آزمایشگاه سیستم عامل

تمرین کامپیوتری ۳

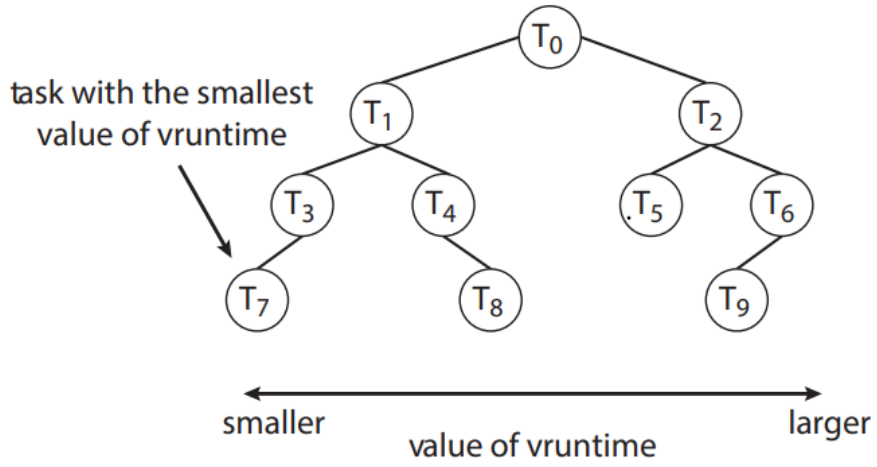
اعضای گروه :
علی حمزه پور- ۸۱۰۱۰۰۱۲۹
نرگس سادات سیدحائری – ۸۱۰۱۰۰۱۶۵
مینا شیرازی - ۸۱۰۱۰۰۲۵۰

سوال یک :چرا فراخوانی تابع sched، منجر به فراخوانی تابع scheduler می‌شود؟

هر هسته تابع scheduler را صدا می‌زند و در این تابع پردازهی برای اجرا پیدا می‌شود و context-switch رخ می‌دهد تا آن هسته مشغول به پردازش آن پردازه شود. سپس هر پردازه زمانی که قرار است از حالت RUNNING خارج شود(که به ۳ دلیل ممکن است رخ دهد: تمام شدن پردازه، تمام شدن تایمر یا یک اینتراپت دیگر)، تابع sched را صدا می‌زند. تابع sched دوباره context-switch را انجام می‌دهد تا به همان پردازهی که عملیات scheduling را انجام می‌داد برگردیم. برای همین بعد از context-switch در تابع sched، ادامه‌ی تابع scheduler، از آنجایی که به یک پردازه سوییچ کرده بودیم، اجرا می‌شود.

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

زمانبند کاملا منصف لینوکس، به جای استفاده از یک ساختار داده صف استاندارد، وظیفه های (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 vruntime) 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.

سوال سه: دو سیستم عامل لینوکس و xv٦ را از منظر مشترک یا مجزا بودن صف های زمانبندی بررسی نمایید. و یک مزیت و یک نقص صف مشترک نسبت به صف مجزا را بیان کنید.

در سیستم عامل xv٦ 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 پردازه ها که می تواند حداکثر ٦٤ پردازه را در خود نگه دارد و یک قفل برای مدیریت کردن دسترسی های 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 صف های پردازه‌ها وجود ندارد، زیرا همه پردازه‌ها 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 می کنیم با این حال قفل می تواند کمی بر روی کارایی (performance) تاثیر بگذارد. علاوه بر آن، از آنجا که پردازه‌ای 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.

