مستندات پروژه عملی دوم پینتوس

استاد: دکتر شهاب‌الدّین نبوی

اعضای گروه:

محمد خدّام

حامد خادمی خالدی

متین زیودار

امیر حلاجی

بهار ۱۴۰۰

چکیده

در سال‌های اخیر، طرّاحان فایل سیستم، متمایل به استفاده از Btree و گونه‌های مختلف راجع به آن برای از بین بردن bottleneckهای در عملکرد دایرکتوری شده‌اند. ایده‌ی اضافه‌کردن دایرکتوری Btree به فایل سیستم Ext2 لینوکس خیلی مورد بحث قرار گرفته؛ امّا هیچ‌گاه به مرحله‌ی پیاده‌سازی نرسیده است و این بیشتر به خاطر بیزاری و نفرت توسعه‌دهندگان نسبت به پیچیدگی کار است و نه به خاطر تنبلی‌شان!

ساختمان داده‌ها و الگوریتم‌ها

ساختمان داده‌ی Htree

یک بیت flag در inode دایرکتوری، به ما نشان می‌دهد که آیا این دایرکتوری index شده‌است یا خیر. اگر بیت flag ما تعیین‌شده باشد(اصطلاحاً set شده‌باشد.)، آنگاه، اولین بلاک دایرکتوری به عنوان root از Htree شناخته می‌شود.

Root فهرست Htree، به عنوان اولین بلاک از فایل دایرکتوری شناخته می‌شود. برگ‌های Htree، بلاک‌های نرمال Ext هستند که به صورت مستقیم و یا غیر مستقیم، به root ارجاع داده می‌شوند. احتمال استفاده از اشاره‌گرهای فیزیکی به دلیل سودمندی بیشتر است؛ ولی به خاطر نیازمندی شدید به پیوستگی‌سازی‌ها برای مدیریت اشاره‌گرهایی که در بسیاری از قسمت‌های Ex2 در بیرون مدیریت کد

دایرکتوری هستند، این ایده و ترفند هیچ وقت به مرحله‌ی استفاده نرسید. خوشبخانه، به خاطر تغییرات اخیر لینوکس به سمت فهرست سازی منطقی(logical indexing)، بلاک‌های اشاره‌گر منطقی دیگر سودمندی و بازدهی کمتری نسبت به نوع فیزیکی ندارند.

یک htree به جای استفاده از اسم‌ها، از کلیدهایی استفاده می‌کند که در واقع همان hash اسامی هستند. هر کلید هَش، لزوماً به یک دایرکتوری منفرد و خاصّی ارجاع نمی‌دهد؛ بلکه به یک محدوده‌ای از ورودی‌ها(entry) که داخل یک بلاک منفرد برگ(single leaf block) مرتب‌سازی شده‌اند، ارجاع می‌دهد.

ریشه‌ی htree، همچنین شامل یک آرایه‌ای از ورودی‌ها به فرمت یکسان با سطح اول است. اشاره‌گرها، به بلاک‌های اندیسی(index blocks)، بیشتر از بلاک‌های برگ(leaf blocks) ارجاع می‌دهند و کلیدهای هش محدوده‌های پایین‌تر را به کلیدهای هشِ ارجاع داده‌شده توسط بلاک‌های اندیسی اختصاص می‌دهند.

کاوشگر هش(Hash Probe)

اولین قدم در هر دایرکتوری نمایه شده(indexed directory)، پیدا کردن ریشه، اولین بلاک از فایل دایرکتوری، است. سپس در مرحله‌ی بعد، برای از بین بردن هر اندیس خراب و منحرف‌کننده‌ای که می‌تواند منجر به ایجاد مشکل در هسته‌ی سیستم‌عامل بشود، تعدادی تست انجام می‌شود. تشخیص هرگونه ناسازگاری منجر به

بازگشتن عملیات دایرکتوری (directory operation) به سمت جستجوی خطی می‌شود. در این حالت، بیشترین شانس برای دسترسی به درایو(volume) خراب، به کاربر داده می‌شود.

