دانشگاه صنعتی امیرکبیر

آزمایشگاه سیستم‌عامل

جلسه چهارم: پیاده سازی
red-block tree

مدرس: مینا یوسف‌نژاد

- **my_node:** این ساختار شامل یک کلید key و یک گره قرمز-سیاه rb_node است.
- **insert:** این تابع یک گره جدید به درخت اضافه می‌کند. اگر کلید تکراری باشد، عملیات اضافه کردن لغو می‌شود.
- **traverse:** این تابع برای پیمایش و چاپ کلیدهای گره‌ها با استفاده از rb_first و rb_next نوشته شده است.
- **free_tree:** این تابع تمامی گره‌ها را از درخت حذف و حافظه‌ی آن‌ها را آزاد می‌کند.
- **rbtree_example_init , rbtree_example_exit:** این توابع ماژول را در هنگام بارگذاری و آزادسازی مدیریت می‌کنند.

تعریف ساختار my_node

```
struct my_node {
    int key;
    struct rb_node node;
};
```

یک گره از درخت قرمز-سیاه را تعریف می‌کند که دارای دو فیلد است:

- key یک مقدار عددی که کلید گره است و برای مرتب‌سازی استفاده می‌شود.
- node از نوع struct rb_node که ساختار اصلی یک گره در درخت قرمز-سیاه در لینوکس است.
- این ساختار از rb_node که در هدر فایل‌های کرنل تعریف شده استفاده می‌کند.

تعریف ریشه درخت

```
static struct rb_root my_tree = RB_ROOT;
```

یک متغیر به نام my_tree از نوع rb_root ایجاد می‌کند که به عنوان ریشه درخت قرمز-سیاه عمل می‌کند. با استفاده از RB_ROOT، متغیر my_tree به طور پیش‌فرض به حالت خالی (NULL) مقداردهی می‌شود.

تابع insert

```c
static int insert(struct rb_root *root, struct my_node *data) {
    struct rb_node **new = &(root->rb_node), *parent = NULL;

    while (*new) {
        struct my_node *this = container_of(*new, struct my_node, node);

        parent = *new;
        if (data->key < this->key)
            new = &((*new)->rb_left);
        else if (data->key > this->key)
            new = &((*new)->rb_right);
        else
            return -1;
    }

    rb_link_node(&data->node, parent, new);
    rb_insert_color(&data->node, root);

    return 0;
}
```

پارامترها:

- struct rb_root *root: اشاره‌گری به ریشه درخت قرمز-سیاه.
- struct my_node *data: اشاره‌گری به داده‌ای که باید وارد درخت شود. این داده شامل یک کلید (key) و یک گره از نوع rb_node است.

مقداردهی اولیه:

- new: این متغیر اشاره‌گری دوگانه به rb_node است که نشان می‌دهد ما در حال حاضر به کدام گره درخت اشاره می‌کنیم. ابتدا روی گره ریشه قرار دارد.
- parent: متغیری از نوع rb_node که در ابتدا NULLاست. این متغیر به گره والد گره‌ای که قرار است وارد شود اشاره خواهد کرد.

container_of

```c
#define container_of(ptr, type, member) \
    ((type *)((char *)(ptr) - offsetof(type, member)))
```

ماکروی container_of که در کرنل لینوکس تعریف شده است، برای دسترسی به ساختار کامل پدر (container) یک عضو یا فیلد مشخص استفاده می‌شود. این ماکرو یک نوع بازیابی اشاره‌گر است که به ما اجازه می‌دهد از طریق آدرس یک فیلد، به ساختار کلی که شامل آن فیلد است، دسترسی پیدا کنیم. این ماکرو در فایل linux/kernel.h تعریف شده است.

- ptr اشاره‌گر به فیلدی است که از آن قصد داریم به ساختار اصلی برسیم.
- type نوع ساختار کلی است.
- member نام فیلدی است که اشاره‌گر ptrبه آن اشاره می‌کند.

این ماکرو از ماکروی offsetof استفاده می‌کند تا فاصله (offset) فیلد node از ابتدای ساختار my_node را محاسبه کند و سپس این فاصله را از آدرس ptr کم می‌کند تا به ابتدای ساختار my_node برسد.

```c
struct my_node *this = container_of(*new, struct my_node, node);
```

تابع insert

```c
static int insert(struct rb_root *root, struct my_node *data) {
    struct rb_node **new = &(root->rb_node), *parent = NULL;

    while (*new) {
        struct my_node *this = container_of(*new, struct my_node, node);

        parent = *new;
        if (data->key < this->key)
            new = &((*new)->rb_left);
        else if (data->key > this->key)
            new = &((*new)->rb_right);
        else
            return -1;
    }

    rb_link_node(&data->node, parent, new);
    rb_insert_color(&data->node, root);

    return 0;
}
```

