

## آزمایشگاه سیستم عامل

### تمرین کامپیوتری 3

#### اعضای گروه :

محمد سوری – 810100249

صفورا علوی پناه – 810100254

ریحانه حاجبی – 810100116

۱) چرا فراخوانی تابع sched() منجر به فراخوانی تابع scheduler() می‌شود؟ (منظور توضیح

شیوه اجرای فرایند است.)

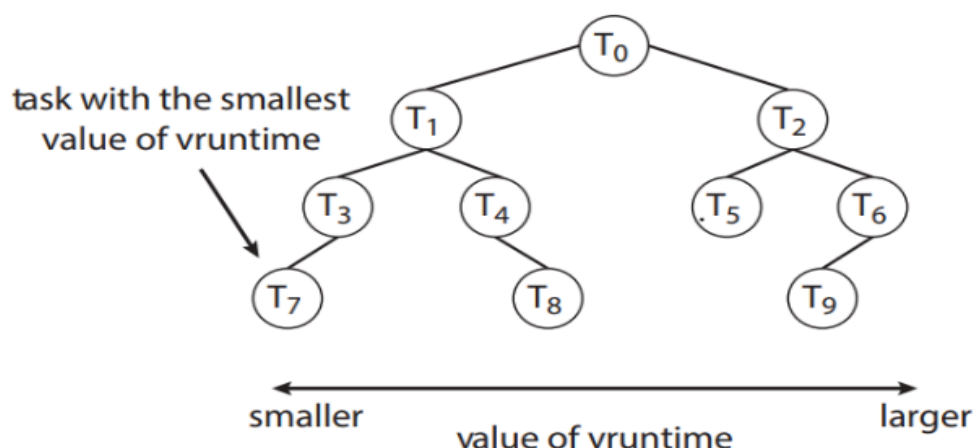
هر هسته تابع scheduler را صدا می‌زند و در این تابع پردازشی برای اجرا پیدا می‌شود و context-switch رخ می‌دهد تا آن هسته مشغول به پردازش آن پردازش شود. سپس هر پردازش زمانی که قرار است از حالت RUNNING خارج شود (که به 3 دلیل ممکن است رخ دهد: تمام شدن پردازش، تمام شدن تایمر یا یک اینترپت دیگر)، تابع sched را صدا می‌زند. تابع sched دوباره context-switch را انجام می‌دهد تا به همان پردازشی که عملیات scheduling را انجام می‌داد برگردیم. برای همین بعد از context-switch در تابع sched، ادامه‌ی تابع scheduler، از آنجایی که به یک پردازش سوئیچ کرده بودیم، اجرا می‌شود.

۲) صف پردازش‌هایی که تنها منبعی که برای اجرا کم دارند پردازنده است، صف آماده یا صف اجرا

نام دارد. در xv6 صف آماده مجزا وجود نداشته و از صف پردازش‌ها بدین منظور استفاده می‌گردد.

در زمان‌بند کاملاً منصف در لینوکس، صف اجرا چه ساختاری دارد؟

زمان‌بند کاملاً منصف لینوکس، به جای استفاده از یک ساختار داده صف استاندارد، وظیفه‌های (task) قابل اجرا در یک درخت قرمز-سیاه قرار می‌گیرند؛ درخت قرمز-سیاه یک درخت جستجوی دودویی متعادل است که در لینوکس کلید آن بر اساس مقدار vruntime است. این درخت در زیر نشان داده شده است:



هنگامی که یک وظیفه به وضعیت قابل اجرا درمی‌آید، به درخت افزوده می‌شود. اگر یک وظیفه در درخت قابل اجرا نباشد (به عنوان مثال، اگر در حالت بلاک شدن به منظور انتظار I/O باشد)، از درخت حذف می‌شود. به طور کلی، وظایفی که زمان پردازش کمتری داشته‌اند (مقادیر کوچکتر runtime) در سمت چپ درخت قرار دارند، و وظایفی که زمان پردازش بیشتری داشته‌اند، در سمت راست قرار دارند. با توجه به خصوصیات یک درخت جستجوی دودویی، چپ‌ترین گرهی درخت کوچک‌ترین مقدار کلید را دارد که به این معنا است که این وظیفه دارای بالاترین اولویت برای اجرا است.

