آزمایشگاه سیستم عامل **پروژه ۳**

سهیل حاجیان منش 810100119
امیرعلی رحیمی 810100146
مهدی نوری 810100231

github repo: https://github.com/MahdiNoori2003/OS-Lab-3
last commit: 374e54fd60f1d1e70e15e95420de4dae14913607

زمان بندی در xv6

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

بطور کلی در سه حالت ممکن است پردازه در حال اجرا تابع sched را فرخوانی کند:

1. پردازه بوسیله فراخوانی سیستمی exit قصد داشته باشد پردازنده را ترک کند.
2. پردازه بوسیله فراخوانی سیستمی sleep به خواب برود.
3. پردازه به دلیل تموم شدن تموم شدن تایمر زمان بند مجبور به ترک پردازنده شود.در این حالت تابع yield فراخوانی میگردد که درون آن تابع sched فراخوانی میشود.

در خود تابع sched عملیات تعویض متن صورت می گیرد که باعث میشود پردازنده از یک پردازه در حال اجرا به یک پردازه در حال اجرای دیگر سوییچ کند.

تابع sched این اعمال را انجام میدهد:

ابتدا قبل از شروع عملیات تعویض متن چند پیش شرط را چک میکند از جمله اینکه مطمئن میشود پردازه دیگری ptable را تغییر نمیدهد، پردازه در حالت اجرا قرار نداشته باشد و پردازنده در حین اجرا زمان بند وقفه پذیر نباشد FL_IF یک نباشد.

قبل از تعویض متن intena(interruped enabled state) پردازنده ی فعلی را ذخیره میکند.

حال با استفاده از تابع switch که در زبان اسمبلی پیاده سازی شده عملیات تعویض متن صورت میگیرد.این تابع context مربوط به پردازه فعلی را در p->context ذخیر میکند و context مربوط به cpu's scheduler را بازیابی میکند.در واقع پردازه ای که هر هسته را آماده به کار می کند، هیچ وقت از تابع scheduler خارج نمی شود و فقط با عملیات switching context از پردازنده خارج می شود و با اجرای تابع sched، دوباره به ادامه کار خود می پردازد.

زمان بندی

2. در زمان بند کاملا منصف در لینوکس، صف اجرا چه ساختاری دارد؟

در سیستم عامل لینوکس، زمان‌بند کاملاً منصف صف اجرا را با استفاده از ساختار داده‌ای به نام "Red-Black Tree" مدیریت می‌کند.
هر پردازه یک گره در Red-Black Tree است. هر گره حاوی داده‌هایی در مورد پردازه مربوط به آن است، مانند اولویت پردازه، زمانی که پردازه باید اجرا شود، و مقدار زمان صرف شده هنگام اخرین اجرا. زمان بند از این داده‌ها برای تصمیم‌گیری در مورد کدام پروسه باید بعدی اجرا شود استفاده می‌کند، با هدف تخصیص زمان پردازش به نحوی که به همه پردازه‌ها "منصفانه" خدمت شود - به این معنی که هر پردازه، نسبت به اولویتش، بخش منصفانه‌ای از CPU را دریافت می‌کند.
زمان‌بند کاملا منصفانه برای اینکه تضمین کند همه پردازه ها زمان CPU دریافت کنند از مفهومی به نام nice value استفاده می‌کند. هر پردازه با یک nice value مشخص می‌شود که تعیین می‌کند با چه ترتیبی CPU به پردازه‌ها تخصیص پیدا کند. پردازه‌هایی با nice value بالاتر (اولویت پایین‌تر) کمتر CPU دریافت می‌کنند در حالی که پروسه‌های با nice value پایین‌تر (اولویت بالاتر)، زمان بیشتری برای اجرا دارند.در چپ ترین گره این درخت پردازه ای قرار گرفته که کمترین برش زمانی در حین اجرا را داشته باشد.