پیمایش درخت:

در این بخش از کد، درخت را به صورت دودویی پیمایش می‌کنیم تا محل صحیح برای درج گره جدید را پیدا کنیم. در هر مرحله:

- از ماکروی container_of استفاده می‌کنیم تا از گره rb_node به ساختار داده‌ی my_node که حاوی keyاست، دسترسی پیدا کنیم.
- مقدار parent را به گره فعلی اختصاص می‌دهیم.
- سپس با استفاده از key ، تصمیم گرفته می‌شود که آیا باید به زیر درخت چپ (rb_left) برویم یا زیر درخت راست (rb_right)
- اگر کلید داده جدید (data->key) کمتر از کلید گره فعلی باشد، به زیر درخت چپ می‌رویم.
- اگر کلید داده جدید بیشتر باشد، به زیر درخت راست می‌رویم.
- اگر کلید تکراری باشد تابع -۱ را برمی‌گرداند که نشان می‌دهد کلید تکراری است و نباید درج شود.

تابع insert

```c
static int insert(struct rb_root *root, struct my_node *data) {
    struct rb_node **new = &(root->rb_node), *parent = NULL;

    while (*new) {
        struct my_node *this = container_of(*new, struct my_node, node);

        parent = *new;
        if (data->key < this->key)
            new = &((*new)->rb_left);
        else if (data->key > this->key)
            new = &((*new)->rb_right);
        else
            return -1;
    }

    rb_link_node(&data->node, parent, new);
    rb_insert_color(&data->node, root);

    return 0;
}
```

درج گره جدید:

- پس از یافتن مکان صحیح برای گره جدید (که * new == NULL است)، گره جدید با استفاده از تابع rb_link_node به درخت متصل می‌شود. این تابع گره جدید را به والد (parent) و مکان صحیح (new) متصل می‌کند.

متعادل‌سازی درخت:

- پس از درج گره جدید، باید درخت را متعادل کنیم تا قوانین درخت قرمز-سیاه حفظ شوند. این کار با استفاده از تابع rb_insert_color انجام می‌شود که با بررسی رنگ گره‌ها و انجام چرخش‌های لازم، درخت را متعادل می‌کند.

تابع traverse

```c
static void traverse(struct rb_root *root) {
    struct rb_node *node;
    for (node = rb_first(root); node; node = rb_next(node)) {
        struct my_node *data = container_of(node, struct my_node, node);
        pr_info("key available: %d\n", data->key);
    }
}
```

پیمایش گره‌ها

- **شروع از اولین (کوچک‌ترین) گره درخت:** با استفاده از rb_firstبه اولین گره در درخت (سمت چپ‌ترین گره) می‌رسیم.
- **پیمایش تمام گره‌ها:** با استفاده از یک حلقه whileو تابع rb_next، از یک گره به گره بعدی می‌رویم و این کار تا زمانی که به پایان درخت برسیم (یعنی node == NULLشود) ادامه دارد.
- **دسترسی به کلید هر گره:** از ماکروی container_of استفاده می‌کنیم تا از گره rb_node به ساختار داده کامل (my_node) که شامل کلید گره است، دسترسی پیدا کنیم.
- **چاپ کلید هر گره:** در هر تکرار حلقه، کلید گره جاری را با استفاده از تابع pr_info لاگ می‌کنیم.

این کد به صورت مرتب و به ترتیب کلیدها، گره‌های درخت را پیمایش می‌کند و کلید هر گره را چاپ می‌کند.

تابع free

```c
static void free_tree(struct rb_root *root) {
    struct rb_node *node;
    struct my_node *data;

    while ((node = rb_first(root))) {
        data = container_of(node, struct my_node, node);
        pr_info("key deleted: %d\n", data->key);
        rb_erase(&data->node, root);
        kfree(data);
    }
}
```

پیمایش درخت از گره اول (کوچک‌ترین گره): با استفاده از rb_first به کوچک‌ترین گره در درخت می‌رسیم.

- **چاپ کلید گره حذف‌شده:** قبل از حذف، کلید گره را چاپ می‌کنیم تا در لاگ سیستم ثبت شود.
- **حذف گره از درخت:** با استفاده از rb_erase، گره جاری را از درخت قرمز-سیاه حذف می‌کنیم.
- **آزادسازی حافظه گره حذف‌شده:** با kfree، حافظه‌ای که برای گره تخصیص داده شده بود را آزاد می‌کنیم.

