گزارش آزمایش ۴ درس سیستم عامل

محمدحسین نوروزی ۸۱۰۱۹۳۴۹۹

سیدعلی سادات اخوانی ۸۱۰۱۹۳۴۲۵

محمد رضایی ۸۱۰۱۹۱۵۱۳

بخش اول: توضیحات الگوریتم

در این بخش می‌خواهیم الگوریتم CFS را توضیح دهیم. اگر بخواهیم به صورت ساده این الگوریتم را توضیح دهیم، اگر چند پردازه اولویت برابر داشته باشند، این الگوریتم به هر کدام از آن‌ها زمان برابری از پردازنده اختصاص احتصاص می‌دهد.

struct sched_entity

هسته‌ی لینوکس زمان تخصیص یافته به هر پردازه را در ساختاری به نام sched_entity نگه‌داری می‌کند. در این struct مقداری به نام vruntime وجود دارد. این مقدار برابر است با زمانی که پردازه تا الان اجرا شده است. هسته‌ی لینوکس بر اساس این عدد مدت زمانی که هر پردازه در آینده اجرا خواهد شد را محاسبه می‌کند. هم‌چنین ساختاری بانام rb_node در آن وجود دارد که در بخش درخت قرمزسیاه بیشتر توضیح خواهیم داد.

update_curr & __update_curr

static void update_curr(struct cfs_rq *cfs_rq)

حال به سراغ تابع update_curr می‌رویم. این تابع مدت زمان اختصاص پردازنده به پردازه‌ی فعلی را کنترل می‌کند. ورودی این تابع اشاره‌گری به cfs_rq است که درخت پردازه‌ها (در ادامه توضیح داده خواهد شد) و همچین پزداره‌ی cur را نگه‌داری می‌کند. این تابع ابتدا مقدار delta_exec را محاسبه می‌کند که برابر است با مدت زمان اجرای پردازه. سپس تابع __update_curr را صدا می‌زند که براساس تعداد پردازه‌های درحال اجرا و اولویت هرکدام، سهم زمانی وزن‌دار پردازه را به وسیله‌ی calc_delta_fair مشخص می‌کند و مقدار vruntime آن را به‌روزرسانی می‌کند.

calc_delta_fair

حال به شرح تابع calc_delta_fair می‌پردازیم. مقدار بازگشتی براساس رابطه‌ی ساده‌ای، مقدار زمان وزن‌دار مخصوص هر پردازه را محاسبه می‌کند.

delta_exec * (NICE_0_LOAD / curr->load.weight)

در مخرج کسر دوم مقدار curr->load.weight قرار دارد که اصطلاحا عدد nice مربوط به هر پردازه است. این عدد نشانگر اولویت هر پردازه است. هر چه پردازه پراولویت‌تر باشد، این عدد کوچک‌تر است و هر چه این عدد کوچک‌تر باشد، کسر مقدار بزرگتری پیدا می‌کند. بنابراین زمان اختصاصی به هر پردازه یعنی delta_exec_weighted در پردازه‌های پراولویت‌تر بزرگ‌تر است.

Kernel Red-Black Tree

در هسته‌ی لینوکس برای تخصیص CPU به یک پردازه و انتخاب پردازه‌ي واجد شرایط از یک درخت قرمزسیاه استفاده می‌شود. درخت قرمزسیاه یک نوع درخت دودویی است که مشکل بالانس‌نبودن در آن با تدابیری رفع شده‌است. پس برای پیدا کردن هر node و درج وحدف آن‌ها پیچیدگی زمانی ما همیشه برابر logn است که این موضوع برای سرعت‌دادن به فرایند scheduling بسیار خوشایند است. Node های این درخت ساختارهای sched_entity هستند که بر اساس vruntime شان مرتب شده‌اند. پس هر

پردازه‌ای که vruntime کم‌تری دارد در سمت‌چپ ترین بخش این درخت قرار دارد. و می‌دانیم که هر پردازه‌ای vruntime کم‌تری دارد، زمان کم‌ترین به آن اختصاص پیداکرده است. پس مینیموم این درخت، شایستگی بیشتری برای به دست‌گرفتن CPU دارد.

