

به نام خدا



دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده برق و کامپیوتر



سیستم عامل

گزارش آزمایش ۳ - آزمایشگاه سیستم عامل
زمان بندی پردازش ها

گروه ۱

علیرضا زارع نژاد
محمدضا عظیمی
امیرحسین عبادی



چرا فرخوانی sched منجر به فراخوانی scheduler می شود؟

برای وارد شدن به زمان بند فرایند ها تابع sched صدا زده می شود که در آن توسط تابع switch ابتدا وضعیت کنونی فرایند که در struct context ذخیره می شود و سپس کنترل به cpu scheduler داده می شود تا فرایند بعدی اجرا شود. سپس در تابع scheduler ابتدا از میان فرایندهای موجود اولین فرایندی که استیت آن runnable است را می یابیم و استیت آن را به runnig تغییر می دهیم و از cpu scheduler در هنگام setup در تابع main.c در فایل scheduler را فراخوانی می کند. این تابع در یک loop که همواره اجرا می شود یک process را انتخاب می کند و کنترل cpu را در اختیار آن قرار می دهد تا اجرا شود.

```
// Common CPU setup code.
static void
mpmain(void)
{
    cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
    idtinit();           // load idt register
    xchg(&(mycpu()->started), 1); // tell startothers() we're up
    scheduler();         // start running processes
}
```

```
// - eventually that process transfers control
// via switch back to the scheduler.
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        // Enable interrupts on this processor.
        sti();

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;

            // Switch to chosen process. It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchvm(p);
            p->state = RUNNING;

            switch(&(c->scheduler), p->context);
            switchkvm();

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            c->proc = 0;
        }
        release(&ptable.lock);
    }
}
```

در زمان بند کاملاً منصف در لینوکس صف اجرا چه ساختاری دارد؟

زمان بند لینوکس از ساختار red black tree استفاده می کند. در این درخت وزن هر نود virtual runtime

است. از آن جایی که در این درخت نود با کمترین وزن در سمت چپ قرار می گیرد پس همواره فرایند سمت چپ که دارای کمترین virtual runme است توسط scheduler انتخاب می شود.

دو سیستم عامل لینوکس و xv6 را از منظر مشترک و مجزا بودن صف های زمان بندی بررسی نمایید ؟

در سیستم عامل لینوکس به ازای هر cpu دو queue ایجاد می شود پس زمان بند هر هسته از دو صف تشکیل شده است. صف زمان بندی هسته ها نیز از یکدیگر مجزا است.

یک صف شامل active process ها و صف دیگر شامل process هایی است که expire شده اند. فرایند هایی که در صف expired هستند کوانتوم زمانی آن ها تمام شده است پس در صف expire قرار می گیرند تا زمانی که محتویات صف های active و expire با هم جا به جا شوند. یک دور صف active به طور کامل پیمایش می شود تا به صورت fair همه فرایندها در یک کوانتوم زمانی کار خود را انجام دهند. در سیستم عامل xv6 هر پردازنده یک صف زمان بندی دارد و از میان تمام پردازنده های موجود در ptable یکی را برای اجرا انتخاب می شود.

در هر اجرای حلقه ابتدای مدتی وقفه فعال می گردد. علت چیست ؟ آیا در سیستم های تک هسته ای به آن نیاز است ؟

گاهی اوقات ممکن است صف پردازنده های آماده خالی باشد به همین دلیل در ابتدای حلقه وقفه ها را فعال می کنیم تا اگر پردازنده ای به علت IO یا هر دلیل دیگری sleep است با آمدن وقفه بیدار شود و در صف ready قرار بگیرد.

در سیستم های تک هسته ای نیز به فعال شدن وقفه در صورت خالی بودن ready queue نیاز داریم.

تابع معادل Oscheduler را در هسته لینوکس بیابید. جهت حفظ اعتبار اطلاعات جدول پردازنده ها قفل گذاری می شود. این قفل در لینوکس چه نام دارد؟

<https://elixir.bootlin.com/linux/latest/source/kernel/sched/fair.c>

این قفل در لینوکس spinlock نام دارد.

در خصوص سازوکار توازن بار در زمان بند لینوکس به صورت مختصر توضیح دهید؟ این عملیات توسط چه موجودیتی و بر چه اساس صورت می گیرد؟

در سیستم های چند هسته ای برای این که الگوریتم زمانبندی درست کار کند ، باید بین صف های اجرا یک تعادلی برقرار باشد. به همین دلیل آنچه لینوکس و سایر schedulerها انجام می دهند این است که به صورت دوره ای یک الگوریتم توازن بار را اجرا می کنند که صف ها را تقریباً متعادل نگه می دارد.