جدول هش(Hash table)

هر جدول هش نرمال، یک آرایه‌ی خطّی از bucket ها است و کلید هش، به صورت مستقیم bucketای را index می‌کند که در جستجوی آن هستیم. بنابراین، جستجو برای پیدا کردن bucket صحیح خیلی سریع می‌انجامد. یک اشکال اصلی در جدول هش این است که اندازه‌ی آن، نه خیلی بزرگ و نه خیلی کوچک باشد. جدول هشی که خیلی بزرگ باشد، منجر به اتلاف فضا و جدول هش خیلی کوچک منجر به برخوردهای مکرر و با تعداد بالا می‌شود. انتخاب یک اندازه‌ی مناسب برای جدول هش، یک مسئله‌ی چالشی است. راه حل این مشکل این است که اندازه‌ی جدول هش به اندازه‌ی تعداد رشته‌های موجود در جدول هش بالا برود. هنگامی که جدول هشِ ما در آستانه‌ی پر شدن قرار می‌گیرد، محتویات فعلی جدول به علّت عدم فضای کافی، به یک جدول بزرگ‌تر منتقل می‌شوند و عملیاتی تحت عنوان «هش مجدد» (rehashing)، صورت می‌گیرد.

اعمال تغییرات جدید

ساختار btable

```
struct btNode {
    int isLeaf;     /* is this a leaf node? */
    int numKeys;    /* how many keys does this node contain? */
    int keys[MAX_KEYS];
    struct dir_entries entries[MAX_KEYS];
    off_t offests[MAX_KEYS];
    struct btNode *kids[MAX_KEYS+1];  /* kids[i] holds nodes < keys[i] */
};
```

همانطور که مشخص است، `btNode` یک `struct` است که تعدادی فیلد دارد.

فیلد `isLeaf` بیان می‌کند که آیا `node` ما یک برگ است یا خیر.

فیلد `numKeys` بیان می‌کند که گره‌ی فعلی چه تعداد کلید را شامل می‌شود.

همچنین، یک آرایه از اعداد صحیح را نگهداری می‌کنیم که اندازه‌ی آن به تعداد بیشینه‌ی کلیدها است. علاوه بر این‌ها، آرایه‌ای از `offset` ها به همین اندازه داریم.

دو ساختار دیگر نیز به نام‌های `entries` از جنس `dir_entries` و یک اشاره‌گر به نام `kids` از جنس `btNode` داریم.

تابع btCreate

```
bTree
btCreate(void)
{
    bTree b;

    b = malloc(sizeof(*b));
    ASSERT (b);

    b->isLeaf = 1;
    b->numKeys = 0;

    return b;
}
```

این تابع، ورودی‌ای دریافت نمی‌کند و یک نمونه از `bTree` را به ما در خروجی برمی‌گرداند. در ابتدای تابع، یک نمونه از `bTree` تعریف می‌کنیم و به اندازه‌ی آدرس آن، به آن حافظه تخصیص می‌دهیم. چون این گره، در حال حاضر کلیدی را شامل نمی‌شود، فیلد `numKeys` را صفر و چون برگ است، فیلد `isLeaf` را 1 می‌گذاریم.

تابع btDestroy

```
void
btDestroy(bTree b)
{
    int i;

    if(!b->isLeaf) {
        for(i = 0; i < b->numKeys + 1; i++) {
            btDestroy(b->kids[i]);
        }
    }

    free(b);
}
```

این تابع، یک ورودی از جنس `bTree` دریافت می‌کند و خروجی‌ای برنمی‌گرداند. نحوه‌ی کارش به این‌گونه است که در ابتدا یک `i` تعریف می‌کنیم و اگر `b` یک برگ نباشد، به اندازه‌ی تعداد کلیدهایش پیمایش می‌کنیم و بچه‌هایش را `destroy` می‌کنیم. پس از این‌ها در انتها، `b` را آزاد می‌کنیم.(`free(b)`).

تابع searchKey

```
static int
searchKey(int n, const int *a, int key)
{
    int lo;
    int hi;
    int mid;

    /* invariant: a[lo] < key <= a[hi] */
    lo = -1;
    hi = n;

    while(lo + 1 < hi) {
        mid = (lo+hi)/2;
        if(a[mid] == key) {
            return mid;
        } else if(a[mid] < key) {
            lo = mid;
        } else {
            hi = mid;
        }
    }
    return hi;
}
```

این تابع به ما یک عدد صحیح برمی‌گرداند که **key** ای است که به دنبال آن می‌گردیم. نحوه‌ی عملکرد آن به این صورت است که **hi** اندیس بالا و **lo** اندیس پایین است. تا زمانی که **lo** به **hi** نرسیده یعنی تا زمانی که کلید مورد نظرمان پیدا نشده است، حلقه را پیمایش می‌کنیم و هر سری اندیس وسط را برابر میانگین بالا و پایین می‌گذاریم. اگر عنصر با اندیس وسط همان کلید موردنظرمان باشد، همان مقدار اندیس**mid** را به عنوان خروجی تابع بر می‌گردانیم.