3. بررسی لینوکس و 6xv از منظر مشترک یا مجزا بودن صفهای زمانبندی

در مقایسه سیستم عامل‌های xv6 و لینوکس با توجه به صف‌های زمان‌بندی هسته‌های پردازنده، دو رویکرد می‌توانیم داشته باشیم: زمان‌بندهای مشترک و زمان‌بندهای مجزا برای هر هسته. در روش مشترک همه هسته‌ها از یک صف زمان‌بندی مشترک استفاده می‌کنند، در حالی که در روش مجزا هر هسته یک صف زمان‌بندی اختصاصی خود را دارد.

xv6:

در xv6 فقط از یک صف زمان بندی برای همه پردازنده ها استفاده میشود مطابق شکل زیر:

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

این صف همزمان به تعداد ۶۴ پردازه را می تواند در خود نگه دارد. برای جلوگیری از مشکلات احتمالی که دسترسی همزمان تمام پردازه ها به صف میتواند میتواند داشته باشد از spinlock استفاده شده است.هر پردازنده برای دسترسی به ptable.proc ابتدا باید ptable.lock را acquire کند و پس از تمام شدن کارش هم آن را release کند.

لینوکس:

لینوکس از CFS(completely fair scheduler) استفاده می‌کند، که یک زمان‌بند عادلانه است و برای هر هسته یک صف زمان‌بندی اختصاصی دارد. این صف‌ها به صورت جداگانه مدیریت می‌شوند اما همگرایی دارند به این معنی که اطلاعاتی در میان هسته‌ها به اشتراک گذاشته می‌شوند تا تصمیم‌های زمان‌بندی بهتری گرفته شوند.

مزیت صف مشترک:

یکی از مزیت‌های اصلی صف‌های زمان‌بندی مشترک این است که بار پردازشی را می‌توان به شکلی انعطاف‌پذیر بین هسته‌ها جابجا کرد، زیرا همه هسته‌ها به مجموعه پروسه‌های یکسانی دسترسی دارند. این امر می‌تواند به افزایش کارایی و استفاده بهینه از منابع CPU کمک کند.

نقص صف مشترک:

وقتی که هسته‌های مختلف برای دسترسی به صف زمان‌بندی مشترک رقابت می‌کنند، ممکن است تداخلات lock (قفل) رخ دهد که می‌تواند منجر به افزایش زمان انتظار برای دسترسی به صف و بنابراین کاهش کارایی کلی شود.

مزیت صف مجزا:

با داشتن زمان‌بندهای مجزا، هر هسته می‌تواند پروسه‌های خود را به طور مستقل مدیریت کند، بدون اینکه نیاز به انتظار برای دسترسی به صف مشترک باشد. این می‌تواند به کاهش تداخل lock و بهبود کارایی کلی بینجامد.

نقص صف مجزا:

وقتی هر هسته دارای صف زمان‌بندی مجزای خود است، ممکن است توازن بار بین هسته‌ها به خوبی صف مشترک اجرا نشود. گاهی اوقات ممکن است یک هسته دارای بار زیادی باشد در حالی که هسته دیگری بیکار باشد. این می‌تواند منجر به کارایی کمتر در استفاده از منابع موجود شود.

4. بررسی دلیل فعال گردیدن وقفه در ابتدای هربار اجرای حلقه. آیا در سیستم تک هسته ای به آن نیاز است؟

دستور ()sti (که مخفف Set Interrupt Flag بوده و به معنی فعال‌سازی وقفه‌ها است) به منظور فعال‌سازی امکان پاسخ‌دهی به وقفه‌های سخت‌افزاری قبل از شروع اسکن کردن جدول پروسه‌ها (ptable) و یافتن پروسه‌ای که قابل اجرا است، اجرا می‌شود. دلیل فعال‌سازی وقفه‌ها در این نقطه این است که زمانی که ptable قفل میشود تمامی وقفه ها بوسیله تابع pushcli غیرفعال میشوند.حال حالتی را فرض کنید که تعدادی از پردازده ها در حال انجام عملیات (I/O) باشند و همچنین پردازه ای هم آماده اجرا در صف ptable نباشد در این حالت هیچ پردازه ای اجرا نمیشود و اگر وقفه ها هم فعال نشوند نمیتوانیم پردازه هایی که عملیات (I/O) را انجام داده اند به حالت RUNNABLE تغییر دهیم که اجرا شوند پس در نتیجه سیستم در همین وضعیت می ماند.به همین دلیل در ابتدا هر بار اجرا حلقه برای مدت کوتاهی وقفه ها فعال میگردند.
حتی اگر سخت‌افزار تک‌هسته‌ای باشد، توانایی پردازش وقفه‌ها هنوز هم ضروری است زیرا اموری مانند پردازش I/O معمولاً به وقفه‌ها برای گزارش پایان یک عملیات متکی هستند. بدون امکان وقفه، یک هسته ممکن است در چرخه بی‌نهایتی از جستجوی پروسه‌ای برای اجرا گیر کند و دیگر عملیات‌ها که به وقفه‌های سخت‌افزاری نیاز دارند ممکن است به تأخیر افتند یا کاملاً نادیده گرفته شوند که می‌تواند منجر به کندی یا عدم پاسخگویی سیستم شود.