**(۳) همان‌طور که در پروژه اول مشاهده شد، هر هسته پردازنده در xv6 یک زمان‌بند دارد. در لینوکس**

**نیز به همین‌گونه است. این دو سیستم‌عامل را از منظر مشترک یا مجزا بودن صف‌های زمان‌بندی**

**بررسی‌نمایید. و یک مزیت و یک نقص صف مشترک نسبت به صف مجزا را بیان کنید.**

در سیستم عامل xv6 از یک صف زمان‌بند برای همه پردازنده‌ها به طور مشترک استفاده می‌شود که ساختار آن به صورت زیر است:

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

در این ساختمان داده از یک صف از پردازنده‌ها که می‌تواند حداکثر 64 پردازنده را در خود نگه دارد و یک قفل برای مدیریت کردن دسترسی‌های همزمان استفاده شده است. اما در سیستم عامل لینوکس هر پردازنده یک صف زمان‌بند مجزا دارد.

مزیت صف مشترک نسبت به صف مجزا این است که نیازی به مدیریت توازن (load balancing) بین صف‌های پردازنده‌ها وجود ندارد، زیرا همه پردازنده‌ها در یک صف قرار می‌گیرند؛ این موضوع باعث افزایش سهولت در پیاده‌سازی و مدیریت زمان‌بندی می‌شود. اما نقص آن مشکل در دسترسی همزمان به صف است که به برای حل این مشکل از قفل استفاده می‌کنیم با این حال قفل می‌تواند کمی بر روی کارایی (performance) تاثیر بگذارد. علاوه بر آن، از آنجا که پردازنده‌ای که در این صف است، هربار در یک پردازنده اجرا می‌شود و بین آنها جهش می‌کند، با توجه به اینکه هر پردازنده، حافظه سریع (cache) سطح بالای خودش را دارد، کارایی حافظه سریع بسیار کمتر می‌شود.

#### ۴) در هر اجرای حلقه، ابتدا برای مدتی وقفه فعال می‌گردد. علت چیست؟ آیا در سیستم تک‌هسته‌ای به آن نیاز است؟

وقتی پردازنده در حالت idle یا در حالتی که تعدادی پردازش در حالت I/O قرار گرفته است باشد و پردازش دیگری نداشته باشیم که در حالت runnable باشد، در این حالت پردازش ای اجرا نمی‌شود. در نتیجه اگر وقفه‌ها نیز فعال نگردد پس از اتمام عمل I/O مربوط به پردازش‌ها، نمی‌توانیم آن‌ها را در حالت runnable قرار دهیم تا اجرا شوند و در این حالت سیستم به اصطلاح فریز می‌شود. در واقع اگر هیچ وقفه‌ای فعال نشده باشد، هیچ CPU دیگری که در حال اجرای یک پردازش باشد نمی‌تواند هیچ تعویض متنی و یا سیستم کال‌های وابسته به پردازش را اجرا کند. حال اگر قفل ptable فعال شود تمامی وقفه‌ها غیرفعال خواهند شد و لازم است جهت جلوگیری از فریز شدن سیستم وقفه‌ها فعال شوند تا اگر حالت پردازش‌ای به تغییر نیاز پیدا کرده است بتوان آن را تغییر داد. همچنین با توجه به اینکه در زمان اجرای پردازش، هیچ Preemptive‌ی نباید رخ بدهد، پس باید قبل از شروع کار پردازش باید اینکار انجام شود.