یک الگوریتم توازن بار strawman ساده اطمینان می دهد که هر یک از صف های اجرا تقریباً تعداد thread های یکسان را دارند. بنابراین ، یک ایده دیگر برای ایجاد تعادل بین صف ها بر اساس وزن threadهاست ، و نه تعدادشان. در الگوریتم نهایی که مشلات دو الگوریتم قبلی را ندارد، CFS، نه تنها براساس وزن ، بلکه براساس معیار دیگری به نام load هم علاوه بر وزن، صف های اجرا را متوازن می کند. که این ترکیبی از همان وزن thread و متوسط CPU Utilization است. اگر thread زیاد از CPU استفاده نکند ، مقدار load آن کاهش می یابد. در نسخه های جدیدتر لینوکس ، برنامه ریز گروهی را برای گروه thread ها فراهم کرده که دو نوع اصلی آن، cgroup feature و autogroup feature هستند.

یک الگوریتم پایه ای توازن بار ، load کلیه هسته ها را مقایسه می کند و سپس taskها را از هسته با بیشترین load به کمترین منتقل می کند. این الگوریتم منجر به انتقال thread در داخل ماشین بدون در نظر گرفتن cache locality یا NUMA خواهد شد. در عوض ، توازن بار از یک استراتژی سلسله مراتبی استفاده می کند. چگونگی گروه بندی هسته ها در سطوح بعدی سلسله مراتب بستگی به نحوه اشتراک منابع فیزیکی ماشین دارد که به هر level یک scheduling domain میگوئیم. تعادل بار scheduling domain ریزی اجرا می شود ، از پایین به بالا شروع می شود. در هر سطح ، یک هسته از هر دامنه

مسئول متعادل کردن بار است. این هسته یا اولین هسته idle در scheduling domain است که می توان از چرخه های رایگان CPU آن برای تعادل بار استفاده کرد ، با اولین هسته در scheduling domain. پس از این ، میانگین بار برای هر گروه برنامه ریزی در هر scheduling domain محاسبه می شود. و busyترین گروه ، انتخاب می شود. اگر load شلوغ ترین گروه پایین تر از load گروه محلی باشد ، بار در این سطح متعادل در نظر گرفته می شود . در غیر این صورت ، بار بین CPU محلی و شلوغ ترین CPU گروه متعادل می شود. برای بهینه سازی برنامه ریز با اجرای الگوریتم توازن بار فقط روی هسته designated core برای scheduling domain داده شده، از انجام کار تکراری جلوگیری می کند. وقتی هر هسته فعال یک تیک ساعت بطور منظم دریافت می کند، شروع به اجرای الگوریتم تعادل بار می کند ، بررسی می کند که آیا پایین ترین هسته شماره گذاری شده در دامنه است یا اینکه پایین ترین idle شماره گذاری شده است . اگر این شرط برقرار بود ، هسته خود را به عنوان designated قلمداد می کند و به اجرای الگوریتم ادامه می دهد.

شرح پروژه:

برای پیاده سازی این صف های زمان بندی درون loop تابع scheduler را به این صورت تغییر می دهیم که ابتدا scheduler مربوط به صف اول را صدا می زنیم تا پردازش ای که برای اجرا شدن انتخاب کرده است را برگرداند و در choice قرار دهد. اگر صف یک خالی بود و پردازش ای انتخاب نشد همین عملیات را برای صف دوم انجام می دهیم و اگر صف دوم خالی بود به سراغ schedule کردن صف سوم می رویم. اگر باز هم صف سوم خالی بود continue می کنیم تا بالاخره یک پردازش برای اجرا شدن پیدا شود.

```

void
scheduler(void){
    struct proc *choice;
    struct cpu *c = mycpu();
    c->proc = 0;
    for(;;){
        // Enable interrupts on this processor.
        sti();
        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        choice = scheduler_q1();
        if(!choice)
            choice = scheduler_q2();
        if(!choice)
            choice = scheduler_q3();

        if(!choice)
        {
            release(&ptable.lock);
            continue;
        }

        // Switch to chosen process. It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
        c->proc = choice;
        switchvm(choice);
        choice->state = RUNNING;
        swtch(&(c->scheduler), choice->context);
        switchkvm();
        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
        release(&ptable.lock);
    }
}

```

:Lottery scheduling

در فایل `proc.h` فیلدهای `queue_number` و `tickets` را اضافه می کنیم. برای پیاده سازی این `scheduler` دو سیستم کال اضافه می کنیم.

`set_queue` : دو آرومان `pid` و `queue` را می گیرد و صف پردازنده ای که شنسه آن `pid` است را برابر `queue` قرار می دهد.

`set_tickets` : دو آرگومان `tickets` , `pid` را می گیرد و مقدار بلیت های پردازنده ای با شناسه `pid` را برابر با `tickets` قرار می دهد.

تغییرات با اضافه شدن فایل های `set_queue.c` و `set_tickets.c` در فایل های `defs.h`, `makefile` , `proc.c` , `syscall.c` , `syscall.h` , `sysproc.c` , `user.h` , `usys.s` قابل مشاهده است.

