اعضا : سید احمد رکنی حسینی ۸۱۰۱۰۰۱۵۴ محمدعلی شاهین فر ۸۱۰۱۰۰۱۶۹ امیرحسین راحتی ۸۱۰۱۰۰۱۴۴

سوال اول :

هر هسته هنگام شروع به کار تابع **mpmain** را صدا می‌زند. این تابع نیز در انتها تابع **scheduler** را صدا می‌زند که عملیات زمانبندی را انجام میدهد . تابع **scheduler** به دنبال پردازه قابل اجرا میگردد و پس از تغییر حافظه به حافظه پردازه توسط تابع **switchuvm**، با استفاده از تابع **swtch** ، عملیات تعویض متن را انجام می‌دهد. این تابع، رجیسترهای **context** قدیمی (**struct context *scheduler**) را در آدرس مربوط به همان **context** ذخیره می‌کند و رجیسترهای مربوط به **context** جدید را از آدرس مربوط به همان **context** بازیابی می‌کند که با این کار، **program counter** نیز به مقدار متناظر آن در **context** جدید تبدیل می‌شود و اینگونه پراسس جدید از وضعیت قبلی اجرا میشود . در ۳ حالت زیر، پردازه در حال اجرا تابع **sched** را فراخوانی می‌کند:

پردازه فراخوانی سیستمی **sleep** را صدا میزند یا با فراخوانی سیستمی **exit** خاتمه میابد یا بوسیله **interrupt** تایمر پردازنده از آن گرفته میشود و تابع **yield** فراخوانی میشود .

در نهایت در تابع **sched**، مجددا عملیات تعویض متن صورت می‌پذیرد و در این حالت **context** ای که در استراکت **cpu** (**struct context scheduler***) بازیابی می‌شود و **context** مربوط پردازه در حال اجرا ذخیره می‌شود. پس از بازیابی **context** مربوط به **scheduler**، تابع **scheduler** دوباره به ادامه کار خود میپردازد . پردازه ای که وظیفه آماده به کار کردن **cpu** را دارد ، همیشه در تابع **scheduler** وجود دارد و فقط زمانی که زمانبندی انجام شده و پردازه دیگری در حال اجرا هست ، در **cpu** فعال نیست

سوال دوم :

صف اجرا در لینوکس توسط یک **red-black tree** پیاده‌سازی می‌شود. در چپ‌ترین گره این درخت، پردازه‌ای قرار گرفته که کمترین اسلایس زمانی در حین اجرا را داشته است.

سوال سوم :

هر پردازنده زمان‌بند خودش را دارد ولی در **xv6** فقط از یک صف زمان‌بندی برای همه پردازنده‌ها به طور مشترک استفاده می‌شود.

این صف از **struct proc**ها حداکثر **64** پردازه همزمان را می‌تواند در خود نگه دارد.

برای جلوگیری از تغییر همزمان روی لیست پردازه ها ، از **spinlock** استفاده میشود به گونه ای که هرگاه قرار است روی لیست تغییری ایجاد شود ، آن را **lock** میکنیم و بعد اتمام کار آن را غیرفعال میکنیم.

داشتن فقط یک صف برای همه پردازنده‌ها، پیاده‌سازی را کمی ساده‌تر می‌کند ولی در عوض به **lock** نیاز دارد که می‌تواند کمی بر روی سرعت نیز تأثیر بگذارد.

از آنجا که پردازه‌ای که در این صف است، هر بار در یک پردازنده اجرا می‌شود و بین آنها جهش می‌کند، با توجه به اینکه هر پردازنده **high-level cache** خودش را دارد، کارایی **cache** بسیار کمتر می‌شود.

در لینوکس، هر پردازنده صف زمان‌بندی مخصوص خودش را دارد و پردازه‌ها به صورت مجزا در آنها قرار می‌گیرند.

در این مدل نیاز داریم که **load** به صورت متوازن بین همه پردازنده ها پخش شود .

سوال چهارم :

زمانی که قفل **ptable** فعال می‌شود، تمامی **interrupt** ها به وسیله تابع **pushcli** غیرفعال می‌شوند. در این حالت ممکن است که تعدادی از پردازه ها در حالت **runnable** نباشند . در این حالت هیچ پردازه دیگری اجرا نمی‌شود و اگر **interrupt**ها نیز هیچ وقت فعال نشود، پس از پایان عملیات **I/O** نمی‌توانیم پردازه‌های مربوطه را به حالت **runnable** تغییر دهیم که بتوانند اجرا شوند، بنابر این سیستم در حالتی قرار میگیرد که شاید هیچ پردازه ای در وضعیت اجرا قرار نگیرد. به همین دلیل است که در این حلقه قبل قفل کردن لیست پردازه ها ، وقفه‌ها فعال می‌شوند تا در صورت نیاز بتوانیم حالت پردازه‌ها را تغییر دهیم.