تابع init

```c
static int __init rbtree_example_init(void) {
    struct my_node *node;
    int keys[] = {10, 20, 15, 30, 25, 5};
    int i;

    pr_info("Initializing Red-Black Tree Module\n");

    for (i = 0; i < sizeof(keys)/sizeof(keys[0]); i++) {
        node = kmalloc(sizeof(struct my_node), GFP_KERNEL);
        if (!node)
            return -ENOMEM;

        node->key = keys[i];
        if (insert(&my_tree, node) != 0) {
            pr_warn("Duplicate key: %d\n", node->key);
            kfree(node);
        }
    }

    pr_info("Traversing the tree:\n");
    traverse(&my_tree);

    return 0;
}
```

- `struct my_node *node;`: اشاره‌گری به ساختار my_node که برای تخصیص حافظه و ذخیره گره‌های جدید استفاده می‌شود.
- `int keys[] = {10, 20, 15, 30, 25, 5};`: آرایه‌ای از کلیدها که مقادیر نمونه برای افزودن به درخت را در خود دارد.
- تابع rbtree_example_init: به عنوان تابع راه‌انداز ماژول کرنل عمل می‌کند و کارهای زیر را انجام می‌دهد:
- پیغام شروع مقداردهی اولیه را در لاگ سیستم ثبت می‌کند.

تابع init

```c
static int __init rbtree_example_init(void) {
    struct my_node *node;
    int keys[] = {10, 20, 15, 30, 25, 5};
    int i;

    pr_info("Initializing Red-Black Tree Module\n");

    for (i = 0; i < sizeof(keys)/sizeof(keys[0]); i++) {
        node = kmalloc(sizeof(struct my_node), GFP_KERNEL);
        if (!node)
            return -ENOMEM;

        node->key = keys[i];
        if (insert(&my_tree, node) != 0) {
            pr_warn("Duplicate key: %d\n", node->key);
            kfree(node);
        }
    }

    pr_info("Traversing the tree:\n");
    traverse(&my_tree);

    return 0;
}
```

- در یک حلقه، برای هر کلید در آرایه keys :
- یک گره جدید از نوع my_node ایجاد می‌کند و کلید را به آن تخصیص می‌دهد.
- سعی می‌کند گره را به درخت اضافه کند. اگر کلید تکراری باشد، پیغام هشدار ثبت شده و گره از حافظه آزاد می‌شود.
- پس از افزودن تمامی کلیدها، تابع traverse را فراخوانی می‌کند تا کلیدهای موجود در درخت را نمایش دهد.
- با بازگشت مقدار ۰ نشان می‌دهد که تابع با موفقیت اجرا شده است.

تابع exit

```c
static void __exit rbtree_example_exit(void) {
    pr_info("Cleaning up Red-Black Tree Module\n");

    free_tree(&my_tree);
}

module_init(rbtree_example_init);
module_exit(rbtree_example_exit);
```

تابع rbtree_example_exit به عنوان تابع خروج ماژول کرنل عمل می‌کند. این تابع هنگام حذف ماژول از کرنل فراخوانی می‌شود و اقدامات زیر را انجام می‌دهد:

- یک پیام لاگ ثبت می‌کند که نشان‌دهنده‌ی شروع فرایند پاکسازی ماژول است.
- با فراخوانی تابع free_tree، تمام گره‌های درخت قرمز-سیاه my_tree را حذف می‌کند و حافظه‌ی تخصیص‌یافته برای آن‌ها را آزاد می‌سازد.
- این کار باعث جلوگیری از نشت حافظه می‌شود و منابع سیستم را آزاد می‌کند.

تمرین ۱

پیاده‌سازی صف FIFO با استفاده از kfifo

- یک صف FIFO از اعداد صحیح با اندازه مشخص ایجاد کنید.
- چند مقدار به صف اضافه کنید و از پر شدن احتمالی صف جلوگیری کنید.
- تمام مقادیر را از صف خارج کرده و آن‌ها را در لاگ سیستم چاپ کنید.
- هنگام تخلیه ماژول، حافظه تخصیص داده شده به صف را آزاد کنید.

تمرین 2

استفاده از Spinlocks برای مدیریت منابع مشترک

- یک متغیر مشترک به نام shared_data ایجاد کنید.
- با استفاده از spinlock_t، دسترسی به این متغیر را از طریق چندین نخ مدیریت کنید.
- هنگام بارگذاری ماژول، مقدار متغیر را افزایش داده و هنگام تخلیه ماژول، آن را کاهش دهید.
- این عملیات باید تحت قفل Spinlock انجام شود.
- از printk برای چاپ مقدار shared_data قبل و بعد از هر تغییر استفاده کنید.