تابع btSearch

```
int
btSearch(bTree b, int key,
         struct dir_entry *ep, off_t *ofsp)
{
    int pos;

    /* have to check for empty tree */
    if(b->numKeys == 0) {
        return 0;
    }
```

```
    /* look for smallest position that key fits below */
    pos = searchKey(b->numKeys, b->keys, key);

    if(pos < b->numKeys && b->keys[pos] == key) {
        ep = b->entries[pos];
        ofsp = b->offests[pos];
        return 1;
    } else {
        return(!b->isLeaf && btSearch(b->kids[pos], key, &ep, &ofsp));
    }
}
```

این تابع برای اعمال جستجو در ساختار `btree` می‌باشد به این صورت که درخت `btree` و کلیدی که قرار است جستجو شود را به عنوان ورودی می‌گیرد و به صورت بازگشتی عملیات باینری سرچ انجام می‌شود. در صورت پیدا کردن کلید در این درخت پارامترهای `ep` و `ofsp` که به ترتیب نمایانگر `directory entry` و `byte offset` می‌باشد ست می‌کند.

تابع btInsertInternal

```
static bTree
btInsertInternal(bTree b, int key, int *median,
            struct dir_entry *ep, off_t *ofsp)
{
    int pos;
    int mid;
    bTree b2;
    pos = searchKey(b->numKeys, b->keys, key);
    if(pos < b->numKeys && b->keys[pos] == key) {
        /* nothing to do */
        return 0;
    }
```

```
if(b->isLeaf) {

        /* everybody above pos moves up one space */
        memmove(&b->keys[pos+1], &b->keys[pos], sizeof(*(b->keys)) * (b->numKeys -
 pos));
        b->keys[pos] = key;
        b->entries[pos] = ep;
        b->offests[pos] = ofsp;
        b->numKeys++;

    }
```

```
else {

        /* insert in child */
        b2 = btInsertInternal(b->kids[pos], key, &mid, &ep, &ofsp);

        /* maybe insert a new key in b */
        if(b2) {

            /* every key above pos moves up one space */
            memmove(&b->keys[pos+1], &b->keys[pos], sizeof(*(b->keys)) * (b->numKeys -
 pos));
            memmove(b->entries[pos+1], &b->entries[pos], sizeof(*(b->entries)) * b-
>numKeys - pos);
```

```
            memmove(b->offests[pos+1], &b->offests[pos], sizeof(*(b->offests)) * b-
>numKeys - pos);
            /* new kid goes in pos + 1*/
            memmove(&b->kids[pos+2], &b->kids[pos+1], sizeof(*(b->keys)) * (b->numKeys -
 pos));
            memmove(b->entries[pos+2], &b->entries[pos+1], sizeof(*(b->entries)) * b-
>numKeys - pos);
            memmove(b->offests[pos+2], &b->offests[pos+1], sizeof(*(b->offests)) * b-
>numKeys - pos);

            b->keys[pos] = mid;
            b->kids[pos+1] = b2;
            b->numKeys++;
```

```
/* we waste a tiny bit of space by splitting now
     * instead of on next insert */
    if(b->numKeys >= MAX_KEYS) {
        mid = b->numKeys/2;

        *median = b->keys[mid];
        b2 = malloc(sizeof(*b2));
        b2->numKeys = b->numKeys - mid - 1;
        b2->isLeaf = b->isLeaf;
```

```
        /* make a new node for
keys > median */
        /* picture is:
         *
         *      3 5 7
         *      A B C D
         *
         * becomes
         *          (5)
         *      3       7
         *      A B     C D
         */
```

```
memmove(b2->keys, &b->keys[mid+1], sizeof(*(b->keys)) * b2->numKeys);
        memmove(b2->entries, &b->entries[mid+1], sizeof(*(b->entries)) * b2->numKeys);
        memmove(b2->offests, &b->offests[mid+1], sizeof(*(b->offests)) * b2->numKeys);
        if(!b->isLeaf) {
            memmove(b2->kids, &b->kids[mid+1], sizeof(*(b->kids)) * (b2->numKeys + 1));
            memmove(b2->entries, &b->entries[mid+1], sizeof(*(b->entries)) * b2-
>numKeys + 1);
            memmove(b2->offests, &b->offests[mid+1], sizeof(*(b->offests)) * b2-
>numKeys + 1);
        }
        b->numKeys = mid;
        return b2;
    } else {return 0;}}
```

این تابع وظیفه درج و `insert` کردن به روش ساختار `btree` می‌باشد با در نظر گرفتن این نکته که در مواقعی که انتقال داریم علاوه بر `key` ها، `directory` های `entry` و `byte offset` هم به بالا منتقل می‌شود.