5.1. دو سطح مدیریت وقفه ها در لینوکس

در لینوکس، مدیریت وقفه‌ها به طور کلی در دو سطح انجام می‌گیرد:
۱. Top Half (بخش بالایی یا اصلی وقفه‌ها): این بخش برای پردازش فوری وقفه‌ها در نظر گرفته شده است. زمانی که یک وقفه رخ می‌دهد، ابتدا بخش بالایی وقفه به اجرا در می‌آید. این بخش فقط شامل کدهایی است که باید بلافاصله اجرا شوند، مثلاً ذخیره کردن مقادیر فعلی رجیسترها و انتقال داده‌ها به یک buffer. این کدها معمولاً بسیار مختصر هستند و باید سریع اجرا شوند تا از ازدحام وقفه‌ها جلوگیری شود. بخش بالایی وقفه‌ها دارای بالاترین اولویت هستند و معمولاً حین اجرای آن‌ها وقفه‌های دیگر غیرفعال می‌شوند. در این سطح یا وقفه ها به طور کامل سرویس دهی می شوند و یا اطلاعات ضروری وقفه - که فقط در زمان وقوع وقفه در دسترسی است - ذخیره شده و یک Bottom Half را برای مدیریت کامل این وقفه زمانبندی می کند.
۲. Bottom Half (بخش پایینی یا تکمیلی وقفه‌ها): به مکانیسم‌هایی مانند tasklets و work queues گفته می‌شود که برای پردازش‌هایی استفاده می‌شوند که نیاز به وقت گیری بیشتری دارند و فوریت کمتری نسبت به بخش بالایی دارند. بخش پایینی پس از بخش بالایی اجرا می‌شود و می‌تواند در زمانی که سیستم فرصت دارد فعالیت کند و مانند یه پردازه انجام میشود.Bottom Half

ها در یک صف اجرا قرار گرفته و منتظر پردازنده می مانند. از آنجا که ممکن است زمان طولانی برای اجرای آنها نیاز باشد، Bottom Half ها نیز معمولا مانند ریسه ها و پردازه ها زمانبندی می شوند. از نظر اولویت، بخش بالایی وقفه‌ها اولویت بالاتری نسبت به بخش پایینی دارند، زیرا آن‌ها بلافاصله وقفه‌ها را پردازش می‌کنند و می‌توانند وقفه‌های دیگر را نیز مسدود کنند. این در حالی است که بخش پایینی وقفه‌ها که اغلب به صورت هم‌زمان با پردازه‌های عادی اجرا می‌شوند، اولویت پایین‌تری دارند و می‌توانند به وسیله بخش بالایی وقفه متوقف شوند. بخش پایینی به طور کلی طولانی‌تر و کم اولویت‌تر است. هدف اصلی این تفکیک این است که اطمینان حاصل شود پردازش وقفه‌ها بدون ایجاد تاخیر یا اخلال در پردازش سایر وقفه‌های ضروری، به شکل منظم و کنترل شده ادامه پیدا کند.

5.2. مشکل گرسنگی پردازه ها چگونه حل شده است؟

در سیستم‌های بی‌درنگ ، که در آن‌ها پاسخ‌گویی و اجرای دستورالعمل‌ها در زمان مشخصی بسیار مهم است، مدیریت وقفه‌ها پیچیدگی خاص خود را دارد. پردازش طولانی وقفه‌ها می‌تواند به گرسنگی پردازه‌هایی که مدت هاست در صف انتظار برای دریافت پردازنده هستند منجر شود و اهداف سیستم بی‌درنگ را نقض کند. برای حل این مشکل، سیستم‌های بی‌درنگ اغلب از چندین استراتژی بهره می‌برند:

1. **اولویت‌بندی وقفه‌ها (Interrupt Prioritization)**: وقفه‌ها اولویت‌های مختلفی داده می‌شوند. برخی از پلتفرم‌ها امکان تعیین اولویت سخت‌افزاری وقفه‌ها را فراهم می‌آورند، به این صورت که وقفه‌های حیاتی با اولویت بالاتر پردازش می‌شوند و وقفه‌های کم‌اهمیت‌تر انتظار می‌کشند.