توضیح ساختار درخت قرمزسیاه:
هر پردازه با ساختاری با نام task_struct نمایش داده می‌شود. در این ساختار برای هر پردازه ساختاری به نام sched_entity داریم که پیش‌تر توضیح داده شد. درون هر sched_entity ساختاری به نام rb_node داریم که دقیقا همان node های درخت هستند و دربرگیرنده‌ی رنگ node و اشاره‌گرهایی به node های چپ و راست خود می‌باشند. قطعا باید اشاره‌گری به ریشه‌ی درخت نیز در جایی بیرونی‌تر توسط هسته‌ي لینوکس نگه‌داری شود. این مقدار در ساختار rb_root که خود از مشخصه‌های ساختار cfs_rq می‌باشد نگه‌داری می‌شود و از طریق آن به درخت دسترسی داریم و اعمال لازم بر روی آن را انجام می‌دهیم. در ادامه به شرح دقیق‌تر فرآیند درج و حذف از درخت می‌پردازیم.

enqueue_entity

enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)

ورودی های این تابع به ترتیب برابر با اشاره‌گری به درخت مربوطه، اشاره‌گری به sched_entity جدید و flag برای مشخص‌کردن نوع آن است. این تابع زمانی فراخوانده می‌شود که پردازه‌ای fork و یا wake شده باشد. ابتدا مقدار vruntime مربوط به sched_entity ورودی را محاسبه می‌کند. (کمترین مقدار موجود در درخت) درادامه از طریق update_curr مقدار vruntime آخرین پردازه را به‌روزرسانی می‌کند. در نهایت node جدیدی که به آن پاس داده شده است به وسیله‌ی تابع زیر در درخت درج می‌شود.

enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)

این تابع اعمال زیر را انجام می‌دهد:
۱- ابتدا درخت را از ریشه پیمایش می‌کند و مکان درست قرارگرفتن node را پیدا می‌کند.
۲- با فراخوانی تابع rb_link_node ، مقدار ورودی به مکان مناسبش درج می‌شود
۳- با فراخوانی تابع rb_insert_color درخت قرمزسیاه با هزینه‌ی logn دوباره متوازن می‌شود.

dequeue_entity

dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int sleep)

ورودی‌های این تابع به ترتیب اشاره‌گری به درخت مربوطه، اشاره‌گری به sched_entity جدید و مقدار sleep است. این تابع زمانی فراخوانی می‌شود که که پردازه‌ای terminate و یا block شده‌باشد. در بدنه این تابع ابتدا مقادیر زمانی مربوط به آخرین پردازه (cfs_rq->curr) به روزرسانی می‌شود. در ادامه اگر پردازه‌ی ورودی برای با cur برابر نبود تابع زیر را صدا می‌زند و آن را حذف می‌کند.

dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)

این تابع کارهای زیر را انجام می‌دهد:
۱- با فراخوانی rb_erase نود sched_entity ورودی را حدف می‌کند.
۲- مقدار leftmost را به‌روزرسانی می‌کند.

بخش دوم: تغییرات کد

در این بخش به بررسی تغییرات موجود در کد می‌پردازیم. برای پیاده‌سازی سیاست fair-shair scheduling نیاز داریم که در ۳ فایل کرنل تغییر ایجاد کنیم که آن‌ها عبارتند از:

sched.h - sched_fair.h - user.h

۱ -ابتدا به بررسی تغییرات sched.h می‌پردازیم.

اصولاً در فایل sched.h تعریف متغیرها، structها و ساختمان داده‌های مربوط به تسک‌ها صورت می‌گیرد. در این‌جا ما برای پیاده‌سازی سیاست fair-shair scheduling در نظر گرفتیم که ۲ متغیر به سیستم اضافه کنیم.

- متغیر اول num_of_procs نام دارد که در struct user_struct قرار می‌گیرد که شامل تعداد پراسس‌هایی می‌شود که کاربر فعلی در حال اجراست.
- متغیر دوم نیز level نام می‌گیرد که در struct task_struct قرار می‌گیرد و نشان‌دهنده‌ی ارتفاع این پردازه در درخت پدر-فرزندی پردازه‌ها می‌باشد.