سوال پنجم :

در لینوکس دو حالت برای مدیریت آن ها داریم :

FLIH : وقفه های مهمتر را در کوتاه ترین زمان ممکن بررسی میکند. یا آن وقفه را به طور کامل هندل میکند یا اطلاعات ضروری آن را برداشته و یک وقفه از نوع **SLIH** ایجاد میکند که آن را مدیریت کند .

SLIH : وقفه های زمانبر در این حالت مدیریت میشوند . این کار مانند یک پردازه انجام می‌شود. **SLIH**ها یا یک ریسه مخصوص در سطح کرنل برای هر هندلر دارند، یا توسط یک **thread pool** مدیریت می‌شوند. آن ها در یک صف مخصوص قرار میگیرند و برای هندل شدن زمانبندی هم روی آن ها صورت میگیرد.

زمانبندی multi level

برای پیاده سازی این ساختار ، ابتدا نیاز است تغییراتی در **struct proc** بدهیم :

Struct BJF برای نگه داری مقادیر مربوط به زمانبند **BJF** به صورت زیر تعریف میشود.

متغیر های زیر نیز به **struct proc** اضافه میشوند که تعداد **cycle** و زمان ایجاد پردازه هم ذخیره میشوند

```
struct BJF{
  uint priority;
  uint priority_ratio;
  uint arrival_time;
  uint arrival_time_ratio;
  uint Executed_cycle;
  uint Execute_cycle_ratio;
  uint procces_size_ratio;
  uint process_size;
};
```

```
enum scheduler_enum  {RR,LCFS,BJF};
// Per-process state
struct proc {
  uint cycle;
  enum scheduler_enum scheduler_queue;
  uint create_time;
  struct BJF bjf;
```

زمانبندی Round-Robin

این زمانبند مشابه زمانبند پیش فرض خود **xv6** است و آن را به یک تابع تبدیل میکنیم که پردازه مورد نظر را انتخاب میکند.

این الگوریتم اولین پردازه در صف که جزو صف **RR** را شناسایی و اجرا میکند.

```
struct proc * RR_func()
{
  struct proc *p ;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->state != RUNNABLE || p->scheduler_queue != RR)
      continue;
    if(p->scheduler_queue == RR)
      return p;
  }
  return 0;
}
```

زمانبندی LCFS

این زمانبند آخرین پردازه ای که ایجاد شده و در این صف قرار دارد را شناسایی میکند و آن را به **cpu** میدهد تا اجرا شود.

```
struct proc* LCFS_func(){ // ptable should be acquire befor
  struct proc* p;
  struct proc* first_proc = 0;
  uint first_time = 0;

  for(p = ptable.proc; p < &ptable.proc[NPROC];p++){
    if(p->state != RUNNABLE || p->scheduler_queue != LCFS)
      continue;
    if(first_time < p->create_time){
      first_time = p->create_time;
      first_proc = p;
    }
  }
  return first_proc;
}
```

زمانبندیBJF

این زمانبند آخرین پردازه ای که ایجاد شده و در این صف قرار دارد را شناسایی میکند و آن را به **cpu** میدهد تا اجرا شود.

```
struct proc* BJF_func(){
  struct proc* p ;
  struct proc* best_proc =0;
  uint least_rank = 100000000;
  for(p = ptable.proc; p < &ptable.proc[NPROC];p++){
    if(p->state != RUNNABLE && p->scheduler_queue != BJF)
      continue;
    if(compute_rank(p->bjf) < least_rank){
      least_rank = compute_rank(p->bjf);
      best_proc = p;
    }
  }
  return best_proc;
}
```

مکانیزم Aging

در تابع **aging** که بعد از افزایش **tick** در **trap.c** صدا زده میشود ، میزان **cycle** هر پردازه افزایش میابد و چک میشود که اگر از ۸۰۰۰ بیشتر بود صف پردازه به **round-robin** تغییر میکند.

```
void aging(void){
  struct proc *p;
  for(p= ptable.proc; p < &ptable.proc[NPROC];p++){
    if(p->state!= RUNNABLE)
      continue;
    p->cycle +=0.1;
    if(p->cycle > 8000){
      p->scheduler_queue = RR;
      p->cycle = 0;
    }
  }
}
```

```
50  switch(tf->trapno){
51  case T_IRQ0 + IRQ_TIMER:
52    if(cpuid() == 0){
53      acquire(&tickslock);
54      ticks++;
55      aging();
56      wakeup(&ticks);
57      release(&tickslock);
58    }
59    lapiceoi();
60    break;
```

زمانبندی Round-Robin

این زمانبند مشابه زمانبند پیش فرض خود **xv6** است و آن را به یک تابع تبدیل میکنیم که پردازه مورد نظر را انتخاب میکند.

