به نام خدا

پروژه سوم آزمایشگاه سیستم عامل

استاد کارگهی

اعضای گروه:

سارا رضایی‌منش 810198576

نرگس غلامی 810198447

سوالات تشریحی

.1

فرآیندی که می‌خواهد CPU را رها کند باید قفل جدول پردازش (ptable.lock) را دریافت کند، هر قفل دیگری را که نگه داشته است آزاد کند، وضعیت خود را به‌روزرسانی کند (proc->state) و سپس sched را فراخوانی کند.

Sched آن شرایط را دوباره بررسی می کند و سپس پیامد آن شرایط را بررسی می کند: تا وقتی که یک قفل نگه داشته می‌شود، CPU باید با interrupts های غیرفعال در حال اجرا باشد. در آخر sched ,سوییچ (swtch) را فراخوانی می‌کند تا current context را در proc->context ذخیره نماید و سپس به scheduler context در cpu->scheduler سوییچ می‌کند. از آنجایی که scheduler تا دستور swtch اجرا شده بود، این خط هنگام فراخوانی دستور swtch در scheduler در pcb این پردازه ذخیره شده است. و پس از بازیابی pcb آن، مجدد از خط بعد اجرای scheduler ادامه پیدا می کند.

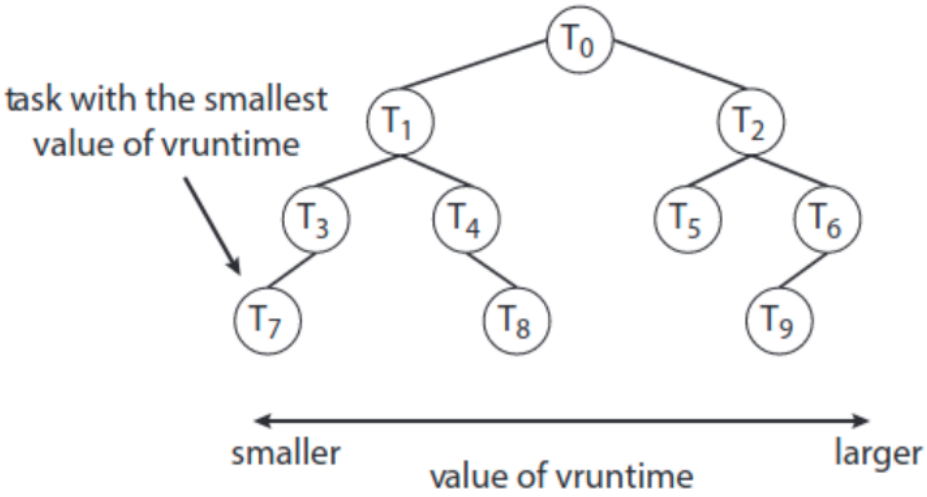
این تابع از ابتدایی که CPU خودش را اجرا می‌کند توسط آن فراخوانی می‌شود و دائما در حال ران شدن است چون داخل یک فور بدون شرط قرار دارد. در هر حلقه به این صورت عمل می‌کند که پراسس مناسب برای اجرا را انتخاب می‌کند و با تابع swtch وضعیت قبلی خود را ذخیره کرده و کنترل cpu را به آن پراسس می‌دهد. هنگامی که کار آن پراسس تمام می‌شود با استفاده از تابع swtch تابع sched کنترل به scheduler برمی‌گرداند و این یعنی کار پراسس تمام شده و تابع scheduler باید پراسس بعدی برای ران شدن را انتخاب کند.

```
void
scheduler(void)
{
  struct proc *p;
  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);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        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;
      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);

  }
}
```

```
void
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;
}
```

.۲

در CFS اولویت به تسکی داده می شود که virtual runtime کمتری داشته باشد. Virtual runtime یا vruntime به nice value و physical runtime یک پردازه بستگی دارد که این مفاهیم در صفحه ۲۷۶ کتاب مرجع توضیح داده شده اند. اگر از یک queue برای نگهداری پردازه ها استفاده می کردیم، برای پیدا کردن پردازه با بیشترین vruntime باید هر دفعه کل صف را پیمایش می‌کردیم که زمان زیادی می‌برد. به همین علت سیستم عامل لینوکس برای نگهداری پردازه های آماده اجرا به جای داده ساختار صف، از داده ساختار درخت قرمز سیاه استفاده می کند که در آن key هر پردازه، همان vruntime آن است.