سوال چهار: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 پردازه در حالت I/O قرار گرفته است باشد و پردازه ی دیگری نداشته باشیم که در حالت runnable باشد،در این حالت پردازه ای اجرا نمی شود. در نتیجه اگر وقفه ها نیز فعال نگردد پس از اتمام عمل I/O مربوط به پردازه ها، نمی توانیم آن ها را در حالت runnable قرار دهیم تا اجرا شوند و در این حالت سیستم به اصطلاح فریز میشود. در واقع اگر هیچ وقفه ای فعال نشده باشد ،هیچ CPU دیگری که در حال اجرای یک پردازه باشد نمی‌تواند هیچ تعویض متنی و یا سیستم کال های وابسته به پردازه را اجرا کند. حال اگر قفل ptable فعال شود تمامی وقفه ها غیرفعال خواهند شد و لازم است جهت جلوگیری از فریز شدن سیستم وقفه ها فعال شوند تا اگر حالت پردازه ای به تغییر نیاز پیدا کرده است بتوان

آن را تغییر داد. همچنین با توجه به اینکه در زمان اجرای پردازه، هیچ Preemptiveی نباید رخ بدهد، پس باید قبل از شروع کار پردازه باید اینکار انجام شود.

بله؛ در سیستم های تک هسته ای نیز به این کار نیاز است. زیرا اگر پردازنده در حالت idle قرار بگیرد، به دلیل غیرفعال شدن وقفه ها، ممکن است I/O ها هیچ وقت نرسند، به طور مثال منتظر یک عمل I/O باشند.

سوال پنج: وقفه‌ها اولویت بالاتری نسبت به پردازه‌ها دارند. به طور کلی مدیریت وقفه‌ها در لینوکس در دو سطح صورت میگیرد. آنها را نام برده و به اختصار توضیح دهید.

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 سرویس دهی به وقفه ها دو حالت وجود دارد:

۱. به وقفه سرویس کامل می دهد یعنی به عبارتی به طور کامل سرویس دهی می‌کند.
۲. 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، SLIH ها معمولا مانند پردازه‌ها زمانبندی می‌شود و SLIHها در یک صف اجرا در انتظار پردازنده قرار می‌گیرند.

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,LCFS = 2, BJF = 3};
```

سپس یک ساختار به نام scheduling_info می‌سازیم که اطلاعات کلی مربوط به scheduling یک پردازه را نگه می‌دارد:

```
struct scheduling_info {
```

```
  enum scheduling_queue queue;
  int last_run;
  int priority;
  float executed_cycles;
  int arrival_time;
  struct bjf_info bjf_coeffs;
};
```

ساختار bjf_info هم ضرایب مربوط به حساب کردن rank در صف bjf را در خود دارد:

```
struct bjf_info {
  float priority_ratio;
  float arrival_time_ratio;
  float executed_cycles_ratio;
  float process_size_ratio;
};
```

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 پردازه در تابع allocproc مقداردهی اولیه می‌کنیم.(برای پردازه init صف را به صف اول ست می‌کنیم و بقیه‌ی پردازه‌ها را به صف دوم.)

تابع scheduler را تغییر می‌دهیم به طوری که اول سعی کند از صف اول پردازه پیدا کند، سپس از صف دوم و درنهایت از صف سوم پردازه پیدا کند.

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

```
struct cpu *c = mycpu();
c->proc = 0;

 // از اینجا را ما تغییر دادیم
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;
  }
  else {
    p = find_next_lcfs();
    if (!p){
      p = find_next_bjf();
      if (!p){
        release(&ptable.lock);
        continue;
      }
    }
  }

  //از اینجا را دیگر ما تغییر ندادیم
  // 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;
  switchuvm(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 پردازه از هر صف را در ادامه توضیح می‌دهیم:

- **سطح اول: زمانبند نوبت گردشی**

در تابع scheduler آخرین پردازه‌ای که از صف نوبت گردشی انتخاب شده را نگه می‌داریم و هر بار از آن پردازه 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 (LCFS)**

برای این صف بین همه‌ی پردازه‌ها سرچ می‌کنیم و پردازهی که در صف LCFS و کمترین زمان ورود را دارد را انتخاب می‌کنیم.

```
struct proc*
find_next_lcfs(){
  struct proc *p;
  struct proc *last_process = 0;
  int max_start_time = -1;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->state != RUNNABLE || p->sched_info.queue != LCFS){
```

```
      continue;
    }
    if (p->sched_info.arrival_time > max_start_time){
      max_start_time = p->start_time;
      last_process = p;
    }
  }
  return last_process;
}
```