این الگوریتم اولین پردازه در صف که جزو صف **RR** را شناسایی و اجرا میکند.

```
struct proc * RR_func()
{
  struct proc *p ;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->state != RUNNABLE || p->scheduler_queue != RR)
      continue;
    if(p->scheduler_queue == RR)
      return p;
  }
  return 0;
}
```

زمانبندی LCFS

این زمانبند آخرین پردازه ای که ایجاد شده و در این صف قرار دارد را شناسایی میکند و آن را به **cpu** میدهد تا اجرا شود.

```
struct proc* LCFS_func(){ // ptable should be acquire befor
  struct proc* p;
  struct proc* first_proc = 0;
  uint first_time = 0;

  for(p = ptable.proc; p < &ptable.proc[NPROC];p++){
    if(p->state != RUNNABLE || p->scheduler_queue != LCFS)
      continue;
    if(first_time < p->create_time){
      first_time = p->create_time;
      first_proc = p;
    }
  }
  return first_proc;
}
```

زمانبندیBJF

این زمانبند آخرین پردازه ای که ایجاد شده و در این صف قرار دارد را شناسایی میکند و آن را به **cpu** میدهد تا اجرا شود.

```
struct proc* BJF_func(){
  struct proc* p ;
  struct proc* best_proc =0;
  uint least_rank = 100000000;
  for(p = ptable.proc; p < &ptable.proc[NPROC];p++){
    if(p->state != RUNNABLE && p->scheduler_queue != BJF)
      continue;
    if(compute_rank(p->bjf) < least_rank){
      least_rank = compute_rank(p->bjf);
      best_proc = p;
    }
  }
  return best_proc;
}
```

محاسبه Rank

در این تابع با توجه به فرمول صورت پروژه مقدار رنک محاسبه و برگردانده میشوند. مقادیر مورد نیاز در این تابع در struct BJF در اطلاعات پردازه وجود دارند و یا حین اجرا تغییر میکنند یا در توابعی مثل fork مقدار دهی میشوند.

```
uint compute_rank(struct BJF bjf){
  uint result = 0;
  result += (bjf.priority * bjf.priority_ratio);
  result += (bjf.arrival_time * bjf.arrival_time_ratio);
  result += (bjf.Executed_cycle * bjf.Execute_cycle_ratio);
  result += (bjf.process_size * bjf.procces_size_ratio);
  return result;
}
```

تغییرات تابع scheduler

در این تابع ، زمانبندی RR به صورت یک تابع در آمده و ابتدا بررسی میشود که آیا در صف RR پردازه ای وجود دارد یا خیر . اگر وجود نداشت به سراغ صف LCFS و اگر خالی بود به سراغ صفBJF میرویم .

```
413 void scheduler(void)
414 {
415   struct proc *p = 0;
416   struct cpu *c = mycpu();
417   c->proc = 0;
418   for(;;)
419   {
420     sti();
421     acquire(&ptable.lock);
422     p = RR_func();
423     if(!p)
424     {
425       p = LCFS_func();
426       if(!p)
427       {
428         p= BJF_func();
429         if(!p)
430         {
431           release(&ptable.lock);
432           continue;
433         }
434       }
435     }
436     c->proc = p;
437     switchuvm(p);
438     p->state = RUNNING;
439     p->cycle += 0.1;
440     swtch(&(c->scheduler), p->context);
441     switchkvm();
442     c->proc = 0;
443     release(&ptable.lock);
444   }
```

تغییر صف پردازه

در این فراخوانی سیستمی process id و صف مقصد دریافت میشود و پردازه موردنظر به صف مقصد منتقل میشود.این فراخوانی سیستمی در یک برنامه سطح کاربر قابل فراخوانی است.

```
change_queue(int pid,int queue){
  acquire(&ptable.lock);
  struct proc *p;
  for(p= ptable.proc; p< &ptable.proc[NPROC];p++){
    if(p->state == UNUSED)
      continue;
    if(p->pid == pid){
      p->scheduler_queue = queue;
      p->cycle = 0;
      release(&ptable.lock);
      return;
    }
  }
  release(&ptable.lock);
}
```