اگر تسکی قابل اجرا شود، به درخت اضافه می شود و در غیر اینصورت(از حالت قابل اجرا به علت عملیات I/O و... خارج شود) از درخت حذف می‌گردد. به صورت کلی تسک هایی که زمان کمتری برای اجرا به آنها اختصاص گرفته(زمان هایی که vruntime کمتری دارند) در سمت چپ درخت و تسک هایی که زمان اجرای بیشتر به آنها اختصاص گرفته در سمت راست قرار دارند. بر اساس خصوصیات درخت جست و جوی دودویی، می دانیم چپ ترین نود در درخت، کمترین مقدار key را دارد. همچنین می دانیم که الگوریتم CFS بیشترین اولویت را به این نود می دهد. از آنجایی که درخت قرمز سیاه balanced است، پیدا کردن این نود در درخت پیچیدگی زمانی O(logn) دارد. اما برای افزایش بهینگی و سرعت پردازنده، scheduler سیستم عامل لینوکس این مقدار را در متغیری به نام rb_leftmost ذخیره می‌کند. در نتیجه برای تعیین اینکه چه تسکی باید بعد از تسک فعلی اجرا شود تنها کافیست مقدار این متغیر را بخوانیم.(اردر زمانی ثابت)

.3

در سیستم های چند هسته ای، پیاده سازی scheduler به شیوه CFS سخت تر می شود. برای اینکه بتوانیم از مزایای scalability(تقسیم thread ها بین هسته ها و تسریع اجرای هر پردازه) استفاده کنیم، باید هسته ها از صف های جداگانه استفاده کنند. هدف از استفاده از صف های جداگانه این است که هر هسته هنگام context switch که به دنبال ریسمانی برای اجرا است، فقط به صف اجرای محلی خود دسترسی داشته باشد. از آنجایی که فرآیند تعویض متن در مسیر بحرانی قرار دارد، باید سریع انجام شود و اینکه هسته تنها به صف اجرایی خود دسترسی داشته باشد، از اینکه scheduler دسترسی های پرهزینه و synchronized داشته باشد جلوگیری می کند. این دسترسی ها در صورتی انجام می شوند که یک صف اجرای مشترک بین هسته ها وجود داشته باشد که از racing جلوگیری کنند و بین پردازه ها برای دسترسی به صف نظم برقرار شود.

اما برای اینکه در حالت وجود صف های اجرای جدا، الگوریتم زمانبند بتواند به درستی و به صورت بهینه کار کند، باید آنها را balanced(تقسیم بار کاری) نگه داریم. عیب این روش این است که load balancing در سیستم های چند هسته ای هم از لحاظ محاسباتی و هم از لحاظ ارتباطات بسیار هزینه بر است. چراکه برای این کار باید روی هزاران صف اجرا گردش کنیم(هزینه محاسباتی) و برخی از داده ساختارهای ذخیره شده را تغییر دهیم که باعث cache miss ها و synchronization های پرهزینه می شود.(هزینه ارتباطات)

پس می‌توان نتیجه گرفت مزیت صف مشترک نسبت به صف مجزا این است که دیگر به الگوریتم های پرهزینه ای مانند load balancing برای استفاده بهینه از ظرفیت پردازنده ها احتیاجی نداریم و کار به درستی بین هسته ها تقسیم می شود و مشکلاتی از قبیل اینکه یک هسته بیکار بماند یا cpu به پردازه های کم اولویت تر اختصاص داده شود اتفاق نمی‌افتد.