بله؛ در سیستم‌های تک‌هسته‌ای نیز به این کار نیاز است. زیرا اگر پردازنده در حالت idle قرار بگیرد، به دلیل غیرفعال شدن وقفه‌ها، ممکن است I/O‌ها هیچ وقت نرسند، به طور مثال منتظر یک عمل I/O باشند.

#### ۵) وقفه‌ها اولویت بالاتری نسبت به پردازش‌ها دارند. به طور کلی مدیریت وقفه‌ها در لینوکس در دو سطح صورت می‌گیرد. آن‌ها را نام برده و به اختصار توضیح دهید. اولویت این دو سطح مدیریت نسبت به هم و نسبت به پردازش‌ها چگونه است؟ مدیریت وقفه‌ها در صورتی که بیش از حد زمان‌بر شود، می‌تواند منجر به گرسنگی پردازش‌ها گردد. این می‌تواند به خصوص در سیستم‌های بی‌درنگ مشکل‌ساز باشد. چگونه این مشکل حل شده است؟

به طور معمول، مدیریت وقفه‌ها در سیستم‌ها به تقسیم وظایف بین یک وقفه سطح اول (FLIH) و یک وقفه سطح دوم (SLIH) می‌پردازد. ابتدا، فرآیند سطح اول اجرا می‌شود و سپس فرآیند سطح دوم. این تقسیم کار به خاطر این است که FLIH مسئول مدیریت وقفه‌های ضروری در کمترین زمان ممکن است، در حالی که SLIH بخش‌های مربوط به وقفه‌های زمان‌بر را مدیریت می‌کند. FLIH مسئول پردازش وقفه‌های ضروری به سرعت است. در سرویس‌دهی به وقفه‌ها دو حالت وجود دارد:

1. به وقفه سرویس کامل می‌دهد یعنی به عبارتی به طور کامل سرویس دهی می‌کند.
2. اطلاعات ضروری وقفه، که تنها در زمان وقوع وقفه در دسترس است، را ذخیره می‌کند و برای سرویس دهی کامل وقفه یک SLIH زمانبندی می‌کند.

همانطور که اشاره شد، SLIH مسئول مدیریت بخش‌های زمان‌بر وقفه است. SLIH‌ها یا یک ریسه مخصوص در سطح هسته برای هر handler دارد و یا توسط یک thread pool مدیریت می‌شوند. به دلیل امکان طولانی شدن زمان اجرا، SLIH‌ها معمولاً مانند پردازش‌ها زمانبندی می‌شود و SLIH‌ها در یک صف اجرا در انتظار پردازنده قرار می‌گیرند تا زمان لازم برای پردازش وقفه‌های زمان‌بر را اختصاص دهند. این تقسیم کار امکان مدیریت بهینه وقفه‌ها را فراهم می‌کند و از گرسنگی پردازش‌ها در صورت طولانی شدن زمان مدیریت وقفه‌ها جلوگیری می‌کند.

## پیاده سازی زمانبندی بازخوردی چند سطحی

ابتدا تعدادی ساختار در proc.h تعریف می‌کنیم تا بتوانیم اطلاعات مربوط به scheduling و صف‌ها را نگه داریم.

یک enum برای نوع صف‌ها تعریف می‌کنیم:

```
enum scheduling_queue {RR = 1, FCFS = 2, SJF = 3, CFS = 4};
```

سپس یک ساختار به نام scheduling\_info می‌سازیم که اطلاعات کلی مربوط به scheduling یک پردازش را نگه می‌دارد:

```
struct scheduling_info {
    enum scheduling_queue queue;
    int last_run;
    float executed_cycles;
    int arrival_time;
    float burst_time;
    int last_execution_time;
};
```

در نهایت در ساختار proc یک متغیر از نوع scheduling\_info اضافه می‌کنیم.

```
struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t* pgdir; // Page table
    .
```

```

.
.
.
struct scheduling_info sched_info;
};

```

مقادیر این متغیرها را برای هر پردازش در تابع allocproc مقداردهی اولیه می‌کنیم.