در تابع `scheduler_q1` بلیت تمام پردازنده هایی که `RUNNABLE` هستند را با هم جمع می کنم و در

متغیر tickets می ریزم. اگر pid پردازش برابر یک بود بدون چک کردن بلیت آن را بر می گردانم تا اجرا شود چون پردازش init باید حتما اجرا شود. سپس یک عدد رندوم تولید کرده و باقی مانده ی آن را به tickets به دست می آورم و دوباره در tickets قرار می دهم. تمام پردازش ها را می گردیم و بلیت پردازش هایی که در صف شماره یک هستند را به ترتیب از متغیر tickets کم می کنیم. اولین پردازش ای که این مقدار را منفی کند پردازش ای است که باید اجرا شود.

```
struct proc *
scheduler_q1(void){
    struct proc *choice;
    int sum_tickets = 0;
    rand = (a*rand + b) % 2147483647;

    for(choice = ptable.proc; choice < &ptable.proc[NPROC]; choice++){
        if(choice->state != RUNNABLE)
            continue;

        if(choice->pid == 1){
            return choice;
        }
        if(choice->queue_number == 1){
            sum_tickets += choice->tickets;
        }
    }
    if(sum_tickets){
        sum_tickets = rand % sum_tickets;
        // cprintf("t1 = %d\n", tickets);
    }
    for(choice = ptable.proc; choice < &ptable.proc[NPROC]; choice++){
        if(choice->state != RUNNABLE)
            continue;

        if(choice->queue_number == 1){
            sum_tickets -= choice->tickets;
            if(sum_tickets < 0)
                break;
        }
    }
    if(choice == &ptable.proc[NPROC])
        return 0;
    return choice;
}
```

: HRRN

در struct proc در فایل proc.h فیلد entry_time را اضافه می کنیم. برای مقدار دهی این فیلد در فایل proc.c تابع allocproc به ازای هر پردازش مقدار entry_time را برابر با ticks قرار می دهیم. Ticks متغیری است که به ازای هر سیستم یک واحد افزایش می یابد.

```

79 static struct proc*
80 allocproc(void)
81 {
82     struct proc *p;
83     char *sp;
84
85     acquire(&ptable.lock);
86
87     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
88         if(p->state == UNUSED)
89             goto found;
90
91     release(&ptable.lock);
92     return 0;
93
94 found:
95     p->state = EMBRYO;
96     p->pid = nextpid++;
97     p->priority = 10;
98     p->queue_number = 2;
99     p->tickets = 10;
100    p->execution_cycle_number = 1;
101    acquire(&tickslock);
102    p->arrival_time = ticks;
103    release(&tickslock);
104
105    if (p->pid == 2)
106    {
107        p->queue_number = 1;
108        p->tickets = 10000;
109    }
110
111    release(&ptable.lock);

```

```

393
394 struct proc * scheduler_q2(void){
395     struct proc* current;
396     struct proc* choice = 0;
397     acquire(&tickslock);
398     uint current_time = ticks;
399     release(&tickslock);
400     float max_response_ratio = 0.0;
401     float current_response_ratio;
402     uint waiting_time;
403
404     for (current = ptable.proc; current < &ptable.proc[NPROC]; ++current){
405         if (current->state != RUNNABLE)
406             continue;
407
408         if (current->pid == 1)
409             return current;
410
411         if (current->queue_number == 2){
412             waiting_time = current_time - current->arrival_time;
413             current_response_ratio = (float)waiting_time / (float)current->execution_cycle_number;
414
415             if (current_response_ratio >= max_response_ratio){
416                 max_response_ratio = current_response_ratio;
417                 choice = current;
418             }
419         }
420     }
421     if (choice != 0)
422         choice->execution_cycle_number += 0.1;
423
424     return choice;
425 }

```

④ There is an avail

در این زمان‌بند، پردازنده‌ها با توجه به اولویتی که دارند در سطوح مختلف قرار می‌گیرند که در این پروژه فرض شده است که سه سطح و متعاقباً سه اولویت وجود دارد. پردازنده‌هایی مثلاً معادل با پردازنده‌های ویرایش متن به طور پیش‌فرض دارای کمترین اولویت (اولویت ۳) هستند. شما برای آزمودن زمان‌بند خود باید فراخوانی سیستمی را پیاده کنید که بتواند اولویت پردازنده‌ها را تغییر دهد تا قادر به جابه‌جا کردن پردازنده‌ها در سطوح

و در اینجا کد زده شده آورده شده است.

```
7
8 struct proc * scheduler_q3(void){
9     struct proc* current;
10    struct proc* choice = 0;
11
12    float minimum_priority = INF;
13    for (current = ptable.proc; current < &ptable.proc[NPROC]; ++current){
14        if (current->state != RUNNABLE)
15            continue;
16
17        if (current->pid == 1)
18            return current;
19
20        if (current -> queue_number == 3){
21            if (current->priority < minimum_priority){
22                minimum_priority = current->priority;
23                choice = current;
24            }
25        }
26    }
27
28    if (choice != 0 && choice->priority >= 0.1)
29        choice->priority -= 0.1;
30    return choice;
31
32 }
```

حال برای تابع یک برنامه که همان foo.c است را ساخته و در back ground اجرا می‌کنیم و سپس با استفاده از توابع زده شده صحت عملکرد خود را می‌آزماییم. در اینجا تابع show_all_info برای چاپ اطلاعات پردازنده خواهد بود.