و عیب آن نسبت به صف مجزا این است که هر هسته برای انتخاب پردازه بعدی برای اجرا باید بین thread های بسیار زیادی گردش کند و دسترسی های پرهزینه انجام دهد. این دسترسی های پرهزینه به این علت اتفاق می افتند که همه پردازنده ها به یک صف دسترسی دارند و مشکل racing پدید می آید و برای جلوگیری از racing از lock ها استفاده می کنند که هر پردازنده هنگام دسترسی به صف اجرای مشترک از دسترسی بقیه پردازه ها به آن جلوگیری کند. اما استفاده از lock ها به صورت کلی بسیار سخت است. چرا که این امکان باید برای همه پردازنده ها فراهم شود و در نتیجه ممکن است دسترسی به صف اجرا به bottleneck فرآیند اجرا تبدیل شود. که در این صورت سربار context switch بسیار زیاد می شود و بخشی از مزیت scalability مربوط به multithreading بلا استفاده می گردد.

.4

زمانبند، ptable.lock را برای اکثر اقدامات خود نگه می دارد، علت این کار وجود racing بین CPU ها میباشد.(CPU ها می خواهند همزمان به صف پراسس ها دسترسی داشته باشند و ممکن است چند سی پی یو بخواهند همزمان یک پراسس را بردارند یا تغییراتی در صف پردازه ها به وجود بیاید که CPU های دیگر که همزمان دسترسی پیدا کرده اند از آن آگاه نشوند.)به این صورت که هر CPU که به پراسس ها نیاز دارد باید حتما ptable.lock آن را قفل بکند و از آن طرف در هر تکرار حلقه بیرونی lock را آزاد می کند (وقفه ها را فعال می کند). این برای موارد خاصی که در آن CPU بیکار است (نمی تواند هیچ فرآیند RUNNABLE ای پیدا کند) مهم است. اگر یک idle scheduler با نگه داشتن قفل به طور مداوم حلقه می‌زد، هیچ CPU دیگری که یک پراسس را اجرا می‌کند امکان context switch یا هر فراخوانی سیستمی دیگر را نخواهد داشت و همچنین هیچوقت نمی‌توانست یک فرآیند RUNNABLE به وجود بیاورد تا وقتی که idling CPU از scheduling loop خارج شود.

در سیستم های تک هسته ای: دلیل فعال کردن وقفه های دوره ای در یک CPU بیکار این است که ممکن است پراسس قابل اجرا وجود نداشته باشد زیرا فرآیندها در انتظار I/O هستند. اگر زمان‌بند وقفه‌ها را همواره غیرفعال نگه دارد، I/O هرگز نمی‌رسد.

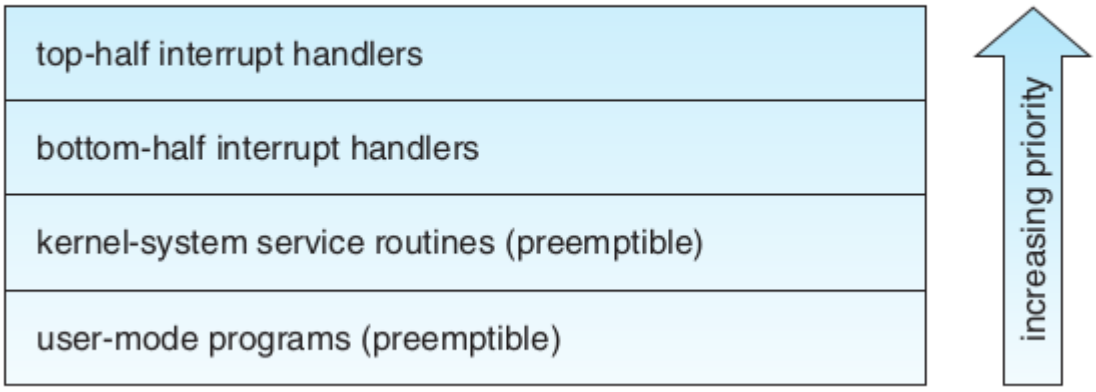
.5

کنترل کننده وقفه باید به سرعت اجرا شود، زیرا از اجرای هر وقفه دیگری جلوگیری می کند. در هسته لینوکس، پردازش وقفه به دو بخش تقسیم می شود:

"نیمه بالایی" کنترل کننده وقفه است. حداقل کارهای لازم را انجام می دهد، معمولاً با سخت افزار ارتباط برقرار می کند و یک flag در جایی در حافظه هسته تنظیم می کند.