تابع scheduler را تغییر می‌دهیم به طوری که اول سعی کند از صف اول پردازش پیدا کند، سپس از صف دوم و سوم و در نهایت از صف چهارم پردازش پیدا کند.

```

void
scheduler(void)
{
    struct proc *p;
    struct proc *last_rr_scheduled = &ptable.proc[NPROC - 1];
    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);

        p = find_next_round_robin(last_rr_scheduled);
        if (p)
            last_rr_scheduled = p;
        if (!p)
        {
            p = find_next_fcfs();
            if (!p)
            {
                p = find_next_sjf();
                if (!p)
                {
                    if (now - last_cfs_scheduled->sched_info.last_execution_time >=

```

```

fair_period)
{
    if (!find_fair_period())
    {
        release(&ptable.lock);
        continue;
    }
    p = find_next_cfs(last_cfs_scheduled);
}
else
    p = myproc(); // Not Sure
    if (p)
        last_cfs_scheduled = p;
    if (!p)
    {
        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 = p;
switchvm(p);
p->state = RUNNING;

swtch(&(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);

}
}

```

توابع انتخاب کردن پردازش از هر صف را در ادامه توضیح می‌دهیم:

- **سطح اول: زمانبند نوبت گردشی**

در تابع scheduler آخرین پردازش‌ای که از صف نوبت گردشی انتخاب شده را نگه می‌داریم و هر بار از آن پردازش جست‌وجو را شروع می‌کنیم:

```
struct proc*  
find_next_round_robin(struct proc* last_scheduled){  
    struct proc *p = last_scheduled;  
    do  
    {  
        p++;  
        if (p == &ptable.proc[NPROC])  
        {  
            p = ptable.proc;  
        }  
  
        if (p->state == RUNNABLE && p->sched_info.queue == RR)  
        {  
            return p;  
        }  
    } while (p != last_scheduled);  
  
    return 0;  
}
```

- **سطح دوم: زمانبند آخرین ورود-اولین رسیدگی (FCFS)**



برای این صف بین همه‌ی پردازها سرچ می‌کنیم و پردازهی که در صف FCFS و بیشترین زمان ورود را دارد را انتخاب می‌کنیم.

```
struct proc *
find_next_fcfs()
{
    struct proc *p;
    struct proc *first_proc = 0;

    int mn = 2e9;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state != RUNNABLE || p->sched_info.queue != FCFS)
            continue;

        if (p->sched_info.arrival_time < mn)
        {
            mn = p->sched_info.arrival_time;
            first_proc = p;
        }
    }
    return first_proc;
}
```

- سطح سوم: زمانبند اول بهترین کار (SJF)

برای انتخاب پرداز در این صف در کل پردازها سرچ می‌کنیم و پردازهی که متعلق به این صف است و کمترین مقدار burst time داده شده را انتخاب می‌کنیم.

```
struct proc *
find_next_sjf()
{
    struct proc *p;
```

```

struct proc *shortest_job_process = 0;
float shortest_job_rank;
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
    if (p->state != RUNNABLE || p->sched_info.queue != SJF)
        continue;
    float current_job_rank = p->sched_info.burst_time;
    if (shortest_job_process == 0 || current_job_rank < shortest_job_rank){
        shortest_job_rank = current_job_rank;
        shortest_job_process = p;
    }
}
return shortest_job_process;
}

```

- سطح سوم: زمانبند کاملاً منصفانه (CFS)

برای انتخاب پروسه از این صف اول مقدار بازه زمانی از قبل تعیین شده PERIOD را باید به تعداد تقسیم کنیم تا بازه زمانی منصفانه خود را به دست بیاوریم و سپس به انتخاب پردازش میپردازیم. (چک کردن تمام شدن پریود در تابع scheduler چک میشود).

```

int find_fair_period()
{
    int now = ticks;

    struct proc *prc;

    int size_of_queue = 0;

    for (prc = ptable.proc; prc < &ptable.proc[NPROC]; prc++){

        if (prc->state == RUNNABLE && prc->sched_info.queue == CFS)

            size_of_queue++;

    }

    if (size_of_queue == 0)

