



به نام خدا



دانشگاه تهران  
دانشکده مهندسی برق و کامپیوتر  
درس آزمایشگاه سیستم عامل

پروژه سوم

نام و نام خانوادگی	دانیال سعیدی (810198571) سروش صادقیان (810898048) محمد قره حسنلو (810198461)
تاریخ ارسال گزارش	۲۵ اردیبهشت ۱۴۰۱
رپیو گیتهاب	<a href="https://github.com/daniel-saeedi/OS-Lab">github.com/daniel-saeedi/OS-Lab</a>
آخرین Commit ID	8a51a04d4d2e919f2ed590ee28db863cee63a34f

## سوال ۱

چون در کد زیر از تابع swtch استفاده شده است و این تابع در واقع عمل context switch را انجام میدهد و چون در اینجا context فعلی (proc->context) ذخیره میشود و به scheduler context قبلی که در cpu->scheduler ذخیره شده است، سوییچ میکنیم، پس در این فرایند از scheduler استفاده کرده ایم.

```
sched(void)
{
    int intena;
    struct proc *p = myproc();
    if(!holding(&ptable.lock))
        panic("sched ptable.lock");
    if(mycpu()->ncli != 1)
        panic("sched locks");
    if(p->state == RUNNING)
        panic("sched running");
    if(readeflags() & FL_IF)
        panic("sched interruptible");
    intena = mycpu()->intena;
    swtch(&p->context, mycpu()->scheduler);
    mycpu()->intena = intena;
}
```

## سوال ۲

در لینوکس زمان بند CFS یک الگوریتم منصفانه برای انتخاب task بعدی استفاده میکند. لینوکس به جای صف از درخت black-red استفاده میکند که در آن CFS خودش به تنهایی اولویت ها را تخصیص نمیدهد بلکه از virtual run time برای اولویت بندی استفاده میکند. که شکل درخت black-red در صفحه 237 کتاب نمایش داده شده است.

## سوال ۳

در xv6 فقط یک صف مشترک (ptable) دارای یک lock برای مدیریت همزمانی دسترسی به صف داریم و همچنین صفی از process ها در استراکچر proc قرار گرفته اند که در ptable است، ولی در سیستم عامل لینوکس هر پردازنده صف مجزای خود را دارد. در صف مشترک نیازی به برقراری توازن بین تعدادی از صف ها نیست چون در کل یک صف داریم ولی این مزیت را در صف های مجزا نداریم. در صف های مجزا نیازی به lock نداریم چون

هر process برای خود صف مجزا دارد و همزمانی دسترسی به صف مانند صف مشترک اتفاق نمی افتد.

## سوال ۴

ممکن است در ابتدا هیچ پردازش ای که state آن RUNNABLE باشد نداشته باشیم و همه آنها در حال انجام کاری مانند عمل I/O هستند. در اینجا با وقفه میتوانیم این حالت را مدیریت کنیم و ورودی و خروجی پردازش ها به پابان برسند و پردازش هایی با حالت RUNNABLE داشته باشیم و این حالتی که توضیح دادیم، برای سیستم های تک هسته ای هم ممکن است اتفاق بیفتد.

## سوال ۵

در دسته top half، در این نوع وقفه ها با شماره یا خط یکسان disable اند ولی بقیه وقفه ها احتمالا ران شوند. (وقفه ها غیرفعال و فعالیت های لازم انجام میشود.) در دسته bottom half، تمام وقفه های فعال شده توسط زمان بند تضمین میکند که این نوع وقفه هرگز آنها را interrupt نمیکند. این قسمت استناد میکند که interrupt service خروج کند. (وقفه های دیگر فعال و فعالیت مورد نیاز در اینجا رخ میدهد.) با توجه به عکس صفحه 796 کتاب، top-half نسبت به bottom-half و bottom-half نسبت به پردازش ها اولویت بیشتری دارد.

```
// Per-process state
struct proc {
    uint sz;                // Size of process memory (bytes)
    pde_t* pgdir;           // Page table
    char *kstack;           // Bottom of kernel stack for this process
    enum procstate state;   // Process state
    int pid;                // Process ID
    struct proc *parent;    // Parent process
    struct trapframe *tf;   // Trap frame for current syscall
    struct context *context; // switch() here to run process
    void *chan;             // If non-zero, sleeping on chan
    int killed;             // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;      // Current directory
    char name[16];          // Process name (debugging)

    uint creation_time;     // Arrival time
    int queue;              // Queue number
    int priority;           // Priority in Queue
    int priority_ratio;
    int arrival_time_ratio;
    int executed_cycle_ratio;
    int exec_cycle;
    int last_cpu_time;
    uint shm;
    int wait_cycles;
};
```

```

void
scheduler(void)
{
    struct proc *p = 0;
    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);

        // Check if process is waiting more than 8000 cycles
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if(p->wait_cycles >= 800) {
                p->wait_cycles = 0;
                p->queue = 1;
            } else if (p->state == RUNNABLE) {
                p->wait_cycles++;
            }
        }

        p = round_robin_finder();

        if (p == 0)
            p = fcfs_finder();

        if(p == 0)
            p = bjf_finder();

        if (p == 0) {
            release(&ptable.lock);
            continue;
        }
        c->proc = p;
        switchvm(p);
        p->state = RUNNING;
        p->wait_cycles = 0;

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

        c->proc = 0;
        release(&ptable.lock);
    }
}

