

زمان بندی در xv6

1. چرا فراخوانی sched() منجر به فراخوانی scheduler() می شود؟

هر هسته ای که شروع به کار می کند، تابع `mpmain` را صدا می زند. این تابع نیز در انتها تابع `scheduler` را صدا می زند که باعث شروع به کار زمان بند مربوط به هر هسته می شود. تابع ذکر شده در `ptable` به دنبال پردازش قابل اجرا می گردد و در صورت یافتن چنین پردازشی (`RUNNABLE`)، پس از تغییر حافظه به حافظه پردازش توسط تابع `switchvm`، با استفاده از تابع `swtch` که در زبان اسمبلی پیاده سازی شده، عملیات تعویض متن را انجام می دهد. این تابع، رجیسترهای `context` قدیمی (`cpu::struct context *scheduler`) را در آدرس مربوط به همان `context` ذخیره می کند و رجیسترهای مربوط به `context` جدید را از آدرس مربوط به همان `context` بازیابی می کند که با این کار، `program counter` نیز به مقدار متناظر آن در `context` جدید تبدیل می شود و به این ترتیب، پردازش جدید شروع به اجرا می کند. در 3 حالت زیر، پردازش در حال اجرا تابع `sched` را فراخوانی می کند:

- 1- پردازش با استفاده از فراخوانی سیستمی `exit`، پردازنده را ترک کند.
 - 2- پردازش با استفاده از فراخوانی سیستمی `sleep`، به حالت `SLEEPING` در آید.
 - 3- پس از `interrupt` ایجاد شده توسط تایمر، پردازش مجبور به خروج از پردازنده شود که در این حالت تابع `yield` فراخوانی شده و در آن تابع نیز تابع `sched` فراخوانی می شود.
- در نهایت در تابع `sched`، مجدداً عملیات تعویض متن صورت می پذیرد و در این حالت `context` ای که در استراکت `cpu (struct context *scheduler)` بازیابی می شود و `context` مربوط پردازش در حال اجرا ذخیره می شود. پس از بازیابی `context` مربوط به `scheduler`، `program counter` به خط 2782 اشاره می کند و باعث ادامه کار `scheduler` می شود. در واقع پردازشی که هر هسته را آماده به کار می کند، هیچ وقت از تابع `scheduler` خارج نمی شود و فقط با عملیات `context switching` از پردازنده خارج می شود و با اجرای تابع `sched`، دوباره به کار خود می پردازد.
- لازم به ذکر است که `program counter` به صورت مستقیم ذخیره نمی شود بلکه همان `return adr` تابع است که در زمان فراخوانی تابع `swtch`، در استک `push` می شود. این آدرس پس از دستور `ret` در خط 3078، از استک `pop` شده و در رجیستر مربوط به `program counter` قرار می گیرد.

زمان بندی

2. ساختار صف اجرا در زمان بند کاملاً منصف لینوکس

صف اجرا در لینوکس توسط یک `red-black tree` پیاده سازی می شود. در چپ ترین گره این درخت، پردازشی قرار گرفته که کمترین برش زمانی در حین اجرا را داشته است. (این مقدار در `vruntime` اطلاعات پردازش ذخیره شده است)