- **سطح سوم: زمانبند اول بهترین کار (BJF)**

برای انتخاب پردازه در این صف در کل پردازه‌ها سرچ می‌کنیم و پردازهی که متعلق به این صف است و کمترین مقدار را طبق فرمول داده‌شده دارد را انتخاب می‌کنیم.

```
float
calc_process_bjf_rank(struct proc* p){
  return p->sched_info.arrival_time * p-
>sched_info.bjf_coeffs.arrival_time_ratio +
         p->sched_info.executed_cycles * p-
>sched_info.bjf_coeffs.executed_cycles_ratio +
         p->sched_info.priority * p->sched_info.bjf_coeffs.priority_ratio +
         p->sz * p->sched_info.bjf_coeffs.process_size_ratio;
}

struct proc*
find_next_bjf(){
  struct proc *p;
  struct proc *best_job_process = 0;
  float best_job_rank;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->state != RUNNABLE || p->sched_info.queue != BJF){
      continue;
    }
    float current_job_rank = calc_process_bjf_rank(p);
    if (best_job_process == 0 || current_job_rank < best_job_rank){
      best_job_rank = current_job_rank;
      best_job_process = p;
    }
```

```
  }
  return best_job_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 پردازه مدنظر 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

۱. تغییر صف پردازه

ابتدا فراخوانی سیستمی مورد نظر را با نام change_process_queue در سیستم عامل (همانطور که در گزارش آزمایشگاه ۲ به‌صورت گام‌به‌گام توضیح داده شد)، تعریف می‌کنیم. در نهایت تابع این فراخوانی را در فایل proc.c به‌صورت زیر تعریف می‌کنیم:

```
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;
}
```

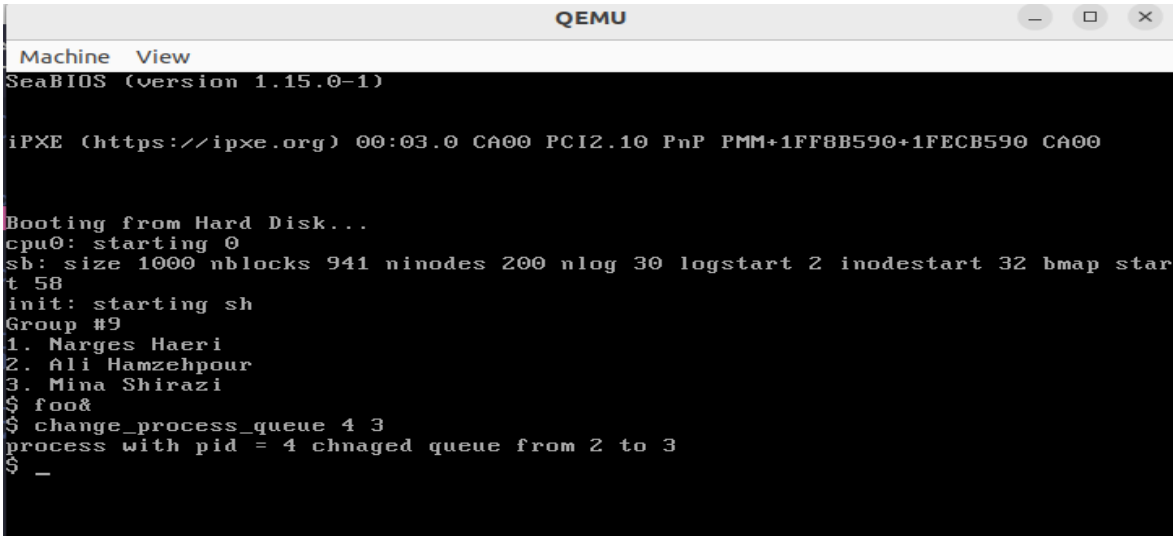
برنامه سطح کاربر زیر با نام change_process_queue.c را برای تست این فراخوانی سیستمی ایجاد کردیم:

```
#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 > 3)
    {
        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 chnaged queue from %d to
%d\n", pid, res, new_queue);
    }
}
int main(int argc,char* argv[]){
    if (argc<3){
        printf(1,"not enough params");
        exit();
    }
    set_queue(atoi(argv[1]),atoi(argv[2]));
    exit();

}
```

خروجی این فراخوانی سیستمی به صورت زیر خواهد بود:



۲. مقدار دهی پارامتر BJF در سطح پردازه

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

```
int
set_bjf_process(int pid, int priority_ratio, int arrival_time_ratio,
```

```
int 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.bjf_coeffs.priority_ratio = priority_ratio;
      p->sched_info.bjf_coeffs.arrival_time_ratio =
arrival_time_ratio;
      p->sched_info.bjf_coeffs.executed_cycles_ratio =
executed_cycles_ratio;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

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

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

void set_process_bjf(int pid, int priority_ratio, int
arrival_time_ratio, int executed_cycle_ratio)
{
    if (pid < 1)
    {
        printf(1, "Invalid pid\n");
        return;
    }
    if (priority_ratio < 0 || arrival_time_ratio < 0 ||
executed_cycle_ratio < 0)
    {
        printf(1, "Invalid ratios\n");
        return;
    }
    int res = set_bjf_process(pid, priority_ratio, arrival_time_ratio,
executed_cycle_ratio);
    if (res < 0)
        printf(1, "Error setting BJF params\n");
```

```
    else
        printf(1, "BJF params set successfully\n");
}

int main(int argc,char* argv[]){
   if (argc < 5)
        {
            printf(1,"not enough params");
            exit();
        }
    set_process_bjf(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]),
atoi(argv[4]));
    exit();
}
```

خروجی این فراخوانی سیستمی به صورت زیر خواهد بود:

```
init: starting sh
Group #9
1. Narges Haeri
2. Ali Hamzehpour
3. Mina Shirazi
$ foo&
$ set_bjf_process 4 2 3 1
BJF params set successfully
$
```

۳. مقدار دهی پارامتر BJF در سطح سیستم:

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

```
void
set_bjf_system(int priority_ratio, int arrival_time_ratio, int
executed_cycles_ratio)
{
  acquire(&ptable.lock);
  struct proc* p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    p->sched_info.bjf_coeffs.priority_ratio = priority_ratio;
    p->sched_info.bjf_coeffs.arrival_time_ratio = arrival_time_ratio;
```

