

آزمایشگاه سیستم عامل

پروژه سه

اعضای گروه:

الهه خداوردی - 810100132

فرشته باقری - 810100089

عاطفه میرزاخانی - 810100220

Repository: <https://github.com/elahekhodaverdi/Operating-System-Lab-Projects>

زمان بندی در xv6:

1) چرا فراخوانی تابع sched(), منجر به فراخوانی تابع scheduler() می‌شود؟

هر پردازنده که شروع به کار می‌کند چه از طریق main() (پردازنده bootstrap) و چه از طریق mpenter() (پردازنده های دیگر) در نهایت تابع mpmain() را صدا می‌زنند که این تابع نیز در نهایت تابع scheduler را صدا می‌زند. در این تابع در ptable به دنبال پردازنده RUNNABLE می‌گردیم و پس از پیدا شدن پردازنده با استفاده از تابع switchvm حافظه را به حافظه پردازنده تغییر می‌دهیم. این تابع نیز با استفاده از کد اسمبلی مربوط به switch عملیات سوییچ را انجام می‌دهد.

حال به این مسئله می‌پردازیم که چگونه تابع sched باعث فراخوانی scheduler می‌شود. پردازنده در 3 حالت تابع sched را فراخوانی می‌کند.

حالت اول: هنگامی که با استفاده از تابع exit به اجرای خود پایان می‌دهد.

حالت دوم: هنگامی که برای رفتن به حالت SLEEPING، تابع sleep را فراخوانی می‌کند.

حالت سوم: پس از گذراندن دوره پردازش خود با ایجاد interrupt توسط timer تابع yield را فراخوانی می‌کند.

در هر سه تابع بالا در نهایت sched فراخوانی شده، حال با توجه به کد sched می‌بینیم که در واقع پس از فراخوانی در این تابع، این تابع context پردازنده فعلی را با context مربوط به scheduler سوییچ می‌کند و این باعث می‌شود که در واقع تابع scheduler دوباره از سر گرفته شود.

در واقع پردازنده ای که هر هسته را از آماده می‌کند هیچوقت از تابع scheduler خارج نمی‌شود و فقط در هنگام اجرای پردازنده‌ها با دستور context switching از پردازنده خارج و با sched دوباره از سر گرفته می‌شود.

(2) صف پردازیهایی که تنها منبعی که برای اجرا کم دارند پردازه است، صف آماده یا صف اجرا نام دارد. در xv6 صف آماده مجزا وجود نداشته و از صف پردازها بدین منظور استفاده می‌گردد. در زمان‌بند کاملاً منصفانه در لینوکس صف اجرا چه ساختاری دارد؟

در زمان‌بندی CFS در لینوکس از ساختار داده red-black tree استفاده می‌شود. هر پردازه قابل اجرا یک زمان مجازی مربوط به خود دارد که در PCB آن پردازه ذخیره می‌شود. هر زمان که context switch رخ می‌دهد، زمان مجازی فرآیند در حال اجرا با مقدار زمانی که اخیراً اجرا شده است، افزایش می‌یابد. تمام فرآیندهایی که در حافظه اصلی هستند، در red-black tree درج می‌شوند و هرگاه فرآیند جدیدی وارد می‌شود، در این درخت درج می‌شود. همانطور که می‌دانیم، درخت‌های سرخ-سیاه درختان جستجوی دودویی خودتنظیم‌شونده هستند. زمانی که باید یک فرآیند جدید برنامه‌ریزی شود، برنامه‌ریز فرآیندی را با کمترین زمان اجرای مجازی از درخت red-black انتخاب می‌کند. این اطمینان می‌دهد که هر فرآیند سهم عادلانه‌ای از زمان CPU را دریافت می‌کند.

(3) همانطور که در پروژه اول مشاهده شد، هر هسته پردازنده در xv6 یک زمان‌بند دارد. در لینوکس نیز به همین گونه است. این دو سیستم عامل از منظر مشترک یا مجزا بودن صف‌های زمان‌بندی بررسی نمایید و یک مزیت و یک نقص صف مشترک نسبت به صف مجزا را بیان کنید.