```

```

    return 0;

    fair_period = PERIOD / size_of_queue;
}

struct proc *
find_next_cfs(struct proc *last_scheduled){

    struct proc *fair_process = 0;

    struct proc *p = last_scheduled;

    while (p != last_scheduled)
    {

        p++;

        if (p >= &ptable.proc[NPROC])

            p = ptable.proc;

        if (p->state == RUNNABLE && p->sched_info.queue == CFS)

            fair_process = p;

    }

    return fair_process;
}

```

برای پیاده‌سازی aging تابع age\_process را تعریف می‌کنیم که هر بار که مقدار ticks در trap.c زیاد می‌شود این تابع را هم صدا می‌زنیم که بررسی کند اگر نیاز بود صف پردازش مد نظر را عوض کند.

```

enum scheduling_queue
change_queue(struct proc *p, enum scheduling_queue new_queue)

```

```

{
    enum scheduling_queue old_queue = p->sched_info.queue;
    p->sched_info.queue = new_queue;
    p->sched_info.last_run = ticks;
    return old_queue;
}

void age_processes()
{
    struct proc *p;
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state != RUNNABLE || p->sched_info.queue == RR)
        {
            continue;
        }
        if (ticks - p->sched_info.last_run > AGING_THRESHOLD)
        {
            change_queue(p, RR);
        }
    }
    release(&ptable.lock);
}

```

## فراخوانی های سیستمی

### 1. تغییر صف پردازش

ابتدا فراخوانی سیستمی مورد نظر را با نام `change_process_queue` در سیستم عامل (همانطور که در گزارش آزمایشگاه 2 به صورت گام به گام توضیح داده شد)، تعریف می کنیم. در نهایت تابع این فراخوانی را در فایل `proc.c` به صورت زیر تعریف می کنیم:

```

int
change_process_queue(int pid, int queue_num) {
    struct proc* p;
    acquire(&ptable.lock);

```

```

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
        break;
    }
}
release(&ptable.lock);
int old_queue_num= change_queue(p, queue_num);
return old_queue_num;
}

```

برنامه سطح کاربر زیر با نام change\_process\_queue.c را برای تست این فراخوانی سیستمی ایجاد کردیم:

```

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

void set_queue(int pid,int new_queue) {
    if(pid < 1) {
        printf(1,"invalid pid\n");
        return;
    }
    if(new_queue < 1 || new_queue > 4) {
        printf(1,"invalid queue\n");
        return;
    }
    int res = change_process_queue(pid, new_queue);
    if(res < 0) {
        printf(1,"Error changing queue\n");
    }
    else {
        printf(1,"process with pid = %d changed queue from %d to %d\n",pid , res , new_queue);
    }
}

```

```
int main(int argc, char* argv[]) {
    if (argc < 3) {
        printf(1, "not enogh params\n");
        exit();
    }

    set_queue(atoi(argv[1]), atoi(argv[2]));
    exit();
}
```

خروجی این فراخوانی سیستمی به صورت زیر خواهد بود:

```
QEMU
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Mohammad Sourì
Safoora Alavi Panah
Reyhaneh Hajebi
$ foo&
$ change_process_queue 4 3
process with pid = 4 changed queue from 2 to 3
$ _
```

2. مقدار دهی پارامتر SJF :

ابتدا فراخوانی سیستمی مورد نظر را با نام `set_sjf_process` در سیستم عامل تعریف می‌کنیم. در نهایت تابع این فراخوانی را در فایل `proc.c` به‌صورت زیر تعریف می‌کنیم:

```
int
set_sjf_process(int pid, int burst_time)
{
    acquire(&ptable.lock);
    struct proc* p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            p->sched_info.burst_time = burst_time;
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
```