```
    p->sched_info.bjf_coeffs.executed_cycles_ratio =
executed_cycles_ratio;
  }
  release(&ptable.lock);
}
```

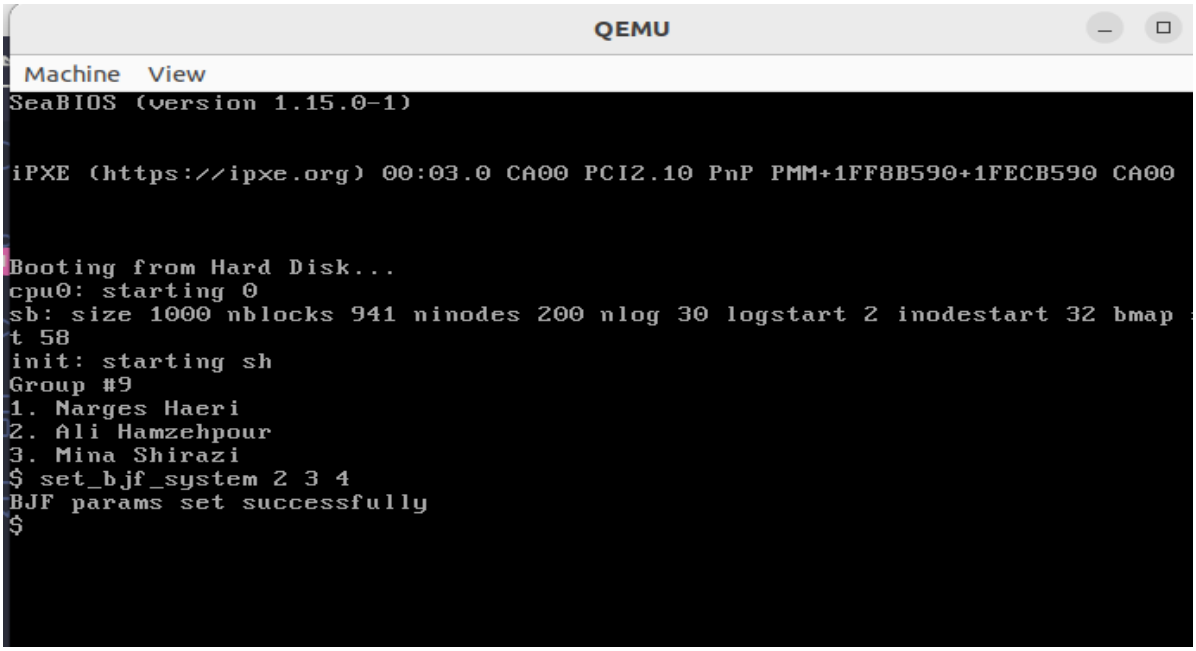
برنامه سطح کاربر زیر با نام set_bjf_system.c را برای تست این فراخوانی سیستمی ایجاد کردیم:

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

void set_system_bjf(int priority_ratio, int arrival_time_ratio, int
executed_cycle_ratio)
{
    if (priority_ratio < 0 || arrival_time_ratio < 0 ||
executed_cycle_ratio < 0)
    {
        printf(1, "Invalid ratios\n");
        return;
    }
    int res = set_bjf_system(priority_ratio, arrival_time_ratio,
executed_cycle_ratio);
    if (res < 0)
        printf(1, "Error setting BJF params\n");
    else
        printf(1, "BJF params set successfully\n");
}

int main(int argc,char* argv[]){
   if (argc < 4)
        {
            printf(1,"not enough params");
            exit();
        }
    set_system_bjf(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]));
    exit();
}
```

خروجی این فراخوانی سیستمی به صورت زیر خواهد بود:



٤. چاپ اطلاعات:

ابتدا فراخوانی سیستمی مورد نظر را با نام print_schedule_infoدر 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, 8, 9, 8, 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] - digitcount(p->pid));

    cprintf("%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.priority);
    printspaces(columns[6] - digitcount(p->sched_info.priority));

    cprintf("%d", (int)p->sched_info.bjf_coeffs.priority_ratio);
    printspaces(columns[7] - digitcount((int)p-
>sched_info.bjf_coeffs.priority_ratio));

    cprintf("%d", (int)p->sched_info.bjf_coeffs.arrival_time_ratio);
    printspaces(columns[8] - digitcount((int)p-
>sched_info.bjf_coeffs.arrival_time_ratio));

    cprintf("%d", (int)p->sched_info.bjf_coeffs.executed_cycles_ratio);
    printspaces(columns[9] - digitcount((int)p-
>sched_info.bjf_coeffs.executed_cycles_ratio));

    cprintf("%d", (int)p->sched_info.bjf_coeffs.process_size_ratio);
```

```
    printspaces(columns[10] - digitcount((int)p-
>sched_info.bjf_coeffs.process_size_ratio));

    cprintf("%d", (int)calc_process_bjf_rank(p));
    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 پردازه جدید ساخته شود و داخل هر پردازه عملیات محاسباتی طولانی و sleep طولانی قرار می‌دهیم:

```
#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(4000);
           int x = 0;
           for (long long j = 0; j < 1000000000000; j++){
                for (long long k = 0; k < 10000000000000; k++){
                    x += k * 12 - j;
                }
            }
            break;
        }
    }
    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_Prty  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 پردازه‌های foo تمام می‌شوند و جدول اطلاعات به شکل زیر می‌شود:

```
$ schedule_info
Process_Name    PID     State    Queue   Cycle   Arrival  Priority R_Prty  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
$
```