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

در سیستم عامل xv6 همه پردازها در یک صف یکسان قرار دارند و صف‌های جداگانه‌ای برای آن‌ها در نظر گرفته نشده است. همه پردازها صرف نظر از state خود درون ptable قرار گرفته‌اند و به وسیله آن پردازها را مدیریت می‌کند و همه پردازنده‌ها از این صف به طور مشترک استفاده می‌کنند. این struct از یک لیست پردازها تشکیل شده که حداکثر NPROC پردازه را می‌تواند درون خود نگه دارد و همچنین یک spinlock نیز دارد که برای جلوگیری از conflict ها باید قبل از دستکاری و دسترسی به proc آن را require کنیم و پس از تغییر آن را release کنیم.

در لینوکس هر پردازنده‌ها صف زمان‌بندی خود را دارند و پردازها به صورت مستقل و مجزا در هر کدام آن‌ها قرار می‌گیرد.

مزایا و معایب:

داشتن یک صف برای تمام پردازنده‌ها پیاده‌سازی را آسان‌تر می‌کند و در عوض نیازمند lock است که می‌تواند روی بارده تاثیر داشته باشد. و به دلیل اینکه پردازه در این صف در پردازنده‌های متفاوتی اجرا می‌شود و هربار بین آن‌ها جهش میکند با توجه به اینکه هر پردازنده cache خود را دارد کارایی cache کم می‌شود.

از آن سمت اگر صف ها مجزا باشند ممکن است پردازنده ها به صورت متعادل در اختیار پردازنده ها قرار نگیرند و این ممکن است سبب شود که در حالی که یک پردازنده پردازنده های زیادی برای اجرا در صف خود دارد یک پردازنده دیگر بیکار باشد و صفش خالی باشد.

4) در هر اجرای حلقه، ابتدا برای مدتی وقفه فعال می گردد. علت چیست؟ آیا در سیستم تک هسته به آن نیاز است؟

زمانی که قفل ptable را به وسیله require فعال می کنیم تمام interrupt ها با فراخوانی تابع pushcli در تابع require غیرفعال می شوند. ممکن است پردازنده در حالتی قرار بگیرد که تعدادی از پردازنده های آن منتظر دستگاه های ورودی و خروجی باشند و دیگر پردازنده ها نیز قابل اجرا نباشند و در نتیجه هیچ یک اجرا نخواهند شد. حتی پس از این عملیات ها نیز اگر هنوز interrupt ها غیر فعال باشند دیگر هیچ پردازنده ای نمی تواند وارد پردازنده شود و اجرا شود و سیستم فریز خواهد شد. به همین دلیل برای اطمینان از enable بودن interrupt ها در ابتدای هر حلقه لازم است که sti را فراخوانی کنیم تا از این مسائل جلوگیری شود.

5) وقفه ها اولویت بالاتری نسبت به پردازنده ها دارند. به طور کلی مدیریت وقفه ها در لینوکس در دو سطح صورت می گیرد. آن ها را نام برده و به اختصار توضیح دهید.

در لینوکس، مدیریت وقفه به دو بخش تقسیم می شود: "نیمه بالایی" و "نیمه پایینی".

1. **نیمه بالایی:** این بخش مهم و اصلی مدیریت وقفه ها است که به محض دریافت وقفه توسط CPU اجرا می شود. در زمان اجرای نیمه بالایی، وقفه ها و برنامه ریز غیرفعال هستند. این بخش فقط شامل critical code است. کار نیمه بالایی این است که اجرا شود، هرگونه وضعیت لازم را ذخیره کند، برای اجرای نیمه پایینی ترتیب دهد، سپس return کند. نیمه بالایی کار حداقلی انجام می دهد، معمولاً با سخت افزار ارتباط برقرار می کند و یک پرچم را در جایی در حافظه هسته تنظیم می کند.

2. **نیمه پایینی:** نیمه پایینی بیشتر کار را انجام می دهد. این بخش بیشتر پردازش وقفه را انجام می دهد، اما بدون اینکه سیستم را متوقف کند. همه وقفه ها در زمان اجرای نیمه پایینی فعال هستند. به همین دلیل در زمان های امن اجرا می شود. در سناریوی معمول، نیمه بالایی داده های دستگاه را در یک بافر خاص دستگاه ذخیره می کند، برای اجرای نیمه پایینی زمان بندی می کند و خارج می شود.

این روش دو سطحی برای مدیریت وقفه طراحی شده است تا زمان صرف شده در بخش حیاتی کد (نیمه بالایی) را که در آن وقفه ها غیرفعال هستند، به حداقل برساند. این امکان می دهد تا سیستم به سرعت به وقفه های جدید پاسخ دهد، در حالی که همچنان امکان پردازش پیچیده را در پاسخ به وقفه ها فراهم می کند.