"نیمه پایین" هر پردازش ضروری دیگری را انجام می دهد، به عنوان مثال کپی داده ها در حافظه پردازش، به روز رسانی ساختارهای داده هسته، و غیره. ممکن است زمان خود را ببرد و حتی انتظار برای قسمت دیگری از سیستم را مسدود کند تا وقتی که وقفه ها فعال شوند.

با توجه به توضیحات داده شده اولویت interrrupt نسبت به پراسس بالاتر و همچنین در خود کنترل کننده وقفه ها اولویت top half بالاتر است. این عکس خلاصه‌ای از جواب می‌باشد.



بیشتر وقفه ها را می توان به صورت موقت غیرفعال کرد و زمان دیگری به آنها رسیدگی نمود. در صورتی که interrupt ها برای مدت طولانی غیر فعال باشند، همه جمع می شوند و هنگامی که توسط سیستم عامل فعال شوند، توسط interrupt handler به آنها رسیدگی می شود. اگر تعداد وقفه های جمع شده به علت فعال نبودن وقفه ها به مدت طولانی زیاد باشد، سیستم مجبور می شود مدت زمان زیادی منتظر پاسخ interrupt handler بماند و interrupt latency افزایش می یابد. برای جلوگیری از این اتفاق، در سیستم عامل لینوکس از deferrable action ها استفاده می شود. با استفاده Deferrable action ها می توان اجرای یک تابع را برای زمان دیگری برنامه ریزی کرد.به عنوان مثال بعد از اینکه interrupt handler کارش تمام شد.

به صورت کلی software handler ها برای کمینه کردن interrupt latency از دو متد nested interrupt handler و prioritization استفاده می کنند. متد اول اجازه می دهد که وقفه های دیگر حتی در حالی که یک وقفه در حال رسیدگی است ایجاد شوند و متد دوم هنگام رسیدگی به یک وقفه، وقفه هایی که اولویت یکسان یا کمتری از وقفه فعلی دارند را نادیده می گیرد. به این صورت وقفه ها با اولویت بالاتر interrupt latency پایین تری خواهند داشت.

توضیح مراحل پیاده سازی زمان بندی بازخوردی چندسطحی:

برای پیاده سازی این نوع از زمانبند کاری که انجام میدهیم این است که به تابع scheduler مراجعه می کنیم و کد این بخش را بنا بر هر سطح تغییر میدهیم. برای پیاده سازی این قسمت نیاز به تغییراتی در استراکت proc داریم.

```
// 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;     // swtch() 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)


  int qnum;                    // process level
  int arrival_time;            // time of arrival
  int cycles;                  // cycles executed
  int HRRN_priority;
  int wait_cycles;
};
```

در استراکت روبرو 5 ویژگی اضافه شده است که در صف های مختلف از آن استفاده شده است.

اولین مورد که اضافه شده است شماره صف پراسس می باشد.

دومین مورد زمان رسیدن پراسس است.

سومین مورد تعداد سایکلی است که یک پراسس اجرا شده است.

چهارمین ضریب MHRRN است که کاربر به ما می دهد و ما برای تعیین اولویت در سیاست MHRRN از آ ن استفاده می کنیم.