2. **استفاده از سطح پایین‌تر وقفه‌ها (Bottom Halves)**: کارهای زمان‌بر در توابع سرویس‌دهی وقفه به بخش‌های پایین‌تر (bottom halves) که وقفه‌های مجدد را بلوکه نمی‌کنند، منتقل می‌شوند.

3. **خدمت‌رسانی محدود وقفه‌ها (Interrupt Throttling)**: محدود کردن تعداد دفعاتی که یک وقفه در یک دوره زمانی خاص قادر به اختلال‌اندازی است، این اطمینان را می‌دهد که کدهای دیگر نیز فرصت اجرا پیدا می‌کنند.

4.**چک کردن دوره‌ای اتفاقات (polling for events)** به جای استفاده از وقفه ها

زمان بندی بازخوردی چند سطحی

تابع زیر را برای قسمت aging نوشته ایم:

```
void age_proc(int uptime_ticks)
{
  acquire(&ptable.lock);

  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[i].state == RUNNABLE && ptable.proc[i].queue != RR)
    {
      if (uptime_ticks - ptable.proc[i].last_run > CHANGE_QUEUE_THRESHOLD)
      {
        release(&ptable.lock);
        change_queue(ptable.proc[i].pid, RR);
        acquire(&ptable.lock);
      }
    }
  }
  release(&ptable.lock);
}
```

ابتدا ptable را acquire میکنیم سپس در یک حلقه تمامی پردازه های موجود را بررسی میکنیم شرط اولیه بررسی میکند که ابتدا پردازه در حالت RUNNABLE باشد و اینکه در صف roundrobin نباشد(چون بالاترین سطح صفوف است و دیگر به صفی با اولویت بالاتر نمی توان آن را انتقال داد) سپس زمان uptime_tick (زمان بالا بودن سیستم عامل) را از زمان آخرین اجرای پردازه کم میکنیم. اگر این مقدار از CHANGE_QUEUE_THRESHOLD بیشتر بود پردازه را با فراخوانی تابع change_queue به صف roundrobin منتقل میکنیم. در انتها هم دوباره ptable را release میکنیم. مقدار ptable.proc.last_run پس از هر بار زمان بندی در تابع scheduler برای پردازه hd که قرار است اجرا شود، مقداردهی می شود.

استراکت bjf_info را به صورت زیر تعریف کرده ایم که شامل ضرایب و سه فیلد مربوط به محاسبه ضریب BJf است. پارامتر proccess_size به صورت پیش فرض برای تمامی پردازه ها توسط سیستم عامل تعیین شده است.

```
struct bjf_info
{
  int priority;
  int arrival_time;
  float priority_ratio;
  float arrival_time_ratio;
  float executed_cycle;
  float executed_cycle_ratio;
  float process_size_ratio;
};
```

همچنین به استراکت proc اطلاعات زیر را برای هر پردازه اضافه می کنیم:
که شامل استراکت bjf_info که در بالا تعریف شد و صفی که پردازه در آن قرار می گیرد به همراه آخرین زمانی که پردازه اجرا شده است و همچنین زمان ورودش به صف lcfs می شود.

```
struct bjf_info bjf_info;
int queue;
int last_run;
int last_in_lcfs;
```

define های زیر را هم به صورت زیر تعریف می کنیم که شامل حالت های صفی است که یک پردازه می تواند داشته باشد.RR برای پردازه ای که در صف roundrobin است، LCFS پردازه ای که در صف lcfs است و BJF پردازه ای که در صف bjf می باشد.

```
#define RR 1
#define LCFS 2
#define BJF 3
```

1.زمان بند نوبت گردشی(roundrobin):