6) مدیریت وقفه‌ها در صورتی که بیش از حد زمان بر شود، می‌تواند منجر به گرسنگی پردازنده‌ها گردد. این می‌تواند به خصوص در سیستم‌های بی‌درنگ مشکل ساز باشد. چگونه این مشکل حل شده است؟

مشکل اصلی الگوریتم‌های زمان‌بندی اولویت، گرسنگی است. فرآیندی که آماده اجرا است اما منتظر CPU است را می‌توان مسدود شده در نظر گرفت. یک الگوریتم زمان‌بندی اولویت می‌تواند برخی از فرآیندهای با اولویت پایین را به طور نامحدود در انتظار باقی بگذارد.

یک راه حل برای مشکل گرسنگی پردازنده‌ها، aging است. aging شامل افزایش تدریجی اولویت فرآیندهایی است که برای مدت طولانی در سیستم منتظر می‌مانند. به عنوان مثال، اگر اولویت‌ها از 0 (کم) تا 127 (بالا) باشد، می‌توانیم به صورت دوره‌ای (مثلاً هر ثانیه) اولویت یک فرآیند مورد انتظار را 1 واحد افزایش دهیم. این کار تا زمانی که پردازنده به پردازنده اختصاص یابد، ادامه می‌یابد. و به این صورت کم اولویت‌ترین فرآیندها پس از مدتی اولویتشان افزایش می‌یابد و به بالاترین سطح اولویت می‌رسند و اجرا می‌شوند. همچنین برای جلوگیری از اختصاص بیش از حد پردازنده به وقفه‌ها می‌توان از مدیریت وقفه سطح بالایی و سطح پایینی که در قسمت قبل توضیح داده شد، استفاده کرد و یا بدترین حالت نرخ به وجود آمدن وقفه‌ها را کم کرد یا به جای وقفه از polling برای چک کردن اتفاقات استفاده کرد و یا نرخ ایجاد وقفه‌ها را محدود کرد.

زمان‌بندی بازخوردی چندسطحی:

برای پیاده‌سازی صف‌های گفته شده در این آزمایش به struct proc، struct schedinfo جدیدی به نام schedinfo اضافه کردیم که در ادامه به توضیح آن‌ها خواهیم پرداخت.

```
enum schedqueue {UNSET, ROUND_ROBIN, LCFS, BJF};
struct schedinfo {
    enum schedqueue queue;
    int last_run;
    struct bjfinfo bjf;
    int arrival_queue_time;
};
```

```

struct bjfinfo {
    int priority;
    float priority_ratio;
    int arrival_time;
    float arrival_time_ratio;
    float executed_cycle;
    float executed_cycle_ratio;
    int process_size;
    float process_size_ratio;
};

```

همچنین ساختار درون حلقه تابع scheduler را تغییر دادیم:

```

p = roundrobin(last_scheduled_RR);

if (p)
{
    last_scheduled_RR = p;
}
else
{
    p = lcfs();
    if (!p)
    {
        p = bestjobfirst();
        if (!p)
        {
            release(&ptable.lock);
            continue;
        }
    }
}
}

```

ابتدا در تابع به دنبال پردازش در صف roundrobin میگردیم و اگر پردازش جدید موجود نبود به ترتیب سراغ صف های بعدی میرویم.

ساز و کار افزایش سن

برای ساز و کار افزایش سن تابع زیر نوشته شده است. در استراکت اضافه شده یک فیلد last_run تعریف شده است که آخرین تیک سیستم که این پردازش اجرا شده است را ذخیره می کند. با محاسبه تفاوت last_run و تیک

سیستم در هر واحد زمان تصمیم می گیریم که پردازش به صف اول انتقال پیدا کند یا نه. این تابع پس از هر بار افزایش مقدار ticks در فایل trap.c صدا زده می شود.

```
void ageprocs(int os_ticks)
{
    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 (os_ticks - p->sched_info.last_run > AGING_THRESHOLD)
            {
                release(&ptable.lock);
                change_queue(p->pid, ROUND_ROBIN);
                acquire(&ptable.lock);
            }
        }
    }
    release(&ptable.lock);
}
```

سطح اول: زمان بند نوبت گردشی (RoundRobin)

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

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

```

if (p == last_scheduled)
    return 0;
}

