УДК 519.688

К.т.н., доцент Замятін Д.С., студент Петрук І.В.

**Національний технічний університет України
«Київський політехнічний інститут»**

ЗАСТОСУВАННЯ СТІЙКИХ СТРУКТУР ДАНИХ ДЛЯ ЗБЕРЕЖЕННЯ МЕТАДАНИХ ФАЙЛОВОЇ СИСТЕМИ У ОПЕРАТИВНІЙ ПАМ'ЯТІ

Abstract

Denis S. Zamyatin, assoc. prof., PhD; Igor Petrouk, student
Application of persistent data structures for filesystem metadata in memory

This paper concerns the suitablity of persistent data structures to store filesystem metadata in memory with high durability and performance requirements.

Вступ

Розробка програмного забезпечення для досліджень побудови розподілених файлових систем потребує структур даних, що будуть відповідати вимогам швидкодії, масштабування, консистентності, надійності, захисту від збоїв, малої затримки при доступі до даних.

Дослідження [1] існуючих архітектур високопродуктивних систем обробки даних показало, що майже усі подібні системи функціонують без використання синхронізації доступу до даних. Одним із способів побудови системи майже без синхронізації є утримування усіх даних, що обробляються у оперативній пам'яті. Модифікувати дані може лише один потік, що працює з максимальною ефективністю, адже не містить введення-виведення, а лише обробляє дані. Інші потоки можуть виконувати введення-виведення і взаємодіють з головним потоком за допомогою високоефективного кільцевого буферу.

Для забезпечення надійності і стійкості від збоїв доцільно застосовувати технологію знімків і журналу. Під час роботи система періодично виконує збереження знімку усіх метаданих файлової системи на диск і постійно виконує журналювання усіх запитів, що модифікують ці метадані. Ці операції не уповільнюють роботу програми, адже збереження знімку повністю асинхронне, а збереження журналу виконується шляхом дописування в кінець файлу. Додавання в кінець файлу виконується сучасними операційними системами надзвичайно швидко. Таким чином, програму можна завершити у нормальному режимі чи аварійно у будь-

який момент. Під час наступного запуску буде завантажено останній знімок і буде виконано усі необхідні зміни до метаданих згідно з журналом. Окрім цього, дані у дереві можуть бути проаналізовані у різних модулях обробки інформації, що працюють у інших потоках.

Для створення знімку необхідно зафіксувати стан файлової системи на певний момент. Це не може досягатись синхронізацією, оскільки синхронізація заборонена згідно обраної архітектури. Таким чином, необхідно забезпечити роботу такої деревоподібної структури даних, що дозволяє з максимальною швидкістю створювати знімки свого стану.

Постановка задачі

Задача полягає в дослідженні придатності *стійкого* дерева для збереження метаінформації файлової системи у оперативній пам’яті з підтримкою швидких знімків стану дерева.

Термінологія

Структура даних називається *стійкою (persistent)*, якщо вона зберігає попередню версію цієї структури після модифікації.

Опис алгоритму

Більшість структур даних можуть бути реалізовані у стійкій формі [3]. Розглянемо приклад реалізації стійкого дерева.

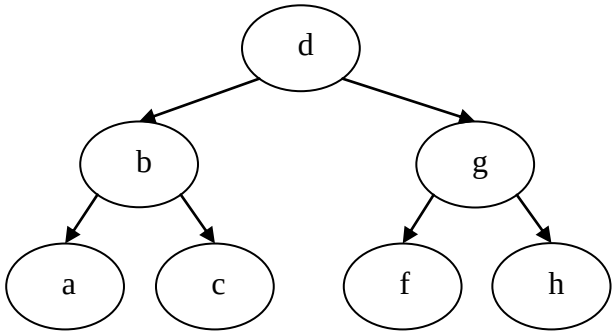

Рис 1. Початкова версія стійкого дерева

На рис.1 зображено дерево, у якому батьківський вузол має посилання на нащадків. Нащадки не мають посилань на батьківські вузли. Найпростіший спосіб надати дереву властивості стійкості без повного копіювання — зробити його вузли незмінними. Усі незмінні структури даних — стійкі, але стійка структура даних не обов'язково має бути незмінною. Таким чином, будь-який алгоритм, отримуючи на вхід

посилання на кореневий вузол дерева у якості вхідних даних, зможе працювати з ним не застосовуючи жодних засобів синхронізації. Також, цей алгоритм не буде ускладнений необхідністю підтримки випадків модифікації дерева під час його роботи. Кожне посилання на дерево є його знімком в певний момент часу.

Модифікація незмінного дерева буде полягати у створенні нової версії дерева. При цьому велика частина дерева може бути використана повторно. Для цього достатньо створити заново усі батьківські вузли зміненого або доданого елемента. Створення нових об'єктів необхідне, адже незмінність елементів дерева не дозволяє змінити посилання на нащадків.