برنامه سطح کاربر زیر با نام `set_sjf_process.c` را برای تست این فراخوانی سیستمی ایجاد کردیم:

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

void set_process_sjf(int pid, int burst)
{
    if (pid < 1){
        printf(1, "Invalid pid\n");
        return;
    }
    if (burst < 0){
        printf(1, "Invalid burst time\n");
        return;
    }
}
```

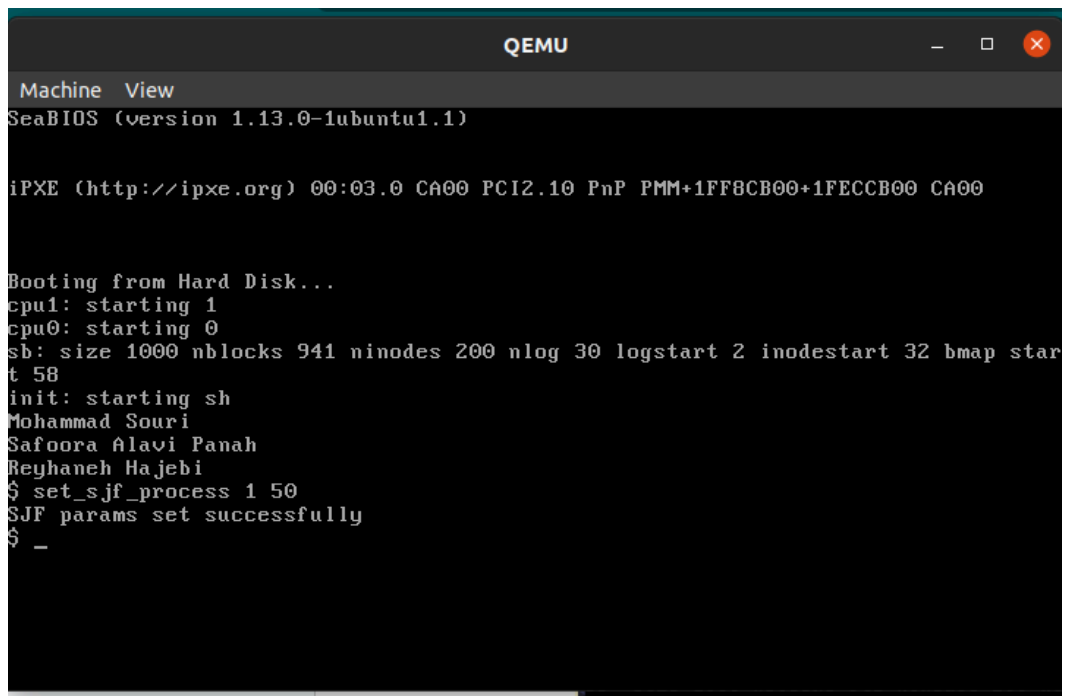
```

int res = set_sjf_process(pid, burst);
if (res < 0)
    printf(1, "Error setting SJF params\n");
else
    printf(1, "SJF params set successfully\n");
}

int main(int argc, char* argv[]){
    if (argc < 3){
        printf(1, "not enough params");
        exit();
    }
    set_process_sjf(atoi(argv[1]), atoi(argv[2]));
    exit();
}

```

خروجی این فراخوانی سیستمی به صورت زیر خواهد بود:



```

QEMU
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECB00 CA00

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Mohammad Souri
Safoora Alavi Panah
Reyhaneh Hajebi
$ set_sjf_process 1 50
SJF params set successfully
$ -

```

3. چاپ اطلاعات:



ابتدا فراخوانی سیستمی مورد نظر را با نام `print_schedule_info` در سیستم عامل تعریف می‌کنیم.  
در نهایت تابع این فراخوانی را در فایل `proc.c` به‌صورت زیر تعریف می‌کنیم:

```
void
print_schedule_info(void) {
static char *states[] = {
    [UNUSED]    "unused",
    [EMBRYO]    "embryo",
    [SLEEPING]  "sleeping",
    [RUNNABLE]  "runnable",
    [RUNNING]   "running",
    [ZOMBIE]    "zombie"
};
static int columns[] = {16, 9, 9, 9, 9, 9, 9, 9};
printf("Process_Name    PID    State    Queue    Cycle    Arrival
Burst\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 = "???";

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

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

    printf("%s", state);
```

```

    printspaces(columns[2] - strlen(state));

    cprintf("%d", p->sched_info.queue);
    printspaces(columns[3] - digitcount(p->sched_info.queue));

    cprintf("%d", (int)p->sched_info.executed_cycles);
    printspaces(columns[4] -
digitcount((int)p->sched_info.executed_cycles));

    cprintf("%d", p->sched_info.arrival_time);
    printspaces(columns[5] - digitcount(p->sched_info.arrival_time));

    cprintf("%d", p->sched_info.burst_time);
    printspaces(columns[6] - digitcount(p->sched_info.burst_time));
    cprintf("\n");
}
}

```

برنامه سطح کاربر زیر با نام `schedule_info.c` را برای تست این فراخوانی سیستمی ایجاد کردیم:

```

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

int main(){
    print_schedule_info();
    exit();
}

```

## برنامه سطح کاربر

در برنامه‌ی کاربر چند بار `fork` را صدا می‌زنیم تا پردازش جدید ساخته شود و داخل هر پردازش عملیات محاسباتی طولانی و `sleep` طولانی قرار می‌دهیم:

```

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

#define NUM_FORKS 10
int main(int argc, char* argv[]){
    for (int i = 0; i < NUM_FORKS; i++){
        int pid = fork();
        if (pid == 0){
            sleep(1000000);
            int x = 0;
            for (long long j = 0; j < 1000000000000; j++){
                for (long long k = 0; k < 1000000000000; k++){
                    x += k * 12 - j;
                }
            }
            break;
        }
        //while(1);
    }
    while (wait() != -1);
    exit();
}

```

خروجی چاپ اطلاعات هم بعد از ساخت برنامه‌ی کاربر در پس‌زمینه به شکل زیر می‌شود:

```

$ foo&
$ schedule_info
Process_Name  PID  State  Queue  Cycle  Arrival  Priority  R_Prt  R_Arvl  R_Exec  R_Size  Rank
-----
init          1    sleeping 1      1      0        3        1      1      1      1      12292
sh            2    sleeping 2      2      4        3        1      1      1      1      16393
foo           5    runnable 2     36     294      3        1      1      1      1      12621
foo           4    sleeping 2      0     294      3        1      1      1      1      12585
foo           6    runnable 2     36     294      3        1      1      1      1      12621
foo           7    runnable 2     36     294      3        1      1      1      1      12621
foo           8    runnable 2     36     294      3        1      1      1      1      12621
foo           9    runnable 2     36     294      3        1      1      1      1      12621
foo          10    runnable 2     36     294      3        1      1      1      1      12621
foo          11    runnable 2     36     294      3        1      1      1      1      12621
foo          12    runnable 2     36     294      3        1      1      1      1      12621
foo          13    runnable 2     36     294      3        1      1      1      1      12621
foo          14    runnable 2     36     294      3        1      1      1      1      12621
schedule_info 15    running  2      1     656      3        1      1      1      1      12948
$

```

بعد از حدود یک دقیقه تمام پردازش‌های foo تمام می‌شوند و جدول اطلاعات به شکل زیر می‌شود:

```
$ schedule_info
```

Process_Name	PID	State	Queue	Cycle	Arrival	Priority	R_Prt	R_Arvl	R_Exec	R_Size	Rank
init	1	sleeping	1	1	0	3	1	1	1	1	12292
sh	2	sleeping	2	3	2	3	1	1	1	1	16392
schedule_info	18	running	2	0	16899	3	1	1	1	1	29190

```
$
```

Github : [link](#)

Last commit message : [link](#)