```

1. این تابع یک اشاره‌گر به ساختار `last_scheduled`، را به عنوان یک آرگومان دریافت می‌کند که آخرین پردازش‌دهنده ای است با نوبت گردش زمانی شده است.
2. سپس وارد یک حلقه بی‌نهایت می‌شود، در هر بار اشاره‌گر `p` (که در ابتدا به `last_scheduled` اشاره می‌کند) را افزایش می‌دهد. (اگر به انتهای صف پردازش‌دهنده ها برسد بر می‌گردد از اول بررسی می‌کند).
3. بررسی می‌کند که آیا فرآیند فعلی که `p` به آن اشاره می‌کند در حالت `RUNNABLE` قرار دارد و در صف `ROUND_ROBIN` است یا خیر. اگر بله، یک اشاره‌گر به این فرآیند را برمی‌گرداند، که نشان‌دهنده این است که این فرآیند بعدی است که باید برنامه‌ریزی شود.
4. اگر تمام فرآیندها در جدول را بررسی کرده و به `last_scheduled` برگشته باشد بدون اینکه فرآیندی قابل اجرا در صف گردش پیدا کند، 0 را برمی‌گرداند.
5. این تابع در تابع `scheduler` برای هندل کردن پردازش‌دهنده ها فراخوانی می‌شود.

سطح دوم: زمان‌بند آخرین ورود-اولین رسیدگی (LCFS)

متغیر تاثیرگذار که برای مدیریت این صف به استراکت `schedinfo` اضافه کردیم، `arrival_queue_time` است. این متغیر را هربار که یک پردازش‌دهنده وارد یک صف می‌شود آپدیت می‌کنیم و به این نحو برای هر پردازش‌دهنده در هر صف زمانی که پردازش‌دهنده وارد آن صف شده را خواهیم داشت. آپدیت کردن مقدار این متغیر در تابع `change_queue` انجام می‌شود. از آنجا که این تابع هنگام ساخت هر پردازش‌دهنده حتما فراخوانی می‌شود مطمئن هستیم که این متغیر مقداردهی خواهد شد. حال به پیاده‌سازی زمان‌بند آخرین ورود-اولین رسیدگی می‌پردازیم:

```

struct proc *
lcfs(void)
{
    struct proc *result = 0;

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

```

```

        if (result->sched_info.arrival_queue_time <
p->sched_info.arrival_queue_time)
            result = p;
    }
    else
        result = p;
    }
    return result;
}

```

در این تابع میان تمام پردازش‌های موجود در صف که RUNNABLE هستند، بر اساس متغیر arrival_queue_time پردازش‌ای که آخرین بار وارد صف شده را پیدا می‌کنیم و آن را برای اجرا return می‌کنیم تا در scheduler مراحل اجرای آن طی شود.

سطح سوم: زمان‌بند اول بهترین کار (BJF)

تابع bestjobfirst مربوط به این زمان‌بندی است در این زمان‌بند به ازای هر پردازش‌ای که وضعیت آن RUNNABLE است و در صف BJF قرار دارد، رنک را مطابق با فرمول داده شده در تابع bjfrank، محاسبه می‌کنیم سپس تمام پردازش‌ها را بررسی کرده و پردازش‌ای که کمترین رنک را دارد، اجرا می‌کنیم.

```

struct proc *
bestjobfirst(void)
{
    struct proc *p;
    struct proc *min_p = 0;
    float min_rank = 2e6;

    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state != RUNNABLE || p->sched_info.queue != BJF)
            continue;
        float p_rank = bjfrank(p);
        if (p_rank < min_rank)
        {
            min_p = p;
            min_rank = p_rank;
        }
    }

    return min_p;
}

```



```
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 +
        p->sched_info.bjf.process_size * p->sched_info.bjf.process_size_ratio;
}
```

فراخوانی‌های سیستمی مورد نیاز

1. تغییر صف پردازش:

سیستم کال change_scheduling_queue به sysproc.c اضافه شده است و تابع اصلی change_queue در فایل proc.c است.

```
int
sys_change_scheduling_queue(void)
{
    int queue_number, pid;
    if(argint(0, &pid) < 0 || argint(1, &queue_number) < 0)
        return -1;

    if(queue_number < ROUND_ROBIN || queue_number > BJF)
        return -1;

    return change_queue(pid, queue_number);
}

void sys_print_processes_info(void) {
    print_processes_info();
}
```

```

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 = LCFS;
        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;

            p->sched_info.arrival_queue_time = ticks;
        }
    }
    release(&ptable.lock);
    return old_queue;
}