در عکس‌های زیر این تغییرات را مشاهده می‌کنید.

```
struct user_struct {
    atomic_t __count;    /* ref
    atomic_t processes; /* How
    atomic_t files;     /* How
    atomic_t sigpending;    /*
    unsigned int num_of_procs;
```

```
struct task_struct {
    volatile long state;
    void *stack;
    atomic_t usage;
    unsigned int flags;
    unsigned int ptrace;

    int lock_depth;
    unsigned int level;
```

۲ -حال به سراغ فایل user.c می‌رویم.

در این فایل که مربوط به user می‌باشد، باید متغیر جدیدی که در user_struct تعریف کرده بودیم را مقدار دهی اولیه کنیم. چون در ابتدای کار هنوز پردازه‌ای در اختیار این یوزر نیست، مقداردهی اولیه‌ی آن‌را به صفر می‌کنیم و به این موضوع نیز توجه می‌کنیم که این عملیات به صورت ATOMIC انجام بگیرد.

حال چیز دیگری که باید در این فایل تغییر دهیم، تابع alloc_uid است. این تابع به ازای uid که به آن داده شود، برای آن فضا allocate می‌کند. در یک بخش از این کد، مقداردهی برای new می‌شود که باید مقدار متغیر num_of_procs آن‌را نیز مقداردهی کنیم و برابر صفر قرار دهیم. شکل زیر:

```c
/* root_user.__count is 2, 1 for init t
struct user_struct root_user = {
    .__count    = ATOMIC_INIT(2),
    .processes  = ATOMIC_INIT(1),
    .files      = ATOMIC_INIT(0),
    .sigpending = ATOMIC_INIT(0),
    .locked_shm     = 0,
    .user_ns    = &init_user_ns,
    .num_of_procs   = ATOMIC_INIT(0),
};
```

```c
struct user_struct *alloc_uid(struct user_namespace *ns, uid_t uid)
{
    struct hlist_head *hashent = uidhashentry(ns, uid);
    struct user_struct *up, *new;

    /* Make uid_hash_find() + uid_hash_insert() atomic. */
    spin_lock_irq(&uidhash_lock);
    up = uid_hash_find(uid, hashent);
    spin_unlock_irq(&uidhash_lock);

    if (!up) {
        new = kmem_cache_zalloc(uid_cachep, GFP_KERNEL);
        if (!new)
            goto out_unlock;

        new->uid = uid;
        atomic_set(&new->__count, 1);
        new->num_of_procs=0;

        new->user_ns = get_user_ns(ns);
```

۳- مهم‌ترین بخش تغییرات کد در sched_fair.c رخ می‌دهد.

در این‌جا باید ۳ تابع را عوض کنیم. اسامی این ۳ تابع عبارتند از:

enqueue_entity - dequeue_entity - __update_curr

- در تابع enqueue_entity باید مقدار متغیر num_of_procs را یکی زیاد کنیم چون یک تسک به یوزرمان اضافه می‌شود و باید تعداد آن‌را یکی زیاد کنیم.

```c
//increasing user's num_of_procs because of enqueue
struct task_struct *p = container_of(se , struct task_struct , se);
p->cred->user->num_of_procs = p->cred->user->num_of_procs + 1;
```

- در تابع dequeue_entity باید مقدار متغیر num_of_procs را یکی کم کنیم چون یک تسک از یوزرمان کم می‌شود و باید تعداد آن‌را یکی کم کنیم.

```
//decreasing user's num_of_procs because of dequeue
struct task_struct *p = container_of(se , struct task_struct , se);
p->cred->user->num_of_procs = p->cred->user->num_of_procs - 1;
```

- در تابع update_cur__ که مهم‌ترین تغییرات در آن روی می‌دهد باید به نکات زیر توجه کرد.
اگر pid یک پردازه‌ای از ۵ بیشتر بود، level آن‌را این‌گونه مقداردهی می‌کنیم:

Level = لول پدر +۱

در غیر این صورت مقدار level آن‌را برابر یک قرار می‌دهیم.