پنجمین مورد هم تعداد سایکلی است که پراسس منتظر میماند تا اجرا شود.

```
482   void
483   scheduler(void)
484   {
485     struct proc *p;
486     struct cpu *c = mycpu();
487     c->proc = 0;
488
489     for(;;){
490       // Enable interrupts on this processor.
491       sti();
492
493       // Loop over process table looking for process to run.
494
495       acquire(&ptable.lock);
496       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
497         if(p->state != RUNNABLE)
498           continue;
499
500         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
501           if(p->wait_cycles >= 8000 && p->state == RUNNABLE) {
502             p->wait_cycles = 0;
503             p->qnum = 1;
504           } else if(p->state == RUNNABLE)) {
505             p->wait_cycles++;
506           }
507         }
508
509         int flag = 0;
510         p = findProcess(&flag);
511         c->proc = p;
512         switchuvm(p);
513         p->state = RUNNING;
514         p->cycles++;
515         p->wait_cycles = 0;
516         swtch(&(c->scheduler), p->context);
517         switchkvm();
518
519         // Process is done running for now.
520         // It should have changed its p->state before coming back.
521         c->proc = 0;
522       }
523       release(&ptable.lock);
524
525     }
526   }
```

به توضیح تابع scheduler می پردازیم:

اگر پراسسی برای ران شدن داشته باشیم برای انتخاب کردن پراس مناسب برای اجرا پیش می رویم.

در این قسمت aging طراحی شده است. در حلقه خط 500 اگر تعداد سیکلی که فرد منتظر مانده است(wait cycles) بیشتر از 8000 شود اولویت این پراسس به صف اول منتقل می شود. در غیر این صورت اگر پراسس منتظر اجرا باشد، یک واحد به wait cycle اضافه می شود.

در مرحله بعد با استفاده از تابع find_process پراسس مناسب برای اجرا را انتخاب میکنیم که بعدا به توضیح منطق این تابع میپردازیم و بعد آن را اجرا می کنیم و این کار در این حلقه دائما تکرار میشود.

توضیحات مربوط به تابع findProcess:

این تابع شامل سه بخش مرتبط به سه الگوریتم هر صف است.

الگوریتم اول که اولویت بیشتری دارد اول بررسی میشود. به این صورت که اگر پراسسی در صف اول وجود داشته باشد با منطق RR انتخاب میشوند و اجرا میشوند. منطق RR به راحتی با تکه کد روبرو پیاده سازی می شود.

هر بار که پراسسی برای اجرا پیدا شود flag یک می شود و پراسس return میشود.

```
struct proc* findProcess(int* flag) {
  struct proc *p;
  //Round Robin
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == RUNNABLE && p->qnum == 1) {

      *flag = 1;
      return p;
    }
```

الگوریتم دوم الگوریتم LCFS می باشد.

این الگوریتم پراسسی را انتخاب میکند که دیرترین arrival time را دارد و آن را بازمیگرداند.

```
  //LCFS
  struct proc *last_p = p;
  int min_arrival_time = 86401;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == RUNNABLE && p->qnum == 2)
      if(p->arrival_time < min_arrival_time) {
        min_arrival_time = p->arrival_time;
        last_p = p;
      }
  if(min_arrival_time != 86401) {
    *flag = 1;
    return last_p;
  }
```

الگوریتم سوم الگورریتم HRRN است.

برای هر پراسس ضریب مخصوص MHRRN محاسبه میشود و پراسس با بیشترین ضریب برای اجرا بازگردانده می شود.

```
  //HRRN
  int curr_time = getTime();
  int max_MHRRN = -1;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == RUNNABLE && p->qnum == 3) {
      int MHRRN = (((curr_time - p->arrival_time + p->cycles)/(p->cycles))+p->HRRN_priority)/2;
      if(MHRRN > max_MHRRN) {
        last_p = p;
        max_MHRRN = MHRRN;
      }
    }
  }
  if(max_MHRRN != -1) {
    *flag = 1;
    return last_p;
  }
  *flag = 0;
  return p;
}
```

حال به توضیح فراخوانی های سیستمی پیاده سازی شده می پردازیم.

به علت شلوغ نشدن گزارش از تغییرات داده شده برای پیاده سازی سیستم کال ها خودداری میکنیم و تنها به توضیح منطق تابع اصلی و نمونه اجرای آن می پردازیم.

تغییر صف پردازه:

یک pid و یک target queue به عنوان ورودی داده میشود و صحت ورودی چک میشود.