تابع زیر برای این زمان بند نوشته شده است.پارامتر last_scheduled_rr در واقع آخرین پردازه ای است که توسط صف roundrobin زمان بندی شده است. در این تابع از آن پردازه به بعد به دنبال پردازه ای قابل اجرا(RUNNABLE) می گردیم و درصورتی که پیدا کردیم آن را برمی گردانیم و در صورت اینکه چنین پردازه ای در صف پیدا نشد و دوباره به پردازه last_scheduled_rr رسیدیم مقدار صفر را برمیگردانیم.

```
struct proc *roundrobin(struct proc *last_scheduled_rr)
{
  struct proc *p = last_scheduled_rr;

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

    if (p->state == RUNNABLE && p->queue == RR)
      return p;

    if (p == last_scheduled_rr)
      return 0;
  }
}
```

2. زمان بند آخرین ورود-اولین رسیدگی(lcfs):

تابع زیر را برای این زمان بند نوشته ایم.پارامتر last_schedued_lcfs آخرین پردازه ایست که در lcfs زمان بندی شده است.ابتدا چک میکنیم اگر این پردازه در حالت RUNNABLE بود یعنی هنوز کارش تمام نشد بوده همان پردازه را برمی گردانیم چون در lcfs پردازه بعدی تنها پس از اجرا کامل آخرین پردازه اجرا میشود.

سپس از بین تمام پردازه های RUNNABLE که در صف lcfs قرار دارند پردازه ای که دیر تر از همه وارد صف شده است را پیدا میکنیم و برمیگردانیم.

```
struct proc *lcfs(struct proc *last_scheduled_lcfs)
{
  struct proc *result = 0;
  int max_arrival_time = -1;

  if (last_scheduled_lcfs != 0 && last_scheduled_lcfs->state == RUNNABLE)
  {
    return last_scheduled_lcfs;
  }

  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[i].state == RUNNABLE && ptable.proc[i].queue == LCFS)
    {
      if (ptable.proc[i].last_in_lcfs >= max_arrival_time)
      {
        max_arrival_time = ptable.proc[i].last_in_lcfs;
        result = &ptable.proc[i];
      }
    }
  }
  return result;
}
```

3. زمان بند اول بهترین کار(bjf):

تابع bjf مربوط به این زمان بند است.ابتدا فیلد های مربوط به چهار معیار به همراه ضرایب آنها به هر پردازه اضافه شده است.فیلد priority عددی بین 1 تا 5 است که 1 نشان دهنده بالاترین اولویت است.در این تابع بین تابع پردازه های RUNNABLE که صف آنها bjf است پردازه با کمترین رنک را پیدا کرده و برمیگردانیم.تابع bjfrank با توجه به چهار فیلد و ضریب های مربوط مقدار رنک را برای پردازه ورودی محاسبه میکند.

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

struct proc *bjf()
{
  struct proc *result = 0;
  float min_rank = MIN_RANK;

  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[i].state == RUNNABLE && ptable.proc[i].queue != BJF)
    {
      float rank = bjfrank(&ptable.proc[i]);
      if (result == 0 || rank < min_rank)
      {
        result = &ptable.proc[i];
        min_rank = rank;
      }
    }
  }
  return result;
}
```

فراخوانی سیستمی

1. تغییر صف پردازه

تابع change_queue مربوط به این هدف در زیر آورده شده است:

```
int change_queue(int pid, int new_queue)
{
  int old_queue = -1;

  acquire(&ptable.lock);
  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[i].pid == pid)
    {
      old_queue = ptable.proc[i].queue;
      if (old_queue == new_queue)
      {
        release(&ptable.lock);
        return -1;
      }
      ptable.proc[i].queue = new_queue;

      if (new_queue == LCFS)
      {
        ptable.proc[i].last_in_lcfs = ticks;
      }
      release(&ptable.lock);
      return old_queue;
    }
  }

  release(&ptable.lock);
  return old_queue;
}
```

در حلقه پردازه مربوط به pid ورودی را پیدا میکنیم و حالت صف آن را برابر new_queue می گذاریم و اگر صف جدید آن lcfs بود مقدار last_in_lcfs آن را برابر زمان سیستم عامل قرار می دهیم.

2.مقدار دهی پارامتر BJF در سطح پردازه:

کد مربوط به این فراخوانی سیستمی در زیر گذاشته شده است. ورودی چهار ضریب مربوط را همراه pid پردازه مورد نظر میگیرد و این مقادیر را برای آن پردازه ست میکند.