با توجه به راهنمایی موجود در صورت پروژه، برای فرایندی که اولویت بالایی دارند و مقدار nice value آن‌ها کمتر از ۰ است نباید سیاست fair-shair پیاده شود. پس باید برای فرایندهایی که مقدار nice value آن‌ها بیش از صفر است این سیاست را اعمال کنیم.
نحوه‌ی اعمال آن نیز به این صورت است:

virtual runtime = مقدار وزن‌دار اجرا شده * level * num_of_procs

```
static inline void __update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
        unsigned long delta_exec)
{
    unsigned long delta_exec_weighted;
    struct task_struct *p = container_of(curr , struct task_struct , curr);

    if(p->pid < 5)
        p->level = 1;
    else
        p->level = p->real_parent->level + 1;

    schedstat_set(curr->exec_max, max((u64)delta_exec, curr->exec_max));

    curr->sum_exec_runtime += delta_exec;
    schedstat_add(cfs_rq, exec_clock, delta_exec);

    delta_exec_weighted = calc_delta_fair(delta_exec, curr);
    if(PRIO_TO_NICE((p)->static_prio) >= 0)
        curr->vruntime = curr->vruntime + delta_exec_weighted * p->cred->user->num_of_procs * p->level;
    else
        curr->vruntime = curr->vruntime + delta_exec_weighted;

    update_min_vruntime(cfs_rq);
}
```

بخش سوم: عملکرد تست‌ها

تست اول:
کد تست:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


int main(int argv, char *argc[]){
        printf("start\n");
        int forks = atoi(argc[1]);
        int i;
        int pid;

        for(i = 0; i<forks;i++){
                pid = fork();
                if(pid ==0)
                        break;
        }
        printf("i am process %d with pid %d (user: %s)\n", i, getpid(), argc[2])
;
        while(wait(NULL)>0);
        return 0;
}
~
~
                                                          17,14-21      All
```

جواب:

```
ali@ali:~/Downloads/4$ ./a.out 10 ali
start
i am process 0 with pid 1844 (user: ali)
i am process 1 with pid 1845 (user: ali)
i am process 2 with pid 1846 (user: ali)
i am process 3 with pid 1847 (user: ali)
i am process 4 with pid 1848 (user: ali)
i am process 5 with pid 1849 (user: ali)
i am process 7 with pid 1851 (user: ali)
i am process 6 with pid 1850 (user: ali)
i am process 10 with pid 1843 (user: ali)
i am process 8 with pid 1852 (user: ali)
i am process 9 with pid 1853 (user: ali)
```

تست دوم:
در اینجا باید ابتدا parent تمام شود و سپس child
این اتفاق را از طریق زمان اتمام می‌فهمیم و بررسی می‌کنیم
کد تست:

```c
#include <stdio.h>
#include <time.h>
#include <syscall.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/resource.h>
#include <sys/ipc.h>
#include <stdlib.h>
int main()
{
    pid_t pid;
    pid = fork();
    printf("pid: %d\n", pid);
    time_t time2, time1;
    long i, j;
    int c1, c2;
    int adder = 0;

    c1 = clock();
    time2 = time(NULL);

    if (nice(adder) == 1)
    {
        printf("Error\n");
        // return 0;
    }

    printf("prio: %d\n", getpriority(PRIO_PROCESS, 0));
    asaBusyWait(4);
    time1 = time(NULL);
    c2 = clock();

    if (pid != 0)
        printf("parent: %ld, %d\n", time2 - time1, c2 - c1);
    else
        printf("child: %ld, %d\n", time2 - time1, c2 - c1);
    if (pid != 0)
        printf("parent: %ld, %d\n", time2 - time1, c2 - c1);
    else
        printf("child: %ld, %d\n", time2 - time1, c2 - c1);
    return 0;
}

int asaBusyWait(double w)
{
    clock_t start, end;
    start = clock();
    end = clock();
    while(((double)(end - start)/CLOCKS_PER_SEC) < w)
        end = clock();
}
```

در این تست هم child و هم parent زمان اتمام خود را چاپ می‌کنند.
در خروجیِ ما مشکل وجود دارد و در بخش (nice(addr ما همیشه برای پراسس child مشکل داریم.

خروجی ما:

```
ali@ali:~/Downloads/4/gg$ ./a.out
pid: 1679
prio: 0
pid: 0
prio: 0
child: 0, 10000
parent: 0, 10000
```