سپس در ptable به دنبال پراسس با این pid میگردد و سپس queue number آن را تغییر میدهد.

در صورتی که ورودی درست نباشد و target queue پیدا نشود 1- بازگردانده میشود.

```
int changeQueue(int pid, int tqnum) {
  struct proc *p;
  if (tqnum < 1 || tqnum > 3)
    return -1;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->qnum = tqnum;
      return 0;
    }
  }
  return -1;
}
```

کدی که برای فایل sysproc.c نوشته شده است:

```
int sys_changeQueue(void) {
  int pid, queue;
  if (argint(0, &pid) < 0)
    return -1;
  if (argint(1, &queue) < 0)
    return -1;
  return changeQueue(pid, queue);
}
```

برنامه ای که برای تست عملکرد نوشته شده است:

```
int main(int argc, char *argv[])
{
    int arg1, arg2;

    if(argc > 2) {
        arg1 = atoi(argv[1]);
        arg2 = atoi(argv[2]);
    }
    else {
        printf(1, "Insufficient inputs\n", sizeof("Insufficient inputs\n"));
        exit();
    }

    changeQueue(arg1, arg2);

    exit();
}
```

نتیجه ی اجرای سیستم کال بالا در qemu:

```
000     $ changeQueue 9 3
$ printProcess
name    pid     state           queue   cycles  arrival time    HRRN    MHRRN
init    1       SLEEPING        1       30      0               2002    1001
h       2       SLEEPING        1       32      5               1876    938
oo      6       RUNNING         2       56894   4482            1       0
oo      5       SLEEPING        1       9       4481            6173    3086
oo      10      RUNNABLE        2       54198   5837            1       0
oo      9       SLEEPING        3       3       5836            18067   9033
rintProcess     13      RUNNING         1       2       56034           2001
```

مقدار دهی پارامتر MHRRN در سطح پردازه:

یک pid و یک priorioty به عنوان ورودی داده می شود. در جدول پراسس ها میگردیم تا pid پراسس مورد نظر را پیدا بکنیم سپس priority آن را عوض میکنیم.

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

```
int sys_changeProcessMHRRN(void) {
  int pid, priority;
  if (argint(0, &pid) < 0)
    return -1;
  if (argint(1, &priority) < 0)
    return -1;
  return changeProcessMHRRN(pid, priority);
}
```

برنامه ای که برای تست عملکرد نوشته شده است:

```
int main(int argc, char *argv[])
{
    int arg1, arg2;

    if(argc > 2) {
        arg1 = atoi(argv[1]);
        arg2 = atoi(argv[2]);
    }
    else {
        printf(1, "Insufficient inputs\n", sizeof("Insufficient inputs\n"));
        exit();
    }

    changeProcessMHRRN(arg1, arg2);
    exit();
}
```

نتیجه ی اجرای سیستم کال بالا در qemu:

```
$ printProcess
name    pid     state           queue   cycles  arrival time    HRRN    MHRRN
init    1       SLEEPING        1       17      0               92      46      s
h       2       SLEEPING        1       21      3               74      37      f
oo      7       RUNNABLE        2       1043    510             1       0       f
oo      6       SLEEPING        1       10      509             105     52      s
h       5       RUNNABLE        2       2       508             523     261     p
rintProcess     8       RUNNING         1       7       1546            1       0
$ changeProcessMHRRN 5 1000
$ printProcess
name    pid     state           queue   cycles  arrival time    HRRN    MHRRN
init    1       SLEEPING        1       17      0               240     120     s
h       2       SLEEPING        1       25      3               163     81      f
oo      10      RUNNABLE        2       207     3863            1       0       f
oo      6       SLEEPING        1       11      509             324     162     s
h       5       RUNNABLE        2       2       508             1781    1390    p
rintProcess     11      RUNNING         1       1       4069            1       0
$
```

