * 11:0-20 - liczba bloków ekstentu.
 */
...

typedef struct xfs_bmbt_rec_32
{
    __uint32_t 10, 11, 12, 13;
} xfs_bmbt_rec_32_t;

typedef struct xfs_bmbt_rec_64
{
    __uint64_t 10, 11;
} xfs_bmbt_rec_64_t;

```

==

B⁺-drzewa (plik źródłowy fs/xfs/xfs_dir_leaf.h):

```
#define XFS_DIR_LEAF_MAPSIZE 3

typedef struct xfs_dir_leafblock {
    struct xfs_dir_leaf_hdr { /* nagłówek */
        xfs_da_blkinfo_t info;
        __uint16_t count; /* liczba elementów na liściu */
        __uint16_t namebytes; /* liczba bajtów zajmowanych przez nazwy */
        __uint16_t firstused; /* pierwszy bajt tabeli nazw */
        __uint8_t holes; /* != 0 jeżeli blok wymaga kondensacji */
        __uint8_t pad1;
        struct xfs_dir_leaf_map { /* mapa bitowa RLE wolnych bajtów */
            __uint16_t base; /* położenie wolnego obszaru */
            __uint16_t size; /* rozmiar wolnego obszaru */
        } freemap[XFS_DIR_LEAF_MAPSIZE]; /* największe wolne obszary */
    } hdr;
    struct xfs_dir_leaf_entry { /* sortowanie po kluczu, nie według nazwy */
        xfs_dahash_t hashval; /* wartość klucza na podstawie nazwy */
        __uint16_t nameidx; /* indeks w nazwach */
        __uint8_t namelen; /* długość nazwy */
        __uint8_t pad2;
    } entries[1]; /* pozycje w katalogu obejmowane przez liść */
    struct xfs_dir_leaf_name {
        xfs_dir_ino_t inumber; /* i-wzrost */
        __uint8_t name[1]; /* nazwa */
    } namelist[1]; /* nazwy */
} xfs_dir_leafblock_t;
```

gdzie:

- w pliku źródłowym fs/xfs/xfs_da_btree.h:

```
typedef struct xfs_da_blkinfo {
    xfs_dablk_t forw; /* poprzedni blok na liście */
    xfs_dablk_t back; /* następny blok na liście */
    __uint16_t magic; /* wartość kontrolna */
    __uint16_t pad;
} xfs_da_blkinfo_t;
```

- w pliku źródłowym fs/xfs/xfs_types.h:

```
typedef __uint32_t xfs_dablk_t; /* numer bloku w katalogu
                                b.d. liście atrybutów */
typedef __uint32_t xfs_dahash_t; /* wartość klucza skrótu dla nazwy
                                w katalogu b.d. atrybutu */
```

==