```
int set_bjf_params_for_process(int pid, float priority_ratio, float arrival_time_ratio, float executed_cycles_ratio, float process_size_ratio)
{
  acquire(&ptable.lock);
  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[i].pid == pid)
    {
      ptable.proc[i].bjf_info.priority_ratio = priority_ratio;
      ptable.proc[i].bjf_info.arrival_time_ratio = arrival_time_ratio;
      ptable.proc[i].bjf_info.executed_cycle_ratio = executed_cycles_ratio;
      ptable.proc[i].bjf_info.process_size_ratio = process_size_ratio;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

3.مقدار دهی پارامتر BJF در سطح سیستم:

کد مربوط به این فراخوانی سیستمی در زیر گذاشته شده است. ضرایب در معادله BJF را برای تمامی پردازه ها ست می کند.

```
void set_bjf_params_for_system(float priority_ratio, float arrival_time_ratio, float executed_cycles_ratio, float process_size_ratio)
{
  acquire(&ptable.lock);

  for (int i = 0; i < NPROC; i++)
  {
    ptable.proc[i].bjf_info.priority_ratio = priority_ratio;
    ptable.proc[i].bjf_info.arrival_time_ratio = arrival_time_ratio;
    ptable.proc[i].bjf_info.executed_cycle_ratio = executed_cycles_ratio;
    ptable.proc[i].bjf_info.process_size_ratio = process_size_ratio;
  }
  release(&ptable.lock);
}
```

4. چاپ اطلاعات:

در این فراخوانی سیستمی که کد آن در ادامه آورده شده است بر روی تمام پردازه ها پیمایش می شود و تمام اطلاعات آن ها چاپ میشود.

```
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 Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank\n"
          "-------------------------------------------------------------------------------------------------------\n");
  acquire(&ptable.lock);
  for (int i = 0; i < NPROC; i++)
  {
    if (ptable.proc[i].state == UNUSED)
      continue;

    const char *state;
    if (ptable.proc[i].state >= 0 && ptable.proc[i].state < NELEM(states) && states[ptable.proc[i].state])
      state = states[ptable.proc[i].state];
    else
      state = "???";

    cprintf("%s", ptable.proc[i].name);
    spacer(columns[0] - strlen(ptable.proc[i].name));

    cprintf("%d", ptable.proc[i].pid);
    spacer(columns[1] - count_digits(ptable.proc[i].pid));

    cprintf("%s", state);
    spacer(columns[2] - strlen(state));

    cprintf("%d", (ptable.proc[i].queue) + 1);
    spacer(columns[3] - count_digits(ptable.proc[i].queue));
```

```
    cprintf("%d", (int)ptable.proc[i].bjf_info.executed_cycle);
    spacer(columns[4] - count_digits((int)ptable.proc[i].bjf_info.executed_cycle));

    cprintf("%d", ptable.proc[i].bjf_info.arrival_time);
    spacer(columns[5] - count_digits(ptable.proc[i].bjf_info.arrival_time));

    cprintf("%d", ptable.proc[i].bjf_info.priority);
    spacer(columns[6] - count_digits(ptable.proc[i].bjf_info.priority));

    cprintf("%d", (int)ptable.proc[i].bjf_info.priority_ratio);
    spacer(columns[7] - count_digits((int)ptable.proc[i].bjf_info.priority_ratio));

    cprintf("%d", (int)ptable.proc[i].bjf_info.arrival_time_ratio);
    spacer(columns[8] - count_digits((int)ptable.proc[i].bjf_info.arrival_time_ratio));

    cprintf("%d", (int)ptable.proc[i].bjf_info.executed_cycle_ratio);
    spacer(columns[9] - count_digits((int)ptable.proc[i].bjf_info.executed_cycle_ratio));

    cprintf("%d", (int)ptable.proc[i].bjf_info.process_size_ratio);
    spacer(columns[10] - count_digits((int)ptable.proc[i].bjf_info.process_size_ratio));

    cprintf("%d", (int)bjfrank(&ptable.proc[i]));
    if (i != NPROC - 1)
      cprintf("\n");
  }
  release(&ptable.lock);
}
```

برنامه سطح کاربر

یک برنامه سطح کاربر به اسم proc_info نوشته ایم که به کاربر فراخوانی های سیستمی همراه با آرگومان های مورد نیاز هر کدام را نمایش می دهد تا از بین آنها هرکدام را می خواهد وارد کند.