```

## تابع round\_robin\_finder

```
struct proc*
round_robin_finder(void)
{
    struct proc *p;
    struct proc *best = 0;

    int now = ticks;
    int max_proc = -100001;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE || p->queue != 1)
            continue;

        if(now - p->last_cpu_time > max_proc){
            max_proc = now - p->last_cpu_time;
            best = p;
        }
    }
    return best;
}
```

## تابع fcfs\_finder

```
struct proc*
fcfs_finder(void)
{
    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->queue != 4)
            continue;

        if (p->creation_time < mn)
        {
            mn = p->creation_time;
            first_proc = p;
        }
    }
    return first_proc;
}
```

```
struct proc*
bjf_finder(void)
{
    struct proc* p;
    struct proc* min_proc = 0;
    float min_rank = 1000001;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if (p->state != RUNNABLE || p->queue != 3)
            continue;
        if (get_rank(p) < min_rank){
            min_proc = p;
            min_rank = get_rank(p);
        }
    }

    return min_proc;
}
```

## تابع ست کردن مقادیر

```
void
set_queue(int pid, int queue)
{
    struct proc *p;

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

void
set_bjf_params(int pid, int priority_ratio, int arrival_time_ratio, int
executed_cycle_ratio)
{
    struct proc *p;

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

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

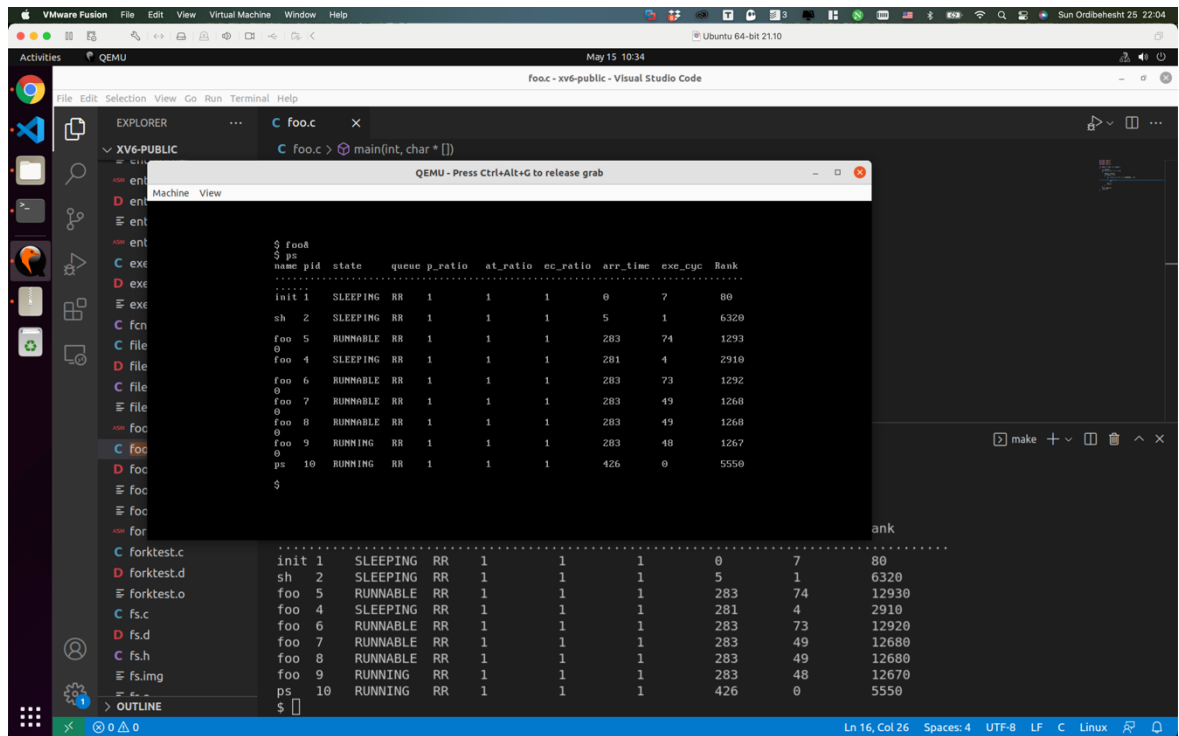


## برنامه سطح کاربر foo