```

2. مقداردهی پارامتر BJB در سطح پردازش

تابع این کار `set-proc-bjf-params` است که به عنوان ورودی PID پردازش مورد نظر و 4 ضریب BJB را میگیرد. و پیاده سازی آن به صورت زیر می باشد:

```

int set_proc_bjf_params(int pid, float priority_ratio, float arrival_time_ratio, float executed_cycle_ratio, float process_size_ratio)
{
    struct proc *p;

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->pid == pid)
        {
            p->sched_info.bjf.priority_ratio = priority_ratio;
            p->sched_info.bjf.arrival_time_ratio = arrival_time_ratio;
            p->sched_info.bjf.executed_cycle_ratio = executed_cycle_ratio;
            p->sched_info.bjf.process_size_ratio = process_size_ratio;
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}

```

```

int
sys_set_proc_bjf_params(void)
{
    int pid;
    float priority_ratio, arrival_time_ratio, executed_cycle_ratio, process_size_ratio;
    if(argint(0, &pid) < 0 ||
        argfloat(1, &priority_ratio) < 0 ||
        argfloat(2, &arrival_time_ratio) < 0 ||
        argfloat(3, &executed_cycle_ratio) < 0 ||
        argfloat(4, &process_size_ratio) < 0){
        return -1;
    }

    return set_proc_bjf_params(pid, priority_ratio, arrival_time_ratio, executed_cycle_ratio, process_size_ratio);
}

```

3. مقداردهی پارامتر BJB در سطح سیستم

تابع این کار set-system-bjf-params است که به عنوان ورودی 4 ضریب BJB را میگیرد و آن را برای همه پردازش ها تنظیم می کند و پیاده سازی آن به صورت زیر می باشد:

```

int set_system_bjf_params(float priority_ratio, float arrival_time_ratio, float executed_cycle_ratio, float process_size_ratio)
{
    struct proc *p;

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        p->sched_info.bjf.priority_ratio = priority_ratio;
        p->sched_info.bjf.arrival_time_ratio = arrival_time_ratio;
        p->sched_info.bjf.executed_cycle_ratio = executed_cycle_ratio;
        p->sched_info.bjf.process_size_ratio = process_size_ratio;
    }
    release(&ptable.lock);
    return 0;
}

```

```

int
sys_set_system_bjf_params(void)
{
    int pid;
    float priority_ratio, arrival_time_ratio, executed_cycle_ratio, process_size_ratio;
    if(argint(0, &pid) < 0 ||
        argfloat(1, &priority_ratio) < 0 ||
        argfloat(2, &arrival_time_ratio) < 0 ||
        argfloat(3, &executed_cycle_ratio) < 0 ||
        argfloat(4, &process_size_ratio) < 0){
        return -1;
    }

    set_system_bjf_params(priority_ratio, arrival_time_ratio, executed_cycle_ratio, process_size_ratio);
    return 0;
}

```

4. چاپ اطلاعات

```

void sys_print_processes_info(void) {
    print_processes_info();
}

```

```
}
```

```
void print_processes_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, 9, 8, 8, 8, 8};
    cprintf("Process_Name   PID      State   Queue   Cycle   Arrival Priority
R_PrtY  R_Arvl  R_Exec  R_Size  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));

        cprintf("%d", p->pid);
        printspaces(columns[1] - count_digits(p->pid));

        . . .
    }
}
```

برنامه سطح کاربر برای تست

برای تست و فراخوانی فراخوان های سیستمی یک برنامه سطح کاربر به نام `schedule` نوشته و از آن استفاده کردیم.

هنگام شروع سیستم:

```
30: size 1000 mblocks 341 inodes 200 inlog 30 logstart 2 inodestart 32 bmap_start 30
init: starting sh
Group #29:
Elaheh Khodaverdi
Fereshteh Bagheri
Atefeh Mirzakhani
$ schedule info
```

Process_Name	PID	State	Queue	Cycle	Arrival	Priority	R_Prt	R_Arvl	R_Exec	R_Size	Rank
init	1	sleeping	1	2	0	3	1	1	1	1	5
sh	2	sleeping	1	1	6	3	1	1	1	1	10
schedule	3	running	2	1	805	3	1	1	1	1	809