نحوه اجرای این برنامه به صورت زیر است:

```
$ proc_info
usage: command <inputs>
list of commands:
    info
    change_queue <pid> <new_queue>
    set_bjf_process <pid> <priority_ratio> <arrival_time_ratio> <executed_cycles_ratio> <process_size_ratio>
    set_bjf_system <priority_ratio> <arrival_time_ratio> <executed_cycles_ratio> <process_size_ratio>
```

(اجرای proc_info)

```
$ proc_info 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 1       1       3       3       1       1       1       1       16391
proc_info       3       running  1       1       566     3       1       1       1       1       16954
$ proc_info set_bjf_process 2 1 2 3 4
the process with id 2 has changed bjf parameters successfully
```

```
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 1       2       3       3       1       2       3       4       65551
proc_info       5       running  1       0       4060    3       1       1       1       1       20447
```

(اجرای proc_info برای ست کردن پارامتر های یک پراسس)

```
$ proc_info set_bjf_system 1 2 3 4
bjf parameters for system has changed successfully
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       2       3       4       49159
sh              2       sleeping 1       2       3       3       1       2       3       4       65552
proc_info       4       running  1       0       2010    3       1       1       1       1       18397
```

(اجرای proc_info برای ست کردن پارامتر های سیستم)

برنامه foo نیز که کد آن را در زیر قرار داده ام نوشته ایم که پنج پردازه می سازد که هر کدام محاسباتی طولانی را انجام می دهند برای این که بتوان فرصت کافی داشت تا عملکرد زمان بند را بررسی کرد.

```c
int main()
{
    change_queue(getpid(), 2);
    for (int i = 0; i < PROCS_NUM; ++i)
    {
        int pid = fork();
        if (pid > 0)
            continue;
        if (pid == 0)
        {
            sleep(2000);
            for (int j = 0; j < 100 * i; ++j)
            {
                int x = 1;
                for (long k = 0; k < 1000000000000; ++k)
                    x++;
            }
            exit();
        }
    }
    while (wait() != -1)
        ;
    exit();
}
```

خروجی هایی از اجرای برنامه به شکل زیر است :

```
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       1       1       1       1       12293
sh              2       sleeping 1       2       4       3       1       1       1       1       16393
foo             5       runnable 2       111     555     3       1       1       1       1       12957
foo             4       sleeping 2       0       554     3       1       1       1       1       12845
foo             6       runnable 2       111     555     3       1       1       1       1       12957
foo             7       runnable 2       111     555     3       1       1       1       1       12957
foo             8       runnable 2       111     555     3       1       1       1       1       12957
foo             9       runnable 2       111     555     3       1       1       1       1       12957
proc_info       11      running  1       0       1667    3       1       1       1       1       18054
```

```
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       1       1       1       1       12293
sh              2       sleeping 1       2       4       3       1       1       1       1       16393
foo             5       runnable 2       163     555     3       1       1       1       1       13009
foo             4       sleeping 2       0       554     3       1       1       1       1       12845
foo             6       runnable 2       163     555     3       1       1       1       1       13009
foo             7       runnable 2       163     555     3       1       1       1       1       13009
foo             8       runnable 2       163     555     3       1       1       1       1       13009
foo             9       runnable 2       163     555     3       1       1       1       1       13009
proc_info       12      running  1       0       2190    3       1       1       1       1       18577
```

(اجرای برنامه با sleep)

```
Group #31:
1. Mahdi Noori
2. AmirAli Rahimi
3. Soheil Hajian
$ foo&
$ proc_info info
Process_Name    PID     State    Queue   Cycle   Arrival Priority R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------
init            1       sleeping 1       2       0       3       1       1       1       1       12293
sh              2       sleeping 1       2       4       3       1       1       1       1       16393
foo             5       runnable 2       0       246     3       1       1       1       1       12537
foo             4       sleeping 2       0       245     3       1       1       1       1       12536
foo             6       runnable 2       0       246     3       1       1       1       1       12537
foo             7       runnable 2       0       246     3       1       1       1       1       12537
foo             8       runnable 2       0       246     3       1       1       1       1       12537
foo             9       runnable 2       45      246     3       1       1       1       1       12582
proc_info       10      running  1       1       691     3       1       1       1       1       17079
```

(اجرای برنامه بدون sleep)