مقدار دهی پارامتر MHRRN در سطح سیستم:

prioriotyٔ به عنوان ورودی داده می شود. در جدول پراسس ها حلقه میزنیم و سپس priority همه ی پراسس ها را عوض میکنیم.

```
int changeSystemMHRRN(int priority) {
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    p->HRRN_priority = priority;
  }
  release(&ptable.lock);
  return 0;
}
```

```
int sys_changeSystemMHRRN(void) {
  int priority;
  if (argint(1, &priority) < 0)
    return -1;
  return changeSystemMHRRN(priority);
}
```

برنامه ای که برای تست عملکرد نوشته شده است:

```
int main(int argc, char *argv[])
{
    int arg1;

    if(argc > 1) {
        arg1 = atoi(argv[1]);
    }
    else {
        printf(1, "Insufficient inputs\n", sizeof("Insufficient inputs\n"));
        exit();
    }

    printf(1, "Calling changeSystemMHRRN\n");
    changeSystemMHRRN(arg1);
    printf(1, "Returned from changeSystemMHRRN()\n");

    exit();
}
```

نتیجه ی اجرای سیستم کال بالا در qemu:

```
$ changeSystemMHRRN 1000
Calling changeSystemMHRRN
Returned from changeSystemMHRRN()
$ printProcess
name    pid     state           queue   cycles  arrival time    HRRN    MHRRN
init    1       SLEEPING        1       18      0               442     721     s
h       2       SLEEPING        1       29      3               274     637     f
oo      13      RUNNABLE        2       511     7432            1       0       p
rintProcess     14      RUNNING         1       1       7942            1       0
foo     5       SLEEPING        1       3       508             2479    1739    $
```

چاپ اطلاعات:

برای تک تک پراسس ها نام پردازه، شماره پردازه، وضعیت، شماره صف، زمان ورود، مقدار ضریب موثر، تعداد سیکلی که پردازنده به آن پردازه اختصاص یافته است و مقدار MHRRN چاپ میشود.

```
int printProcess(void) {
  struct proc *p;
  acquire(&ptable.lock);
  cprintf("name\tpid\tstate\t\tqueue\tcycles\tarrival time\tHRRN\tMHRRN\n");
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == 0) continue;
    cprintf("%s\t", p->name);
    cprintf("%d\t", p->pid);
    switch (p->state)
    {
      case 0:
        cprintf("%s\t\t", "UNUSED");
        break;
      case 1:
        cprintf("%s\t\t", "EMBRYO");
        break;
      case 2:
        cprintf("%s\t", "SLEEPING");
        break;
      case 3:
        cprintf("%s\t", "RUNNABLE");
        break;
      case 4:
        cprintf("%s\t\t", "RUNNING");
        break;
      case 5:
        cprintf("%s\t\t", "ZOMBIE");
        break;

      default:
        break;
    }

    cprintf("%d\t", p->qnum);

    cprintf("%d\t", p->cycles);
    cprintf("%d\t\t", p->arrival_time);
    cprintf("%d\t", ((getTime() - p->arrival_time + p->cycles)/(p->cycles)));
    cprintf("%d\t", (((getTime() - p->arrival_time + p->cycles)/(p->cycles))+p->HRRN_priority)/2);
  }
  release(&ptable.lock);
  return -1;
}
```

```
int sys_printProcess(void) {
  return printProcess();
}
```

برنامه ای که برای تست عملکرد نوشته شده است:

```
int main(int argc, char *argv[])
{
    if(argc != 1) {
        printf(1, "Insufficient inputs!\n", sizeof("Insufficient inputs!\n"));
    }
    printProcess();
    exit();
}
```

نتیجه ی اجرای سیستم کال بالا در qemu:

```
$ foo &
$ printProcess
name    pid     state           queue   cycles  arrival time    HRRN    MHRRN
init    1       SLEEPING        1       30      0               344     172     s
h       2       SLEEPING        1       28      5               368     184     f
oo      6       RUNNING         2       7152    4482            1       0       f
oo      5       SLEEPING        1       9       4481            646     323     f
oo      10      RUNNABLE        2       4456    5837            1       0       f
oo      9       SLEEPING        1       3       5836            1486    743     p
rintProcess     11      RUNNING         1       2       6292            2001    1
```