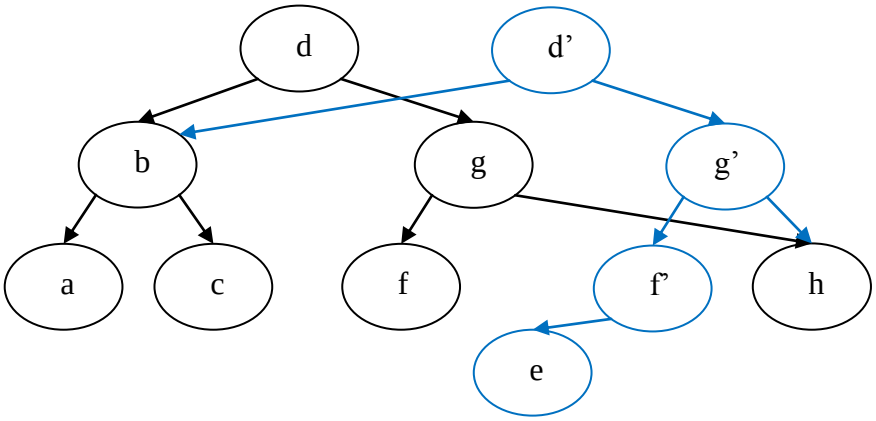

Рис 2. Нова і початкова версії дерева

На рис.2 додавання елементу *e* призвело до створення нових версій *f, g, d*, але перевикористання *a,b,c,h*. Оскільки *d* — корінь дерева, то створення нової версії *d* призвело до створення нової версії дерева. Аналогічно виконуються інші операції з модифікації даних.

З точки зору структур даних ці два дерева – незалежні в тому сенсі, що робота з одним з них ніяк не впливає на роботу з іншим, незважаючи на спільні частини. Якщо версію дерева після додавання *e* оголосити поточною або актуальною, то таке незмінне і стійке дерево зможе зовні надавати інтерфейс з підтримкою модифікації. У програмній реалізації це може виглядати як посилання на поточний корінь дерева, що змінюється, на нове значення після кожної модифікації.

Згідно такої реалізації дерева незначна модифікація елементу, який розташований на глибокому рівні у дереві, призвела б до створення значної кількості об'єктів і також видалення з пам'яті старих версій, якщо вони не будуть використовуватись. Це може призводити до значного уповільнення роботи файлової системи.

Для підвищення ефективності роботи рекомендується використовувати середовище зі «збиральником сміття». В сучасних

платформах зі збиральником сміття, наприклад у JVM, виділення пам'яті схоже за своїм принципом роботи і має таку саму швидкодію, як виділення пам'яті у стеку [4]. Відмінність полягає у тому, що при переповненні виділеної області пам'яті *A* програма призупиняє свою роботу і виконує пошук об'єктів, що доступні з усіх потоків додатку і переносить їх у нову область пам'яті *B*, щільно розміщуючи їх на початку нової області пам’яті. Ті об'єкти, що існували протягом декількох перенесень, оголошуються довгожителями і переносяться у іншу область пам'яті і до них застосовуються інші алгоритми.

При цьому об'єкти, посилання на яких не доступне, або ті, що не встигли проіснувати досить довго, не обробляються і залишаються у початковій області пам’яті, яка потім позначається як очищена. Таким чином, час роботи збирача сміття пропорційний кількості доступних об'єктів, а вартість збирання недоступних “молодих” об'єктів нульова.

Такий принцип роботи дозволяє досить часто і ефективно створювати нові об'єкти за умови, що досить скоро вони стануть недоступними. Стійке дерево, описане вище, при високому навантаженні буде досить часто створювати нові вузли. Чим вище вузол знаходиться у дереві, тим частіше він буде замінений і тим більше ймовірність для багатьох версій бути пропущеними збиральником сміттям.

Незважаючи на ефективну роботу «збиральника сміття» зі стійким деревом матимуть місце витрати часу на копіювання і підвищиться частота викликів збиральника сміття. Рішення про застосування стійкої структури даних має прийматись в залежності від задачі.

Для перевірки ефективності методу для збереження метаінформації файлової системи було розроблено програму тривіальної реалізацій стійкого дерева. Алгоритм полягав у створенні ієрархії вкладених каталогів. На кожному кроці у поточному каталозі створювалась фіксована кількість каталогів, для одного з цих каталогів операція повторювалась. На рис.3 зображено результати для режимів з 500 і 1000 каталогів на рівні.