مقداردهی پارامتر BJF درسطح پردازه

در این فراخوانی سیستمی پارامتر های مقدار دهی BJF یک پردازه دریافت میگردد و آن به مقادیر پردازه خواسته شده assign میکنیم.

```
value_BJF_parameters_process(int pid,
        uint priority_ratio,
        uint arrival_time_ratio,
        uint Execute_cycle_ratio,
        uint procces_size_ratio){
  acquire(&ptable.lock);
  struct proc *p;
  for(p= ptable.proc; p< &ptable.proc[NPROC];p++){
    if (p->state == UNUSED)
      continue;
    if(p->pid == pid){
      p->bjf.priority_ratio = priority_ratio;
      p->bjf.arrival_time_ratio = arrival_time_ratio;
      p->bjf.Execute_cycle_ratio = Execute_cycle_ratio;
      p->bjf.procces_size_ratio = procces_size_ratio;
      release(&ptable.lock);
      return 1;
    }
  }
  release(&ptable.lock);
  return 0;
}
```

مقداردهی پارامتر BJF در سطح سیستم

این فراخوانی سیستمی مشابه فراخوانی قبل است با این تفاوت که مقادیر داده شده به همه پردازه ها assign میشوند و این مقادیر هم بوسیله برنامه سطح کاربر نوشته شده دریافت شده و در فراخوانی سیستمی استفاده میشوند

```
value_BJF_parameters_sys(uint priority_ratio,
        uint arrival_time_ratio,
        uint Execute_cycle_ratio,
        uint procces_size_ratio){
  acquire(&ptable.lock);
  struct proc *p;
  for(p= ptable.proc; p< &ptable.proc[NPROC];p++){
    if (p->state == UNUSED)
      continue;
    p->bjf.priority_ratio = priority_ratio;
    p->bjf.arrival_time_ratio = arrival_time_ratio;
    p->bjf.Execute_cycle_ratio = Execute_cycle_ratio;
    p->bjf.procces_size_ratio = procces_size_ratio;
  }
  release(&ptable.lock);
  return 1;
}
```

چاپ اطلاعات

در این قسمت با اضافه کردن سیستم کال **monitor** مقادیر خواسته شده را برای هر پردازه چاپ میکنیم.

در این سیستم کال برای هر پردازه برخی اطلاعات مثل آیدی ، استیت ، زمان ایجاد ، تعداد **cycle** ها و ... چاپ میشوند .

```
for(p= ptable.proc; p< &ptable.proc[NPROC];p++)
{
  if(p->state == UNUSED)
    continue;

  rank = compute_rank(p->bjf);
  cprintf("%s     %d     %s     %d     %d     %d     %d     %d     %d     %d     %d     %d\n"
   , p->name , p->pid , states[p->state], p->scheduler_queue, p->cycle, p->bjf.arrival_time,
    p->bjf.priority, p->bjf.priority_ratio, p->bjf.arrival_time_ratio, p->bjf.Execute_cycle_ratio,
    p->bjf.procces_size_ratio, rank);
  cprintf("-----------------------------------------------------------------\n");
}
return 1;
```

دو برنامه سطح کاربر با نام **foo** و **monitor** داریم که با اجرای آن ها به ترتیب میتوان چند پردازه ایجاد کرد و لیست پردازه ها را مشاهده کرد.

```
int main()
{
    for (int i = 0; i < 3; i++)
    {
        int pid = fork();
        if (pid > 0)
            continue;
        if (pid == 0)
        {
            sleep(1000);
            for (int j = 0; j < 500 ; j++)
            {
                long x = 0;
                for (long k = 0; k < 10000000000; k++)
                    x++;
            }
            exit();
        }
    }
    for(int i = 0 ; i < 3 ; i++)
        wait();
    exit();
}
```

monitor.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       procs_monitor();
9       exit();
10  }
```

در **foo** تعدادی پردازه با محاسبات زیاد ایجاد میشود تا به اندازه کافی طول بکشد و فرصت اجرای دستورات و سیستم کال ها و مشاهده وضعیت پردازه ها را داشته باشیم

لینک repository در GitHub :

https://github.com/muhammadali-shahinfar/osLab3/tree/master

شناسه آخرین commit :

1fd8724a7cb1fd161ef1546580e113325e7ce7a5