```
int main(int argc, char *argv[])
{
    int pid[10];
    for (int i = 0 ; i < 5 ; i++)
    {
        pid[i] = fork();
        if (pid[i] == 0)
        {
            for (long int j = 0 ; j < 2000000000 ; j++)
            {
                int x = 1;
                x += 1;
            }
            exit();
        }
    }
    while (wait());
    return 0;
}
```

## خروجی ps

```
IPXE (http://ipxe.org) 00:03:0 CA00 FC12:10 PnP PMM-1F7B5B0+1FECB5B0 CA00
Booting from Hard Disk...
cpu0: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 50
init: starting sh
$ ps
name pid state queue p_ratio at_ratio ec_ratio arr_time exe_cyc Rank
.....
init 1 SLEEPING RR 1 1 1 0 3 40
sh 2 SLEEPING RR 1 1 1 6 2 2170
ps 3 RUNNING RR 1 1 1 150 0 5750
$
```



QEMU - Press Ctrl+Alt+G to release grab

```

$ ps
name pid state queue p_ratio at_ratio ec_ratio arr_time exe_cyc Rank
.....
init 1 SLEEPING RR 1 1 1 0 7 80
sh 2 SLEEPING RR 1 1 1 5 1 6320
foo 5 RUNNING RR 1 1 1 283 4936 6155
foo 4 SLEEPING RR 1 1 1 281 4 2910
foo 6 RUNNABLE BKF 1 1 1 283 4855 6074
foo 7 RUNNABLE RR 1 1 1 283 3310 4529
foo 8 RUNNABLE RR 1 1 1 283 3308 4527
foo 9 RUNNABLE RR 1 1 1 283 3306 4525
ps 12 RUNNING RR 1 1 1 10140 1 105260
$

```

با توجه به مکانیزم aging از BKF به RR رفت:

```

foo.c
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4 #include "fcntl.h"
5
6 int main(int argc, char *argv[])
7 {
8     int pid[10];
9     for (int i = 0 ; i < 5 ; i++)
10     {
11         pid[i] = fork();
12         if (pid[i] == 0)
13         {
14             for (long int j = 0 ; j < 2250000000 ; j++)
15             {

```

```

$ ps
name pid state queue p_ratio at_ratio ec_ratio arr_time exe_cyc Rank
.....
init 1 SLEEPING RR 1 1 1 0 7 80
sh 2 SLEEPING RR 1 1 1 5 1 6320
foo 5 RUNNABLE RR 1 1 1 283 6317 75360
foo 4 SLEEPING RR 1 1 1 281 4 2910
foo 6 RUNNABLE RR 1 1 1 283 6095 73140
foo 7 RUNNABLE RR 1 1 1 283 4275 54940
foo 8 RUNNABLE RR 1 1 1 283 4272 54910
foo 9 RUNNABLE RR 1 1 1 283 4271 54900
ps 13 RUNNING RR 1 1 1 12901 0 133670
$

```

```

C foo.c > main(int, char* [])
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4 #include "fcntl.h"
5
6 int main(int argc, char *argv[])
7 {
8     int pid[10];
9     for (int i = 0 ; i < 5 ; i++)
10     {
11         pid[i] = fork();
12         if (pid[i] == 0)
13         {
14             for (long int j = 0 ; j < 2250000000 ; j++)
15             {

```

ps 15 RUNNING RR 1 1 1 21671 1 225400  
\$ set\_all\_bjf 5 6 7  
\$ ps

name	pid	state	queue	p_ratio	at_ratio	ec_ratio	arr_time	exe_cyc	Rank
init	1	SLEEPING	RR	5	6	7	0	7	540
sh	2	SLEEPING	RR	5	6	7	5	1	31670
foo	5	RUNNABLE	RR	5	6	7	283	12546	942000
foo	4	SLEEPING	RR	5	6	7	281	4	17440
foo	6	RUNNABLE	RR	5	6	7	283	12315	925830
foo	7	RUNNING	RR	5	6	7	283	8428	653740
foo	8	RUNNABLE	RR	5	6	7	283	8424	653460
foo	9	RUNNABLE	RR	5	6	7	283	8423	653390
ps	17	RUNNING	RR	1	1	1	25355	1	256040

QEMU - Press Ctrl+Alt+C to release grab

```

$ set_hbf 6 1 2 3
$ ps
name pid state queue p_ratio at_ratio ec_ratio arr_time exe_cyc Rank
init 1 SLEEPING RR 1 1 1 0 7 80
sh 2 SLEEPING RR 1 1 1 5 1 6320
foo 5 RUNNABLE RR 1 1 1 283 10704 119230
foo 4 SLEEPING RR 1 1 1 281 4 2910
foo 6 RUNNABLE RR 1 2 3 283 10476 329300
foo 7 RUNNABLE RR 1 1 1 283 7201 84200
foo 8 RUNNABLE RR 1 1 1 283 7196 84150
foo 9 RUNNABLE RR 1 1 1 283 7195 84140
ps 15 RUNNING RR 1 1 1 21671 1 225400
$ _

```