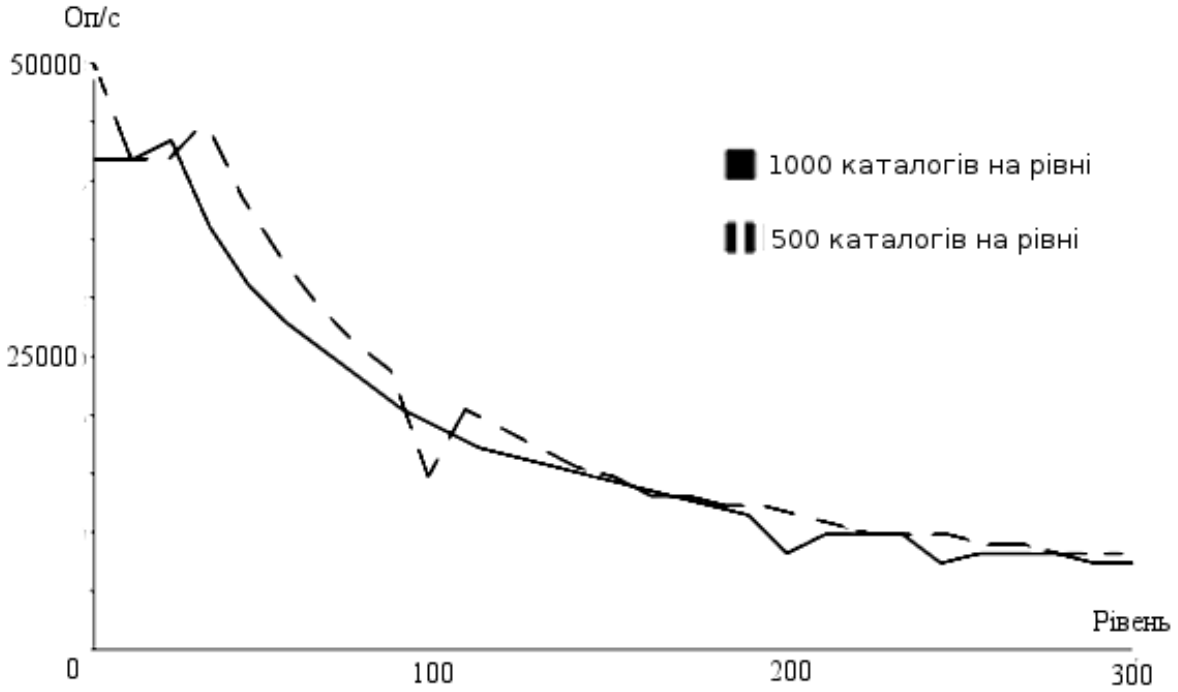

Рис 3. Графік швидкодії

На осі X вказано глибину дерева, на осі Y – кількість операцій додавання каталогу в секунду на найглибшому рівні може виконувати система за умови забезпечення стійкості.

Для тестування було використано тривіальний алгоритм, але структура даних придатна до оптимізації. Оскільки лише один потік виконує запис у стійке дерево, то досить просто реалізувати пакетне додавання підкаталогів у каталог. Таким чином, вузли верхнього рівня будуть перезаписані лише один раз для великої кількості вузлі-нащадків. Можливі оптимізації стосуються лише додавання або модифікації елементів. Читання елементів, що буде відбуватись частіше, не створює нових об’єктів, не потребує зміни версій дерева і виконується на порядки швидше.

Висновки

Швидкодія описаного алгоритму виявилась цілком прийнятною для глибини каталогів середньої реальної файлової системи. Час виконання операції залежить від глибини елемента у дереві, але майже не залежить від кількості елементів у каталозі. При цьому буде забезпечено коректну роботи програми у багатопоточному середовищі. Операції аналізу даних, такі як збереження знімку на диск, будуть безпечно проводитись у інших потоках і не впливати на швидкість роботи основного потоку, що виконує зміну даних. З цього можна зробити висновок, що стійке дерево –

прийнятна структура даних для збереження метаінформації файлової системи, ця структура данних відповідає вимогам щодо швидкодії і забезпечує захист від збоїв.

Література

1. Disruptor [Електроний ресурс]: High performance alternative to bounded queues for exchanging data between concurrent threads. – Електрон. дан. – [USA], 2011. – Режим доступу: http://disruptor.googlecode.com/files/Disruptor-1.0.pdf. – Загол. з титул. екрану. – Мова: англ. – Перевірено: 12.03.2012.
2. *Kaplan, Haim* (2001). "Persistent data structures". Handbook on Data Structures and Applications (CRC Press).
3. *Chris Okasaki* (1998). «Purely Functional Data Structures», Cambridge University Press.
4. Oracle.com [Електроний ресурс]: Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine – Електрон. дан. – [USA], 2010. – Режим доступу: http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html. – Загол. з титул. екрану. – Мова: англ. – Перевірено: 12.03.2012.