3. بررسی لینوکس و xv6 از منظر مشترک یا مجزا بودن صف های زمان بندی

هر پردازنده زمان بند خودش را دارد ولی همانطور که در شکل دیده می شود، در xV6 فقط از یک صف زمان بندی برای همه پردازنده ها به طور مشترک استفاده می شود:

```
struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```

این صف از `struct proc` ها حداکثر `NPROC` (64) پردازنده همزمان را می تواند در خود نگه دارد. برای جلوگیری از خرابی حاصل از تغییرات همزمان چند پردازنده روی این صف، از یک `spinlock` استفاده می شود. به این صورت که هنگام دسترسی به `ptable.proc` ابتدا باید `ptable.lock` را `acquire` کنیم و پس از انجام تغییرات در آن، آن را `release` می کنیم. داشتن فقط یک صف برای همه پردازنده ها، پیاده سازی را کمی ساده تر می کند ولی در عوض به `lock` نیاز دارد که می تواند کمی بر روی `performance` نیز تأثیر بگذارد. از آنجا که پردازهای که در این صف است، هر بار در یک پردازنده اجرا می شود و بین آنها جهش می کند، با توجه به اینکه هر پردازنده `high-level cache` خودش را دارد، کارایی `cache` بسیار کمتر می شود. در لینوکس، هر پردازنده صف زمان بندی مخصوص خودش را دارد و پردازنده ها به صورت مجزا در آنها قرار می گیرند. این طراحی نیازمند `load balancing` است یعنی توازن در همه صف های پردازنده ها برقرار باشد و یک پردازنده خالی و دیگری پر نماند. نیازی به این کار در صف مشترک نیست.

4. چرا در اجرای حلقه ابتدا وقفه فعال می گردد؟ آیا در سیستم تک هسته ای به آن نیاز است؟

زمانی که قفل `ptable` فعال می شود، تمامی `interrupt` ها به وسیله تابع `pushcli` غیرفعال می شوند. حال ممکن است پردازنده در حالتی قرار بگیرد که تعدادی از پردازنده های آن منتظر پایان عملیات I/O باشند و هیچ کدام از پردازنده های دیگر نیز در حالت `RUNNABLE` نباشند. در این حالت هیچ پردازنده دیگری اجرا نمی شود و اگر `interrupt` ها نیز هیچ وقت فعال نشود، پس از پایان عملیات I/O نمی توانیم پردازنده های مربوطه را به حالت `RUNNABLE` تغییر دهیم که بتوانند اجرا شوند، در نتیجه سیستم فریز می شود. به همین دلیل است که در این حلقه برای مدت کوتاهی (تا پیش از قفل کردن `ptable`)، وقفه ها فعال می شوند تا در صورت نیاز بتوانیم حالت پردازنده ها را تغییر دهیم.

5. دو سطح مدیریت وقفه ها در لینوکس

مدیریت وقفه ها در لینوکس و بسیاری از سیستم عامل های امروزی، در دو سطح اول و دوم صورت می گیرد. به این دو سطح `FLIH`¹ و `SLIH`² گفته می شود. علاوه بر آن در لینوکس به `FLIH`، نیمه بالایی³ و به `SLIH`، نیمه پایینی⁴ گفته می شود. وظیفه `FLIH` مدیریت وقفه های ضروری در کمترین زمان ممکن است؛ یا وقفه را به طور کامل سرویس دهی می کند و یا اطلاعات ضروری وقفه را - که فقط در زمان وقوع وقفه در دسترسی است - ذخیره کرده و یک `SLIH` را برای مدیریت کامل این وقفه زمان بندی می کند. در روال جواب دهی به وقفه ها

¹ First-Level Interrupt Handler

² Second-Level Interrupt Handler

³ Upper half

⁴ Lower half or bottom half

در FLIH، یک تعویض متن¹ صورت گرفته و کد مربوط به مدیریت‌کننده وقفه صورت‌گرفته بارگزاری و اجرا می‌شود. FLIH می‌تواند باعث ایجاد لغزش² (لگ) در پردازنده‌ها شود. علاوه بر آن FLIH باعث چشم‌پوشی³ کردن از وقفه‌ها می‌شود.

SLIH وظیفه بخش‌هایی از پردازش وقفه‌ها را بر عهده دارد که زمان‌بر می‌باشند؛ این کار مانند یک پردازنده انجام می‌شود. SLIH‌ها یا یک ریسسه مخصوص در سطح کرنل برای هر هندلر دارند، یا توسط یک thread pool مدیریت می‌شوند. SLIH‌ها در یک صف اجرا قرار گرفته و منتظر پردازنده می‌مانند. از آنجا که ممکن است زمان طولانی برای اجرای آن‌ها نیاز باشد، SLIH‌ها نیز معمولاً مانند ریسسه‌ها و پردازنده‌ها زمان‌بندی می‌شوند.

6. گرسنگی پردازنده‌ها چگونه حل شده است؟

گاه‌ها ممکن است پردازنده‌ای، به دلیل داشتن اولویت کمتر نسبت به باقی پردازنده‌های در حال اجرا، مدت نامشخصی را در صف آماده‌سپری کند؛ به این اتفاق گرسنگی پردازنده گفته می‌شود. برای حل این موضوع راهکار aging ارائه شده است؛ Aging بدین معنی است که هر چه یک پردازنده با اولویت کمتر در صف اجرا باقی بماند، اولویت آن به مرور زمان افزایش پیدا می‌کند؛ مثلاً اگر اولویت‌ها از صفر تا 127 شماره‌گذاری شده باشند و یک پردازنده با اولویت صفر (کم‌ترین اولویت ممکن) در صف اجرا موجود باشد، پس از مدت مشخصی، شماره اولویت را یک واحد افزایش می‌دهیم. این کار تا زمانی که پردازنده به پردازنده اختصاص یابد، ادامه می‌یابد. علاوه بر آن، راهکارهای زیر به منظور جلوگیری از اختصاص بیش از حد پردازنده به وقفه‌ها نیز در برخی سیستم‌ها اجرا شده‌اند:

- کوتاه و سریع نگه داشتن مدیریت وقفه‌ها؛ برای مثال استفاده از FLIH و SLIH که در بالاتر توضیح داده شد، راهکار مناسبی برای این مورد می‌باشد.
- محدود کردن نرخ ایجاد وقفه‌ها
- کم کردن worst-case نرخ ایجاد وقفه‌ها؛ این کار در سطح دستگاه‌هایی که باعث ایجاد وقفه می‌شوند صورت می‌گیرد.
- چک کردن دوره‌ای اتفاقات⁴ بجای استفاده از وقفه‌ها

زمان‌بندی بازخوردی چند سطحی

در ابتدا یک استراکت به نام schedinfo به فیلدهای استراکت proc اضافه کردیم تا تمامی متغیرهای مربوط به زمان‌بندی یک پردازنده در این استراکت قرار بگیرد. یکی از فیلدهای این استراکت، یک enum به نام schedqueue است که نشان می‌دهد پردازنده مربوطه در کدام صف قرار دارد. برای بخش aging تابع زیر نوشته شد:

¹ Context switch

² Jitter

³ Mask

⁴ Polling for events