سپس با اجرا کردن برنامه سطح کاربر `foo`:

```
$ foo&
schedule info
```

Process_Name	PID	State	Queue	Cycle	Arrival	Priority	R_Prt	R_Arvl	R_Exec	R_Size	Rank
init	1	sleeping	1	2	0	3	1	1	1	1	5
sh	2	sleeping	1	2	6	3	1	1	1	1	11
foo	6	runnable	2	80	22898	3	1	1	1	1	72133
foo	5	sleeping	2	0	22896	3	1	1	1	1	22899
foo	7	runnable	2	80	22898	3	1	1	1	1	22981
foo	8	runnable	2	80	22898	3	1	1	1	1	22981
foo	9	runnable	2	80	22898	3	1	1	1	1	22981
foo	10	runnable	2	80	22898	3	1	1	1	1	22981
schedule	11	running	2	0	23703	3	1	1	1	1	23706

سپس با استفاده از فراخوانی های سیستمی پارامتر های مربوط به زمان بندی BJB را ابتدا برای یک پردازش مشخص و سپس برای همه پردازش ها تغییر دادیم.

```
schedule set_process_bjf 6 8 8 8 8
BJF params set successfully
$ schedule info
```

Process_Name	PID	State	Queue	Cycle	Arrival	Priority	R_Prt	R_Arvl	R_Exec	R_Size	Rank
init	1	sleeping	1	2	0	3	1	1	1	1	5
sh	2	sleeping	1	2	6	3	1	1	1	1	11
foo	6	runnable	2	443	22898	3	8	8	8	8	579972
foo	5	sleeping	2	0	22896	3	1	1	1	1	22899
foo	7	runnable	2	443	22898	3	1	1	1	1	23344
foo	8	runnable	2	443	22898	3	1	1	1	1	23344
foo	9	runnable	2	443	22898	3	1	1	1	1	23344
foo	10	runnable	2	443	22898	3	1	1	1	1	23344
schedule	13	running	2	0	27335	3	1	1	1	1	43722

همچنین برای پردازش با pid 24 نیز صف را تغییر دادیم و تست کردیم.

```
$ schedule s_system_bjf 9 9 9 9
BJF params set successfully
$ schedule info
```

Process_Name	PID	State	Queue	Cycle	Arrival	Priority	R_Prt	R_Arvl	R_Exec	R_Size	Rank
init	1	sleeping	1	2	0	3	9	9	9	0	53
sh	2	sleeping	1	3	6	3	9	9	9	0	113
foo	16	runnable	2	291	33429	3	9	9	9	0	303514
foo	15	sleeping	2	0	33429	3	9	9	9	0	300889
foo	17	runnable	2	291	33429	3	9	9	9	0	303513
foo	18	runnable	2	291	33429	3	9	9	9	0	303513
foo	19	runnable	2	291	33430	3	9	9	9	0	303524
foo	20	runnable	2	291	33430	3	9	9	9	0	303522
schedule	22	running	2	0	36348	3	1	1	1	1	52735

```
$ foo&
$ schedule info
```

Process_Name	PID	State	Queue	Cycle	Arrival	Priority	R_Prt	R_Arvl	R_Exec	R_Size	Rank
init	1	sleeping	1	2	0	3	9	9	9	0	53
sh	2	sleeping	1	3	6	3	9	9	9	0	116
foo	25	runnable	2	66	76180	3	1	1	1	1	125401
foo	24	sleeping	2	0	76179	3	1	1	1	1	88470
foo	26	runnable	2	66	76180	3	1	1	1	1	88537
foo	27	runnable	2	66	76180	3	1	1	1	1	88537
foo	28	runnable	2	66	76181	3	1	1	1	1	88538
foo	29	runnable	2	66	76181	3	1	1	1	1	88538
schedule	30	running	2	0	76842	3	1	1	1	1	93229

```
$ schedule set_queue 24 3
Queue changed successfully
$ schedule info
```

Process_Name	PID	State	Queue	Cycle	Arrival	Priority	R_Prt	R_Arvl	R_Exec	R_Size	Rank
init	1	sleeping	1	2	0	3	9	9	9	0	53
sh	2	sleeping	1	4	6	3	9	9	9	0	120
foo	25	runnable	2	249	76180	3	1	1	1	1	125584
foo	24	sleeping	3	0	76179	3	1	1	1	1	88470
foo	26	runnable	2	249	76180	3	1	1	1	1	88720
foo	27	runnable	2	249	76180	3	1	1	1	1	88720
foo	28	runnable	2	249	76181	3	1	1	1	1	88721
foo	29	runnable	2	249	76181	3	1	1	1	1	88721
schedule	32	running	2	0	78681	3	1	1	1	1	95068