تابع btInsert

```
void
btInsert(bTree b, int key,
        struct dir_entry *ep, off_t *ofsp)
{
    bTree b1;   /* new left child */
    bTree b2;   /* new right child */
    int median;

    b2 = btInsertInternal(b, key, &median, &ep, &ofsp);
```

```
    if(b2) {
        /* basic issue here is that we are at the root */
        /* so if we split, we have to make a new root */
        b1 = malloc(sizeof(*b1));
        ASSERT (b1);
        /* copy root to b1 */
        memmove(b1, b, sizeof(*b));
        /* make root point to b1 and b2 */
        b->numKeys = 1;
        b->isLeaf = 0;
        b->keys[0] = median;
        b->kids[0] = b1;
        b->kids[1] = b2; }}
```

این تابع عملکردی مشابه تابع بالا دارد با این تفاوت که `ep` و `ofsp` به

ساختار `Htable`

ساختار htable node

```
struct htNode {
    struct bTree b;
    struct hash h;
};
```

هر `htNode`، یک `struct` است که دو فیلد از جنس `bTree` و `hash` دارد.

تابع htCreate

```
struct hTree
htCreate(void)
{
    hTree ht;

    ht->b = btCreate();
    hash_init(ht->h, dir_hash, dir_less, NULL);

    return ht;
}
```

این تابع با استفاده از ساختار **btree** و **hash table** یک **htree** می‌سازد و آن را برمی‌گرداند.

تابع htDestroy

```
void
htDestroy(hTree ht)
{
    btDestroy(ht->b);
    hash_clear(ht->h);
}
```

این تابع درخت **htree** را با پاک کردن **btree** و **hash table** از بین می‌برد و فضای تخصیص‌داده شده به آن را آزاد می‌کند.

تابع htSearch

```
struct bool
htSearch(const struct hTree ht, const char *name,
        struct dir_entry *ep, off_t *ofsp)
{
    int key;

    e = hash_find (ht->h, &name);
    if (e == NULL)
        return NULL
    key = hash_entry (e, struct dir_entry, char);
    btSearch(ht->b, key, &ep, &ofsp);
}
```

این تابع عملیات سرچ در `htree` را انجام می‌دهد به این شکل که ابتدا وجود کلید مورد نظر و منتسب به نام دایرکتوری (ورودی دوم تابع) در `htable` با استفاده از `hash_find` بررسی می‌شود. اگر چنین کلیدی وجود داشت با استفاده از `hash_entry` گرفته می‌شود و سپس با استفاده این کلید و تابع `btSearch` که از قبل تعریف کردیم عملیات جستجو انجام می‌شود.

تابع htInsert

```
void
htInsert(const struct hTree ht, const char *name, block_sector_t inode_sector,
        struct dir_entry *ep, off_t *ofsp)
{
    struct dir_entry e;
    int key;
    if (hash_insert (ht->h, &name) == NULL)
        return;
    btInsert(ht->b, key, &ep, &ofsp)        // panic if sth goes wrong
    e.in_use = true;
    strlcpy (e.name, name, sizeof e.name);
    e.inode_sector = inode_sector;
}
```

این تابع وظیفه‌ی اضافه کردن دایرکتوری جدید به ساختار `htree` را برعهده دارد به این شکل که ابتدا نام دایرکتوری به `hash table` اضافه می‌شود. در صورت موفقیت‌آمیز بودن، کلید منتسب به آن به همراه `directory entry` و `byte offset` به ساختار `btree` با استفاده از تابع `btInsert` اضافه می‌شود. در انتها مشخصات `dir_entry` جدید ساخته شده را با استفاده از مشخصات آن یعنی نام، `inode_sector` و همچنین `in_use` بودن تکمیل می‌کنیم.