```

void
ageprocs(int osTicks)
{
    struct proc *p;
    acquire(&ptable.lock);

    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state == RUNNABLE && p->sched_info.queue != ROUND_ROBIN)
        {
            if (osTicks - p->sched_info.last_run > AGING_THRESHOLD)
            {
                release(&ptable.lock);
                change_queue(p->pid, ROUND_ROBIN);
                acquire(&ptable.lock);
            }
        }
    }

    release(&ptable.lock);
}

```

این تابع پس از هر بار افزایش مقدار `ticks` در `interrupt` مربوط به تایمر در فایل `trap.c`، فراخوانی می‌شود. لازم به ذکر است که مقدار `p->sched_info.last_run` پس از هر بار زمان‌بندی در تابع `scheduler` برای پردازهای که قرار است اجرا شود، مقداردهی می‌شود.

1. زمان‌بند نوبت‌گردشی

برای این زمان‌بند تابع زیر را به سیستم عامل اضافه کردیم. لازم به ذکر است که پارامتر `lastScheduled` در واقع آخرین پردازهای است که توسط الگوریتم Round-Robin زمان‌بندی شده و این الگوریتم، از پردازش بعد از `lastScheduled` در صف، به دنبال پردازهای `RUNNABLE` می‌گردد و در صورت یافتن همچنین پردازهای، آن را زمان‌بندی می‌کند.

```

struct proc*
roundrobin(struct proc *lastScheduled)
{
    struct proc *p = lastScheduled;
    for (;;)
    {
        p++;
        if (p >= &ptable.proc[NPROC])
            p = ptable.proc;

        if (p->state == RUNNABLE && p->sched_info.queue == ROUND_ROBIN)
            return p;

        if (p == *lastScheduled)
            return 0;
    }
}

```

2. زمان‌بند بخت آزمایی

تابع `lottery` برای پیاده‌سازی این زمان‌بند ایجاد شده است. این تابع ابتدا یک عدد تصادفی ایجاد کرده و سپس در جدول پدازه‌ها (`ptable.proc`) به دنبال پدازه‌هایی می‌گردد که در صف `LOTTERY` و دارای حالت `RUNNABLE` می‌باشد. هر پدازه دارای یک مقدار `tickets_count` بوده که نشان‌دهنده تعداد بلیط‌های تخصیص یافته به آن پدازه است. برای آنکه پدازه برنده مشخص شود، ابتدا مجموع تعداد بلیط‌ها محاسبه می‌شود و یک عدد تصادفی بین صفر و این مجموع تولید می‌شود. سپس در هر مرحله یک متغیر به نام `prev_interval_begin` را در نظر می‌گیریم که نشان‌دهنده تعداد بلیط‌ها تا آن مرحله از جستجو می‌باشد. در صورتی که عدد تصادفی بین `prev_interval_begin` و آن به علاوه تعداد `ticket` پدازه کنونی باشد، آن پدازه انتخاب می‌شود. به عبارتی دیگر از جمع انباشته بلیط‌ها برای انجام بخت‌آزمایی استفاده شده است. کد این تابع به صورت زیر می‌باشد:

```
struct proc*
lottery() {
    struct proc* result = 0;

    uint tickets_sum = 0;
    for (int i = 0; i < NPROC; ++i) {
        if ((ptable.proc[i].state == RUNNABLE) &&
            (ptable.proc[i].sched_info.queue == LOTTERY)) {
            tickets_sum += ptable.proc[i].sched_info.tickets_count;
        }
    }
    if (tickets_sum == 0)
        return result;

    uint ticket = rand() % tickets_sum;

    uint prev_interval_begin = 0;
    for (int i = 0; i < NPROC; ++i) {
        if (ptable.proc[i].state != RUNNABLE)
            continue;
        if (ptable.proc[i].sched_info.queue != LOTTERY)
            continue;
        if (
            (ticket >= prev_interval_begin) &&
            (ticket <= prev_interval_begin +
             ptable.proc[i].sched_info.tickets_count)
        ) {
            result = &ptable.proc[i];
            break;
        }
        prev_interval_begin +=
            ptable.proc[i].sched_info.tickets_count;
    }

    return result;
}
```

3. زمان‌بند اول بهترین کار

تابع `bestjobfirst` این نوع زمان‌بندی را انجام می‌دهد. برای پیاده‌سازی، ابتدا فیلدهای 3 معیار و ضریب متناظر با هر کدام به پردازش اضافه شده است. فیلد `priority` در اینجا عددی از 1 تا 5 است که نشان دهنده اولویت آن پردازش است (1 بالاترین اولویت). در تابع زمان‌بند، بین همه پردازش‌های `runnable` که نوع صف آنها BJB است، Rank کمینه حساب می‌شود و آن به عنوان پردازش برای اجرا انتخاب می‌شود.

```
static float
bjfrank(struct proc* p)
{
    return p->sched_info.bjf.priority *
           p->sched_info.bjf.priority_ratio +
           p->sched_info.bjf.arrival_time *
           p->sched_info.bjf.arrival_time_ratio +
           p->sched_info.bjf.executed_cycle *
           p->sched_info.bjf.executed_cycle_ratio;
}

struct proc*
bestjobfirst(void)
{
    struct proc* result = 0;
    float minrank;

    struct proc* p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE || p->sched_info.queue != BJB)
            continue;
        float rank = bjfrank(p);
        if(result == 0 || rank < minrank){
            result = p;
            minrank = rank;
        }
    }

    return result;
}
```

فراخوانی‌های سیستمی

1. تغییر صف پردازش

این فراخوانی سیستمی با نام `change_scheduling_queue` به سیستم عامل اضافه شده است. تابع نهایی مربوط به این فراخوانی سیستمی (در فایل `proc.c`) در بخش زیر قابل مشاهده است:

```

int
change_queue(int pid, int new_queue) {
    struct proc *p;
    int old_queue = -1;
    if (new_queue == UNSET)
    {
        if (pid == 1)
            new_queue = ROUND_ROBIN;
        else if (pid > 1)
            new_queue = LOTTERY;
        else
            return -1;
    }

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            old_queue = p->sched_info.queue;
            p->sched_info.queue = new_queue;
            release(&ptable.lock);
            return old_queue;
        }
    }
    release(&ptable.lock);
    return old_queue;
}

```

2. مقداردهی بلیت بخت آزمایی

این فراخوانی سیستمی با نام `set_lottery_tickets` پیاده‌سازی شده است. کد مربوط به آن در زیر قابل مشاهده است:

```

int
set_lottery_ticket(int pid, int tickets) {
    struct proc* p;

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if ((p->pid == pid) && (p->sched_info.queue == LOTTERY)) {
            p->sched_info.tickets_count = tickets;
            release(&ptable.lock);
            return 1;
        }
    }
    release(&ptable.lock);
    return -1;
}

```

3. مقداردهی پارامتر BJJ در سطح پردازش

تابع سیستمی این کار با نام `set_bjfq_params_process` که به عنوان ورودی PID پردازش مورد نظر و 3 ضریب BJF را می‌گیرد مطابق کد زیر تعریف شده است:

```
int
set_bjfq_params_process(int pid, float priority_ratio, float
arrival_time_ratio, float executed_cycles_ratio)
{
    acquire(&ptable.lock);
    struct proc* p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            p->sched_info.bjfq.priority_ratio = priority_ratio;
            p->sched_info.bjfq.arrival_time_ratio = arrival_time_ratio;
            p->sched_info.bjfq.executed_cycle_ratio =
executed_cycles_ratio;
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
```

4. مقداردهی پارامتر BJF در سطح سیستم

تابع سیستمی این کار با نام `set_bjfq_params_system` که به عنوان ورودی 3 ضریب BJF را می‌گیرد و آن را برای همه پردازش‌ها تنظیم می‌کند، مطابق کد زیر تعریف شده است:

```
void
set_bjfq_params_system(float priority_ratio, float
arrival_time_ratio, float executed_cycles_ratio)
{
    acquire(&ptable.lock);
    struct proc* p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        p->sched_info.bjfq.priority_ratio = priority_ratio;
        p->sched_info.bjfq.arrival_time_ratio = arrival_time_ratio;
        p->sched_info.bjfq.executed_cycle_ratio =
executed_cycles_ratio;
    }
    release(&ptable.lock);
}
```

5. چاپ اطلاعات

در این فراخوانی سیستمی، روی همه پردازش‌ها پیمایش می‌شود و همه مشخصات خواسته شده پرینت می‌شوند:


```

void
print_process_info()
{
    static char *states[] = {
        [UNUSED]    "unused",
        [EMBRYO]     "embryo",
        [SLEEPING]   "sleeping",
        [RUNNABLE]   "runnable",
        [RUNNING]    "running",
        [ZOMBIE]     "zombie"
    };

    static int columns[] = {16, 8, 9, 8, 8, 8, 8, 9, 8, 8, 8, 8};
    cprintf("Process_Name   PID      State   Queue   Cycle
Arrival Ticket  Priority R_PrtY  R_Arvl  R_Exec  Rank\n"
           "-----\n");

    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state == UNUSED)
            continue;

        const char* state;
        if(p->state >= 0 && p->state < NELEM(states) && states[p-
>state])
            state = states[p->state];
        else
            state = "???";

        cprintf("%s", p->name);
        printspaces(columns[0] - strlen(p->name));
        ...
    }
}

```

برنامه سطح کاربر

در ابتدا یک برنامه سطح کاربر برای اجرای فراخوانی‌های سیستمی نوشته شده ایجاد کردیم. نحوه استفاده از این برنامه سطح کاربر به نام `schedule` به شکل زیر است:

```

$ schedule
usage: schedule command [arg...]
Commands and Arguments:
info
set_queue <pid> <new_queue>
set_tickets <pid> <tickets>
set_process_bjf <pid> <priority_ratio> <arrival_time_ratio> <executed_cycle_ratio>
set_system_bjf <priority_ratio> <arrival_time_ratio> <executed_cycle_ratio>
set_priority_bjf <pid> <priority>

```

برنامه سطح کاربر دیگری به نام `foo` نیز نوشته شده که 5 پردازش می‌سازد که هر کدام پس از مدتی `sleep` کردن، شروع به انجام محاسباتی طولانی می‌کنند.

```

#include "types.h"
#include "user.h"

#define PROCS_NUM 5

int main()
{
    for (int i = 0; i < PROCS_NUM; ++i)
    {
        int pid = fork();
        if (pid > 0)
            continue;
        if (pid == 0)
        {
            sleep(5000);
            for (int j = 0; j < 100 * i; ++j)
            {
                int x = 1;
                for (long k = 0; k < 1000000000000; ++k)
                    x++;
            }
            exit();
        }
    }
    while (wait() != -1)
        ;
    exit();
}

```

\$ schedule info											
Process_Name	PID	State	Queue	Cycle	Arrival	Ticket	Priority	R_Prt	R_Arvl	R_Exec	Rank
init	1	sleeping	1	1	0	0	3	1	1	1	4
sh	2	sleeping	2	1	4	6	3	1	1	1	8
schedule	3	running	2	1	254	10	3	1	1	1	258
\$ foo&											
\$ schedule info											
Process_Name	PID	State	Queue	Cycle	Arrival	Ticket	Priority	R_Prt	R_Arvl	R_Exec	Rank
init	1	sleeping	1	1	0	0	3	1	1	1	4
sh	2	sleeping	2	2	4	6	3	1	1	1	9
foo	6	sleeping	2	44	473	8	3	1	1	1	520
foo	5	sleeping	2	1	472	3	3	1	1	1	476
foo	7	sleeping	2	44	473	6	3	1	1	1	520
foo	8	sleeping	2	44	473	4	3	1	1	1	520
foo	9	sleeping	2	44	474	3	3	1	1	1	521
foo	10	sleeping	2	44	474	1	3	1	1	1	521
schedule	11	running	2	0	912	4	3	1	1	1	915
\$ schedule info											
Process_Name	PID	State	Queue	Cycle	Arrival	Ticket	Priority	R_Prt	R_Arvl	R_Exec	Rank
init	1	sleeping	1	1	0	0	3	1	1	1	4
sh	2	sleeping	2	2	4	6	3	1	1	1	9
schedule	12	running	2	0	2728	8	3	1	1	1	2731
foo	5	sleeping	2	1	472	3	3	1	1	1	476
foo	8	running	2	193	473	4	3	1	1	1	669
foo	9	runnable	2	180	474	3	3	1	1	1	657
foo	10	runnable	2	99	474	1	3	1	1	1	576